What to Do Next?
Analysing and Optimising System Behaviour in Time

Anton Wijs

Printed by Ponsen & Looijen B.V.
ISBN 978-90-6464-174-9
IPA Dissertation Series 2007-13

Typeset with LaTeX2ε using the New Century Schoolbook, 10pt

No Zebra Finches were harmed during the development of this thesis.

For the cover image:
© ≫Unfold II≪, 100 x 100 cm, diasec, 2006 by sascha weidner

The research in this thesis has been carried out at the Centre for Mathematics and Computer Science (CWI), under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was carried out in project 612.064.205 granted by the Dutch organization for Scientific Research (NWO), on "Tools and Techniques for Integrating Performance Analysis and System Verification".

VRIJE UNIVERSITEIT

What to Do Next?

Analysing and Optimising System Behaviour in Time

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 2 oktober 2007 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Antonius Jacobus Wijs

geboren te Goes

promotoren: prof.dr. W.J. Fokkink
prof.dr. J.C. van de Pol

Contents

List of Algorithms

List of Figures

List of Tables

Preface

THIS THESIS DEALS WITH THE incorporation of timing aspects in (mostly) explicit state model checking. Therefore, time as a concept appears everywhere in the thesis, whether as an aspect of timed modelling languages, as a notion of cost in weighted state spaces, or encoded in timed labelled transition systems. Besides that, different notions of time are also represented in the work; it considers both discrete relative time, where periods of time can be divided into atomic time intervals and moments in time are always referred to in relation to earlier moments in time, and continuous absolute time, where no atomic time interval can be identified, and moments in time are referred to with respect to a global clock.

There is, however, one more aspect of time present in the thesis, but only implicitly, and only visible, often partially, to people involved with the work in between these covers. This book is more than just a list of chapters, and more than the contents, at least to me. It embodies a four year period of work at CWI, a period which I enjoyed very much, and as I flip through this thesis, memories come flooding back to me. Although I can hardly cover them all, in this preface, I will try to make this very personal aspect of time somewhat more explicit to the reader.

For example, the work on translating χ_t to μCRL is, besides appearing in the first chapters, also chronologically speaking the first subject I looked at within those four years. I remember being introduced to Elena Bortnik and Nikola Trčka in Eindhoven, and the three of us being assigned to deal with a turntable case study, in order to get familiar with model checking. Little did we know that this small case study would, almost literally, haunt us for the complete first year. I say haunt, since on a few occasions, it even caused me to dream at night about products being rotated, drilled, and tested. Luckily it turned out that every product eventually leaves the table, and likewise, the turntable eventually left our focus. Anyway, I enjoyed working with Elena and Nikola very much, and wish to thank them here.

The first part is concluded by a chapter, which stems from, funnily enough, work created almost at the very end of my PhD period. In a way, it therefore completes a circle. This particular chapter is one of my personal favourites, as I feel that I was only able to write this chapter because I had done the work encompassed by Parts II and III. Though these parts might, at first sight, not appear to be that related to that chapter, writing them helped me to get more into touch with aspects such as the inclusion of time in process algebras, and the design of search algorithms.

Part II is the result of very different events, at different locations. Chronologically, the whole part grew out of one particular moment, which I can still remember well. I was looking into pruning techniques, as Mohammad Torabi Dashti appeared in my room, asking whether it would be possible in the μCRL toolset to associate priority values with action labels, allowing a search algorithm to only traverse transitions whose action labels have high enough priorities. Right at that moment, I was implementing that very mechanism in the toolset. What followed was a collaboration, resulting in papers on beam search, which form the basis of Chapters 8 and 9, and papers on partial order reduction. It was an enjoyable experience, and I thank him for this.

Chapter 7 of Part II is based on my contribution to a, yet to appear, survey paper on the area of Directed Model Checking, which grew out of a Dagstuhl seminar on Directed Model Checking. I thank the people involved in that seminar for inviting me, and Husain Aljazzar, Dragan Bošnački, Stefan Edelkamp, Ansgar Fehnker, and Viktor Schuppan in particular, with whom I wrote the survey paper. Both the seminar and the writing of the survey paper were very helpful for me to get into the subject of Directed Model Checking. This experience finally helped me to create Chapter 6, which I associate with Braga, Portugal, as the basis of it was written by me in between and after the talks at the TACAS 2007 conference.

For Part III of the thesis, I thank my supervisor Wan Fokkink and co-author Jun Pang. In fact, I thank Wan for all his supervision, but specifically the theory of timed bisimilarities was a subject which I would not have been able to deal with without him. I am glad that I got involved in this research, which happened when I was explaining my initial ideas for modelling time in μCRL to Wan. Part III forms, in my opinion, a nice theoretical addition to this otherwise practically oriented thesis.

I thank all my colleagues in the TIPSy project which I have not mentioned already, namely Jos Baeten, Koos Rooda, Asia van de Mortel-Fronczak, Bas Luttik, and Ralph Meijer, for all the useful discussions at the regular meetings, and Bert van Beek for the discussions on χ. Of course, I thank Jaco van de Pol, my other supervisor, for his help, mostly concerning Part II. His comments pushed me to go further in my research. I thank all my former colleagues at CWI, in particular Bert Lisser, who was always there to implement new techniques and explain the inner workings of the μCRL toolset, and Jens Calamé, for helping to solve the occasional LaTeX2ε problem, and with whom I could by now write a paper concerning the social behaviour of the rose-ringed parakeet.

Thanks go out to the members of the reading committee, namely Jos Baeten, Stefan Edelkamp, Radu Mateescu, and Jan Willem Klop. Their comments were very helpful, and resulted in a further improvement of the thesis.

I thank my friends Coen, Sander, Tim, and Wouter. It is intriguing to see how we are all doing such different things with the knowledge we gained during our study. I thank Marjolein for being a good friend. And I thank my parents for their constant support.

Anton Wijs
Auckland, New Zealand, July 2007

Chapter 1

Introduction

> *The past is but a beginning of a beginning, and all that is and has been is but the twilight of a dawn.*
>
> (*H.G. Wells*)

1.1 The Basic Concepts

MODEL CHECKING IS THE PROCESS of checking whether a given *model* satisfies a given *logical formula*, as Wikipedia tells us (the italics are placed by the author). In this description, particularly the terms *model* and *logical formula* attract our attention. A model, first of all, usually represents the behaviour of a system in real life. Such a system may be a piece of technology, e.g. a mobile phone or a car, but it can as well be something else, such as an organism (the usual kind of subject in the area of bio-informatics). By creating a model of a system, we obtain a 'map', so to say, of all possible (re)actions of that system. For the sake of presentation, let us consider a detective, who has to solve a murder mystery. The detective, together with his environment, let us say, the house where the murder was committed, and the suspects present in the house, constitute the system here. More specifically, one might note that they constitute a so-called *concurrent system*, in which the detective, the house, and the suspects represent co-existing agents who may communicate with each other. As a model of this system, we have a so-called 'interactive novel'. Reading the novel should commence at page one, as is the case for any usual book, but at some point the reader may hit a decision point in the story; at such a point, the reader, who is usually identified as the protagonist in the story, i.e. the detective, must decide what to do next. The following example shows in which fashion such a decision point is presented:

> You approach a door. What to do next?
>
> - If you want to open the door, continue reading at page a;
> - If you want to ignore it, continue reading at page b.

Given such a choice (always between a finite number of alternatives), the reader might either continue reading at page a, b, etc., from where the story continues until the next decision point is reached. One can imagine that the book contains many decision points, and thus allows the reader considerable freedom to decide how the story develops. In fact, let us assume that the book covers all possible unfoldings of events.

The second highlighted term in the description of model checking which started this chapter, is *logical formula*. Such a formula describes (usually undesired) behaviour of the system to be investigated, i.e. a *property*. For instance, considering again our book describing the murder mystery, we may wish to check that “it never happens that a second murder is committed”.

How to check that this undesired behaviour never occurs? In fact, this is an important question in the area of model checking, and, as Part II of this thesis demonstrates, it is one which can be answered in many ways. If we stick to the existing reading structure of the book, one way to read the whole book is by reading all individual alternatives of a decision point up to the next decision point right after each other. Taking the example above, we would continue reading at page a. We read from there until we hit the next choice, let us say at page a'. Now, we move to the second alternative of the earlier decision point, i.e. we move to page b. There we continue until we hit the next choice at page b'. In this manner we go on until all the alternatives at the original decision point have been covered up to one decision point deeper. After that, we move our attention to all the alternatives of the decision point at page a', at page b', etc. Such a reading procedure would be called *breadth-first* reading (searching) in model checking. We cover the decision points in order of the number of decision points between them and the first page. Note that it does not necessarily mean that we need to read the whole book; the moment we discover a second murder we may stop reading. Of course, this reading procedure, in general, does not really imply a good reading experience.

Opposed to that, there is the *depth-first* way of reading the book. Let us just read the book how we are supposed to, but each time we reach a decision point, we write down the page and the decision we make. This gives us a list of decisions on paper while reading. The moment we hit an ending, instead of closing the book, we ‘backtrack’ to the page of the last decision point we encountered. There, we now make a choice different from our first one and read on. We continue to read and backtrack, skipping decision points that we have already covered completely. As with breadth-first reading, this depth-first way of reading also covers all the reachable pages in the book, since the book only has a finite number of pages. Furthermore, also here, we do not need to read the whole book, unless a second murder is never committed, or the second murder is encountered for the first time at exactly the last page considered by us.

Making the connection to the usual systems subjected to model checking mentioned earlier, a system, for instance a mobile phone, might be modelled using a *modelling language*, describing all the possibilities of that phone. From this model, a so-called *state space* can be derived, which should be viewed as a virtual space describing all potential behaviour of the phone. If there is some undesired behaviour of the phone possible, however unlikely it may be that it actually occurs in real life, it will be represented in the state space. If we describe undesired behaviour as a logical formula, for instance "the phone explodes", then we can check whether this appears anywhere in the state space or not, searching it in either a breadth-first, a depth-first, or any other manner. This is what differentiates model checking from *simulation*. A simulation can be compared with reading the interactive novel exactly once, in one of the possible ways, from the beginning to an ending. If the undesired behaviour is not found in a simulation, then this is no guarantee that the undesired behaviour can never be encountered in any simulation.

The techniques which allow us to perform model checking, i.e. a modelling language, the derivation of a state space from a given system model, a logical language to express a property in a logical formula, the checking of a logical formula, etc., are called *formal methods*. In short, formal methods are mathematical techniques in order to reason about the correctness of systems. Section 1.3 gives an overview of the main tools using formal methods which have been applied during the research covered in this thesis.

At this point it is relevant to explain how the terms *model checking*, *state space search*, and *state space generation* relate to each other, as all three appear multiple times in this thesis. Model checking, as already explained, involves checking whether a specific situation occurs in a state space. State space search refers to the traversal of a state space. Of course, by means of state space search one can perform model checking, but the main focus of state space search itself is the strategy used to traverse the state space. Finally, state space generation, in the most basic sense, does not focus on searching or checking at all; it merely means the creation of a state space (e.g. printing and binding the interactive novel). Also here, there is a connection with state space search; a state space can be generated by employing the traversal strategy of a state space search. However, note that from a generation point of view, there is no interesting distinction between any two (exhaustive) strategies, as the focus is on the end result of the traversal, which is the state space.

To complete the circle, model checking can be performed both after and during state space generation. The latter way, which is called *on-the-fly* model checking, may often, resource-wise, be preferable over generating a state space first and checking a property later. If the novel is incredibly thick, then we can save a lot of trees by searching the book for a second murder while printing it. If a second murder is found, we can stop printing.[1] Likewise, the state space of a formal model of a concurrent system may be so large that generating and storing it fully on disk takes a lot of time.

[1] This is, of course, a weak part of the used metaphor, since the searching can be done much easier through an electronic version of the book, but the author hopes that the reader gets the point.

1.2 The TIPSy Project

The work in this thesis is the result of working in the NWO project 612.064.205 – “Tools and Techniques for Integrating Performance Analysis and System Verification” (TIPSy). Academic partners in this project were the Formal Methods group at the department of Computer Science of the Technical University of Eindhoven (TU/e), the Systems Engineering group at the department of Mechanical Engineering of the TU/e, and the Embedded Systems group at the Centrum voor Wiskunde en Informatica (CWI). The Dutch company ASML was an industrial partner.

The purpose of the TIPSy project was to combine performance analysis and formal verification. In performance analysis, the focus is on performance aspects of a given system (such as ‘throughput’ and ‘flow time’), aspects which are usually quantifiable. Formal verification, on the other hand, deals with the functional correctness of a system. The main questions here are whether the system does what it is supposed to do, and whether it cannot do what it is not supposed to do. This kind of analysis concerns the quality of the system. Questions asked in formal verification can usually be answered by either ‘yes’ or ‘no’, e.g. “is it impossible for this machine to crash?”.

About ten years ago, the Systems Engineering group at the TU/e developed a modelling language called χ. This language can be used to model manufacturing systems, consisting, for instance, of a manufacturing line or a single machine. Such a model can then be analysed using specific tools. The analysis produces results considering the behaviour of the system, which can then be used to improve the system. In the past, χ specifications were mainly subjected to performance analysis by measurement through simulation. When considering functional verification, however, the usefulness of simulation is rather limited; one may find errors using this technique, but it is not possible to determine the absolute absence of errors. At this point, formal methods can help in improving the analysis.

Recently, χ has been redesigned as a *hybrid* modelling language. With a hybrid language, it is possible to describe systems which exhibit both continuous and discrete dynamic behaviour. The discrete event subset of this language, called *timed* χ or χ_{t}, is comparable with the old version of χ.

One of the goals of the TIPSy project was to integrate χ_{t} with existing formal methods. The common denominator here is *process algebra*; process calculi form a family of algebraic modelling languages for the specification of concurrent systems. These specifications are usually subjected to functional verification. Prominent examples of process calculi are CCS (Milner, 1989), CSP (Hoare, 1985, 1978), and ACP (Bergstra and Klop, 1984b). As the language χ_{t} is based on CSP, the possibility to connect χ_{t} with formal methods available for functional verification was very plausible. In order to achieve such a connection, first, we needed to determine which formal methods actually existed. In the TIPSy project, three PhD students were active: Elena Bortnik, from the Systems Engineering group at the TU/e, Nikola Trčka, from the Formal Methods group at the TU/e, and the current author, from the Embedded Systems group at CWI. We started by investigating the current offer of formal methods. This was done by taking a case study, a turntable system, and trying to model and verify the system using different methods, comparing the experiences with these tools along the way. The majority of the first part of this thesis is a result of this research.

From then on, each of us focussed on different techniques relevant for the project, but still, we collaborated at times and discussed, together with our supervisors, the progress of the project at regular intervals. Naturally, the work of one of these students resulted in the current thesis. The work of the others is briefly described next.

Bortnik further concentrated on translating χ_{t} specifications to timed automata usable in the model checker UPPAAL. She designed a general translation scheme, proved its correctness, and applied it in practice. Together with the current author, she investigated the working of a Clinical Chemical Analyser, a system presented in the second part of this thesis. She contributed in developing techniques to integrate models and implementation artifacts. Such a combination is useful for early analysis of an integrated system, enabling the detection and subsequent prevention of problems before they would actually occur during real integration. Finally, she investigated the possibility to avoid large state spaces by statically analysing specifications, and, in the field of supervisory machine control, made some existing techniques available in the χ toolset to automatically derive a supervisor for a specified system.

Trčka (2007), on the other hand, investigated the possibility to do performance analysis using formal methods through the usage of so-called Markov Chains. He has contributed to the field of Markov processes, in order to improve the possibilities for performance analysis, a practical result of which is the tool 'Markovian Chi'. Besides that, he has solved some problems for the functional verification of timed systems, in particular he has extended the notion of *labelled transition system* (a way to describe a state space) to incorporate state labels, discrete time steps, and successful termination. He has defined a language to generate these state spaces, and has extended the existing equivalence relation *branching bisimulation*, which is applicable to untimed state spaces, to a version which is applicable to state spaces involving a (relative) notion of time. Finally, he has made a connection between the theory of transition systems and the theory of Markov processes, by describing the first theory in the setting of the second, i.e. using matrix theory.

1.3 Model Checking Tools

As mentioned earlier, formal methods are mathematical techniques in order to reason about the correctness of systems. One way in which formal methods are available in practice is through a large number of model checking tools. In this section, we introduce the ones most often referred to in the current work. Besides these, at times in subsequent chapters, other tools will be mentioned and presented.

μCRL toolset A toolset, described, for instance, by Blom et al. (2001) and Wouters (2001), which accepts specifications written in the μCRL modelling language (Groote and Ponse, 1995). The language, which is based on the process algebra ACP extended with equational abstract data types (Loeckx et al., 1996), will be described in more detail in Chapter 2. The main purpose of the toolset is the functional analysis of concurrent systems and communication protocols, by means of simulation, model checking and theorem proving (i.e. proving mathematical theorems by means of a computer program). Most of these techniques rely on explicit state space generation. The verification environment of μCRL together with the model checker CADP (to be described next), which can serve as a back-end to the toolset, have been used to analyse for instance an in-flight data acquisition unit (Fokkink et al., 2002), a distributed system for lifting trucks (Groote et al., 2003), a sliding-window protocol (Badban et al., 2005), and a cache-coherence protocol (Pang et al., 2007).

As case studies get bigger, the limits of the available memory and computation power of a single workstation become apparent. In order to deal with this, most of the techniques in the toolset have been adapted for usage in a *distributed* setting, i.e. a setting where several workstations, forming a so-called cluster, perform the analysis together. Blom and Orzan (2005) developed the distributed state space minimisation techniques, which can be used to minimise a state space modulo strong or branching bisimulation. An example of distributed analysis is the automatic detection of strongly connected components in state spaces, which can be done using a tool created by Orzan and Van de Pol (2005). Blom et al. (2007) provide an overview of using the distributed techniques of the μCRL toolset for a number of case studies, in a number of application areas, such as security, scheduling, testing, and game solving. The usefulness of the toolset for scheduling purposes is one of the subjects of Part II of the thesis.

CADP The *Construction and Analysis of Distributed Processes* (CADP) toolbox (Garavel et al., 2002), formerly known as the Cæsar Aldébaran Development Package, is a toolbox for the design of communication protocols and concurrent systems. For this toolbox, specifications may be written in either LOTOS (Bolognesi and Brinksma, 1987), finite state machines, or μCRL. It includes several equivalence checking tools for the comparison and minimisation of state spaces modulo bisimulation relations. Besides that, it is, for example, possible to check properties of a system, either on-the-fly or not, perform symbolic verification, and compositionally minimise a state space. It also incorporates distributed analysis techniques (Garavel et al., 2006). Finally, some additional functionalities are provided, such as visual checking and performance analysis.

SPIN SPIN (Holzmann, 1997, 2004) is a model checker developed in the USA. Its input language for the specification of systems is called PROMELA, which is derived from the *C* programming language (Kernighan and Ritchie, 1988), also using the communication primitives of CSP (Hoare, 1985), and control flow statements based on the guarded command language (Dijkstra, 1976). With SPIN, it is possible to do simulation and (on-the-fly) model checking. Most of the model checking is done in a (bounded) depth-first manner, either exhaustively or using an efficient approximation method. Some basic data types are supported by the tool.

UPPAAL UPPAAL (Behrmann et al., 2004; Larsen et al., 1997) is a model checker using real-valued clocks for the validation and verification of real-time systems, modelled as networks of timed automata. These automata are presented graphically by means of a well-developed and documented graphical user interface. The tool supports a number of basic data types. Its name is derived from the two sites where the tool is constructed, namely the Uppsala University in Sweden and the Aalborg University in Denmark. All the available techniques are focussed on symbolic model checking, i.e. they deal with symbolic states, which are regions of states in a state space. In other words, the individual states are not dealt with explicitly. It is possible to do both random and directed simulation, and on-the-fly verification. While generating a state space, UPPAAL has several techniques to perform abstraction and symmetry reduction. More recently, a spin-off tool, called UPPAAL CORA, has been developed (Behrmann et al., 2005), which incorporates specific techniques to deal with scheduling problems.

1.4 Contributions and Structure of the Thesis

How the majority of the work in this thesis relates to the basic concept of model checking described earlier, can best be described by returning to our interactive novel. Let us consider the metaphor in a new way in order to explain this relation. Say that, for a change, we are interested in some *desired* behaviour of the system concerning the murder mystery, for instance “the murderer is caught”.[2] On top of wishing to know whether this happens at all in the novel, we wish to find a *trace*, i.e. a storyline running from the first page to the murderer being caught, which takes the smallest amount of time to read. If we consider a reader who reads at a constant speed, then we can roughly express the time needed to read a storyline as the number of lines in the storyline. The question “what to do next?” in the example given earlier now literally refers to a point in the future. The inclusion of time in state spaces raises a number of issues for model checking; for instance, how can state spaces involving time be described elegantly by means of a modelling language? And can these state spaces still be described using an untimed language? Part I of the thesis deals with the relation between timed and untimed modelling languages, and the possibilities and impossibilities of expressing timed behaviour using an untimed modelling language. In particular, we provide a

[2] In model checking, this is usually described by stating that “it never happens that the murderer is caught”; if the reader produces a counter-example to this statement, then we know that somewhere in the novel, the murderer is actually caught.

general translation scheme from χ_t to μCRL, by which it is possible to verify χ_t specifications with the techniques available for μCRL. By doing so, we provide a bridge between performance and functional analysis, since χ_t is mainly used for the former kind of analysis, and μCRL for the latter kind. Furthermore, the work provides insight into the conceptual differences and similarities of timed and untimed modelling languages. Following, we show how the verification of χ_t specifications can be done in this manner in practice. Finally, the research led to $\mu\mathrm{CRL}^{tick}$, an extension of μCRL with a notion of time, which can actually be mapped back to the standard μCRL. This allows the specification of timed systems in an untimed setting, the benefit of which is that the methods and tools available for untimed μCRL specifications can be reused for specifications involving time. The approach builds on earlier proposals to model time with an untimed process algebra. The contributions are that in the $\mu\mathrm{CRL}^{tick}$ approach the emphasis is put on ease of use for the modeller, meaning that the modeller does not need to be concerned about the correctness of the time mechanism in a specification, and the incorporation of time jumps of arbitrary size. The verification of timing properties is achieved by extending Linear Temporal Logic (LTL) with timing constraints (subsequently translated into deterministic finite automata) and applying a verification approach previously proposed by Ioustinova (2004). The main contribution of the work is that it provides more insight into an approach not often explored in the field of timed process algebras, in which it is investigated in how far timed behaviour can be expressed using an untimed modelling language.

Returning to the novel, another issue raised when incorporating (reading) time is that this time must now be taken into account when searching the novel; a search method needs to keep track of the number of lines of all the different storylines. In order to do this, sometimes a search might need to go through some parts of the novel multiple times, since they are shared by several storylines, whereas if we are not interested in the number of lines of all the storylines, we can avoid rereading altogether.

The kind of searching described here can be related to searching for an optimal solution for a given *scheduling* problem. If we consider some industrial production process, then it might be an interesting question how the process can produce as many products as possible in a given time interval, or how it can produce a fixed amount of products as fast as possible. In the latter case, the desired situation is “all the necessary products are produced”, and the search tries to find this situation, such that the trace in the state space leading to this situation is as short (time-wise) as possible. The actual solution is then represented by this trace, which describes the actions to take by the process. In the second part of this thesis, we first describe a range of ways to search a state space, also considering searches where additional information is fed to the search beforehand, such that it may go to the more interesting areas first, or even altogether ignore the non-promising areas. Our contribution to the field here, is that we give a uniform presentation, which emphasises the connections between different existing searches, thereby highlighting a framework in which individual searches can be placed. Along the way, we add more and more features to a basic best-first search, ending up with a search algorithm which covers the most prominent searches in the field. We propose a further generalisation, in which a search may consist of a number of so-called *phases*, which introduces compositionality of best-first searches. Next, we describe a setting in which transitions play a central role in directing a search, as opposed to states. The overview of the field is concluded by presenting a glossary with terms related to Directed Model Checking.

Following, we narrow our viewpoint slightly to those searches useful for scheduling with model checkers. We discuss how to model scheduling problems in μCRL, PROMELA, and priced timed automata, consider techniques to solve scheduling problems available in the model checkers SPIN and UPPAAL CORA (sometimes extending them in order to improve efficiency or quality of the solution), and propose additional techniques to solve scheduling problems, which we have implemented in the μCRL toolset. After that, we narrow our viewpoint one more time, and focus on one particular search useful for scheduling, called *beam search*, which restricts the search to those areas in a state space which are estimated to be the most interesting, considering the target of the search. This is done by means of a heuristic function, provided by the user. Originally, beam search stems from the field of Artificial Intelligence, where it is typically applied on search trees. We propose a number of extensions of the search, making it effective when applied on arbitrary state spaces. These extensions yield a spectrum of beam searches, instances of which turn out to be comparable with other prominent search algorithms. To be able to compare searches, we establish a mechanism to do so; searches can be identified by means of a *guiding signature*, and two guiding signatures may be compared using some equivalence notion. We propose two such equivalence notions, namely *strong* and *weak guiding equivalence*.

Next, we explain how both these beam search extensions and another exhaustive search for scheduling can be moved to a distributed setting, allowing these searches to be performed by a cluster of computers. Finally, we present several case studies, one of which is a Clinical Chemical Analyser, which we used to analyse the proposed searches in practice. One could say that Part II provides possible answers for someone wishing to explore a state space and asking “what to do next?”.

A major problem in model checking is the so-called *state space explosion problem*, meaning that a linear growth of the number of processes placed in parallel in a specification leads to an exponential growth of the resulting state space. Adding time to a specification makes this problem even more difficult, often leading to infinite state spaces, such as when an action is allowed to happen at any time. It stands to reason, therefore, that dealing with the state space explosion problem in a timed setting is one of the main concerns in the first and second part of the thesis.

Another new way to consider the metaphor, is by looking at the book as a whole. At this ‘higher’ viewpoint, we may check whether two books are equal or not, meaning that they contain exactly the same set of possible storylines. We can read the two books from the beginning, relating the first page of one book with the first page of the other, and continue reading the two books at the same time, always opening them at pages which we identify as being ‘related’. If we never encounter a situation in one book which is not also present in the other, then we conclude that the two books are ‘equal’.

In model checking, state spaces can be compared using a range of relations, which are often a kind of *bisimilarity*. In practice, these relations are usually used to minimise a state space to a smaller, but equivalent, one, making the system behaviour easier to check. It shows in the literature that when including a notion of time in a state space, as described earlier in this section, constructing useful timed relations to reason about these state spaces, and, moreover, prove these relations to be correctly defined, is a very complex matter. This is the focus of the third part of this thesis, where we look at the properties of *timed branching bisimilarity*. We consider this relation in an absolute, continuous time setting, and show that the existing definition needs to be extended. Then we prove that the extended notion is an *equivalence*, i.e. that it is *reflexive* (a timed process is timed branching bisimilar to itself), that it is *symmetric* (if a timed process p is timed branching bisimilar to a timed process q, then q is also timed branching bisimilar to p), and that it is *transitive* (if p is timed branching bisimilar to q, and q is timed branching bisimilar to a timed process r, then p is also timed branching bisimilar to r). After that, we prove that a so-called *rooted* version of timed branching bisimilarity is a *congruence* over a process algebra with parallelism, successful termination, and deadlock, meaning that if p and q, expressed in this process algebra, are rooted timed branching bisimilar, then the results of any possible algebraic operation according to this algebra on these timed processes are also rooted timed branching bisimilar, i.e. $f(p)$ and $f(q)$ are also rooted timed branching bisimilar if f is a possible algebraic operation of the process algebra.

1.5 How to Read the Thesis

The thesis can be read in a number of ways. First of all, Chapter 2 presents the basic notions of the work, ordered by the way in which an average modeller uses them in model checking. When a modeller wishes to verify a system, he or she first creates a specification using a modelling language, after which a state space is derived from the specification (possibly in a distributed setting, in case the state space is very large), and finally, properties to be checked can be expressed using a temporal logic.

After that, the three parts of the thesis deal with the inclusion of time (and/or cost) into this basic model checking approach. Each part can mostly be read independently from the others; the reader can choose which steps of the model checking approach to focus on. The first part mostly deals with modelling and verifying timed systems, the second part deals with searching (and therefore, indirectly, generating and model checking) a state space including a notion of cost (which may, if appropriate, be seen as a notion of time). Finally, the third part focuses on comparing timed state spaces.

As a final guideline, we describe how the thesis can be read as a reachability problem using a μCRL specification $\mathcal{M}$. In $\mathcal{M}$, $\mathcal{D} = \{\mathbb{B}\}$, $\mathcal{F} = \{\vee, \wedge\}$, $\mathcal{A} = \{\mathit{samenvatting}, \mathit{summary}, \mathit{prelim}, \mathit{partI}, \mathit{partII}, \mathit{partIII}, \mathit{finished}\}$, $\mathcal{C} = \emptyset$, $\mathcal{P} = \{\mathit{Reading}\}$, with *Reading* as displayed in Figure 1.1, and finally, either $\mathcal{I} = \mathit{Reading}(\mathrm{F}, \mathrm{F})$ or $\mathcal{I} = \mathit{Reading}(\mathrm{F}, \mathrm{T})$, depending on the level of expertise of the reader. In the state space resulting from $\mathcal{M}$, the goal is to reach the action *finished*. For the reader to be able to read this 'map', he or she may be forced to read Chapter 2 first, which is anyway advised. Depending on the willingness to spend energy into reading, the reader can choose a path towards *finished*.

$$
\begin{array}{l}
\mathit{Reading}(\mathit{ReadTheSummary} : \mathbb{B}, \mathit{HaveBasicKnowledge} : \mathbb{B}) = \\
\quad \mathit{samenvatting} \cdot \mathit{Reading}(\mathrm{T}, \mathit{HaveBasicKnowledge}) \lhd \text{“Ik prefereer Nederlands”} \rhd \delta + \\
\quad \mathit{summary} \cdot \mathit{Reading}(\mathrm{T}, \mathit{HaveBasicKnowledge}) \lhd \text{“I wish to (re)read the summary”} \rhd \delta + \\
\quad \mathit{prelim} \cdot \mathit{Reading}(\mathit{ReadTheSummary}, \mathrm{T}) \lhd \text{“I wish to (obtain } \vee \text{ refresh my) basic knowledge”} \rhd \delta + \\
\quad \mathit{partI} \cdot \mathit{Reading}(\mathit{ReadTheSummary}, \mathit{HaveBasicKnowledge}) \\
\qquad\qquad \lhd \text{“I want to read about modelling and verifying timed systems”} \wedge \mathit{HaveBasicKnowledge} \rhd \delta + \\
\quad \mathit{partII} \cdot \mathit{Reading}(\mathit{ReadTheSummary}, \mathit{HaveBasicKnowledge}) \\
\qquad\qquad \lhd \text{“I want to read about best-first state space searches”} \wedge \mathit{HaveBasicKnowledge} \rhd \delta + \\
\quad \mathit{partIII} \cdot \mathit{Reading}(\mathit{ReadTheSummary}, \mathit{HaveBasicKnowledge}) \\
\qquad\qquad \lhd \text{“I want to read about comparing timed state spaces”} \wedge \mathit{HaveBasicKnowledge} \rhd \delta + \\
\quad \mathit{finished} \cdot \mathit{Reading}(\mathit{ReadTheSummary}, \mathit{HaveBasicKnowledge}) \lhd \mathit{ReadTheSummary} \rhd \delta
\end{array}
$$

Figure 1.1: How to read the thesis

Chapter 2

Preliminaries

> *They always say that time changes things, but you actually have to change them yourself.*
>
> (*Andy Warhol*)

2.1 The Modelling Language μCRL

THE LANGUAGE μCRL IS BASED on the process algebra ACP (Bergstra and Klop, 1984b), extended with equational abstract data types (Loeckx et al., 1996). It comes with a toolset (Blom et al., 2001) that can build a state space from a specification and store it in the `.aut` format, one of the input formats of the model checker CADP (Garavel et al., 2002). Next to that, in order to strive for precision in proofs, an important research area is to use theorem provers such as PVS (Owre et al., 1992) to help in finding and checking derivations in μCRL. A large number of distributed systems have been verified in μCRL, often with the help of a proof checker or theorem prover (e.g. Badban et al., 2005; Groote et al., 1998).

We will give an overview of the language, including most of its axioms and transition rules, necessary for understanding this thesis. Groote and Ponse (1995), Groote and Reniers (2001), and Wouters (2001) provide more elaborate explanations. For example, the abstraction operator is omitted in the current description, since it is not used in any subsequent chapter. An introduction to process algebra is provided by e.g. Fokkink (2000b).

Definition 1 (μCRL specification). *We define a μCRL specification $\mathcal{M}$ as a sextuple $(\mathcal{D}, \mathcal{F}, \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{I})$, where*

- *$\mathcal{D}$ is the set of data domains used;*
- *$\mathcal{F}$ is the set of functions defined over the data domains in $\mathcal{D}$;*
- *$\mathcal{A}$ is the set of actions used;*

- $\mathcal{C}$ *is the set of communication rules for the actions;*
- $\mathcal{P}$ *is the set of recursive equations in the specification;*
- $\mathcal{I}$ *is the initialisation line, combining and initialising the processes.*

Following is a more detailed description of each element in the sextuple.

Data domains and functions In order to intertwine processes with data, actions and recursion variables can be parameterised with data types. Each specification should start by defining the necessary data types and the functions that work on them. For this purpose, μCRL incorporates the usage of *equational abstract data types*. An equational abstract data type contains the following:

1. A declaration list, starting with the keyword **func**, of constructors. A constructor of a data type is a function with this data type as the target domain. Together, the constructors define the structure of the data type.
2. A declaration list of additional functions (started by the keyword **map**), which are not constructors. Their definitions are given by means of a finite number of variables and equations, in the following lists.
3. A declaration list of variables, indicated by the keyword **var**, used in the definitions in the next list.
4. A list of equations, with the keyword **rew**, defining the constants and functions.

In fact, it is mandatory to define the boolean data type in each specification, since the conditional construct, which is one of the μCRL operators, works with boolean expressions. Let us look informally at the definition of the boolean data type $\mathbb{B}$ with functions '=' (equality) and '$\wedge$' (conjunction). We follow the general description of an equational abstract data type, and start the definition of the data type with the keyword **sort**:

sort $\mathbb{B}$
func $\mathrm{T}, \mathrm{F} :\rightarrow \mathbb{B}$ (T is *true* and F is *false*)
map $=, \wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
var $b : \mathbb{B}$
rew $b = b \triangleq \mathrm{T}$
$\mathrm{T} = \mathrm{F} \triangleq \mathrm{F}$
$\mathrm{F} = \mathrm{T} \triangleq \mathrm{F}$
$\mathrm{T} \wedge b \triangleq b$
$\mathrm{F} \wedge b \triangleq \mathrm{F}$

One can virtually define any data type as an equational abstract data type. For more on equational abstract data types, see e.g. the work of Loeckx et al. (1996). In this thesis, we assume the presence of the domains of the booleans ($\mathbb{B}$) and the natural numbers ($\mathbb{N}$), and definitions of the most common functions, such as disjunction, conjunction, and negation (for the booleans) and addition and multiplication (for the natural numbers). In other words, for a specification $\mathcal{M}$, $\mathbb{B}, \mathbb{N} \in \mathcal{D}$, and, for example, $\wedge, \vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}, \neg : \mathbb{B} \to \mathbb{B}, +, \times : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \in \mathcal{F}$. Furthermore, it is mandatory that equality is defined for each data type, since data elements must be comparable when synchronising actions. Finally, a variable d of type $\mathbb{D}$ is written as $d : \mathbb{D}$.

Actions In μCRL one can declare actions. These actions may have zero, one or several data parameters. We denote actions a, b, etc. appearing in a specification $\mathcal{M}$ as being elements of $\mathcal{A}$. Each atomic action $a(e)$, with $a : \mathbb{D}_a$ and $e \in \mathbb{D}_a$, can execute itself, leading to successful termination (we note that actually type $\mathbb{D}_a$ may be a Cartesian product of data types). This is denoted as $a(e) \xrightarrow{a(e)} \surd$. Only in the state $\surd$ can successful termination occur, which is expressed as $\surd\downarrow$.[1] Finally, the process deadlock (δ), which cannot execute itself, nor terminate successfully, and the internal action τ are predefined, with $\tau, \delta \notin \mathcal{A}$ and $\tau \xrightarrow{\tau} \surd$.

Communication rules It is possible to define communication rules for actions. For instance, for $a, b, c \in \mathcal{A}$, one can define the rule $a \mid b = c$, meaning that a and b can synchronise with each other, forming action c. For every rule $a \mid b = c$, we have $(a, b, c) \in \mathcal{C}$. Communication can only take place, if the data parameters of a and b have the same types and values. Communication in μCRL is commutative, i.e. the order of a and b in the rules of $\mathcal{C}$ does not matter. In other words, $(a, b, c) \in \mathcal{C}$ iff $(b, a, c) \in \mathcal{C}$. Furthermore, communication in μCRL is associative, meaning that $(a \mid b) \mid c = a \mid (b \mid c)$. We return to these communication rules when describing parallel composition.

Recursive specifications The set $\mathcal{P}$, called *recursive specification*, is a finite set of so-called *recursive equations*, describing the behaviour of a system. Before explaining these notions, we first describe the notion *process term*.

Process terms can be created by combining actions from $\mathcal{A}$ and a given set of operators. We call the domain of process terms $\mathbb{P}$. Next, we will give a description of the operators used.

Operators There are four basic operators for creating process terms in μCRL.

1. The alternative composition operator ($+$). A process term $P + Q$ proceeds (non-deterministically) as P or Q (if they can proceed). Differentiating the cases that P or Q successfully terminates or not, this operator gives rise to four transition rules:

[1] An alternative approach present in the literature is the usage of a basic constant for successful termination ϵ. For this, see e.g. Baeten and Reniers (2000) and Baeten (2003).

$$\frac{P \xrightarrow{a(e)} P'}{P + Q \xrightarrow{a(e)} P'} \qquad \frac{P \xrightarrow{a(e)} \surd}{P + Q \xrightarrow{a(e)} \surd}$$

$$\frac{Q \xrightarrow{a(e)} Q'}{P + Q \xrightarrow{a(e)} Q'} \qquad \frac{Q \xrightarrow{a(e)} \surd}{P + Q \xrightarrow{a(e)} \surd}$$

2. The sum operator ($\sum_{d:\mathbb{D}} X(d)$), with $X(d)$ a mapping from domain $\mathbb{D} \in \mathcal{D}$ to process terms, behaves as $X(d_1) + X(d_2) + \ldots$, with $d_1, d_2, \ldots \in \mathbb{D}$, i.e. as the possibly infinite choice between $X(d)$ for any data term d taken from $\mathbb{D}$. This operator is mostly used to describe a process that is reading some input over a data type. See the work of Luttik (2002) for a formal treatment of the sum operator (also known as choice quantification). It is difficult to reason about the sum operator, since it acts as a binder just like the lambda in the lambda calculus (e.g. Barendregt (1984)). In order to avoid having to deal with all the technicalities of substitutions, we say here that the variable X can be instantiated with functions from $\mathbb{D}$ to $\mathbb{P}$, an approach also used by Groote and Reniers (2001). The transition rules of the sum operator can then be stated as follows:

$$\frac{Xd \xrightarrow{a(e)} P'}{\sum_{d:\mathbb{D}} X \xrightarrow{a(e)} P'} \qquad \frac{Xd \xrightarrow{a(e)} \surd}{\sum_{d:\mathbb{D}} X \xrightarrow{a(e)} \surd}$$

3. The sequential composition operator ($\cdot$). A process term $P{\cdot}Q$ proceeds as P, which upon successful termination is followed by Q. These are its transition rules:

$$\frac{P \xrightarrow{a(e)} P'}{P{\cdot}Q \xrightarrow{a(e)} P'{\cdot}Q} \qquad \frac{P \xrightarrow{a(e)} \surd}{P{\cdot}Q \xrightarrow{a(e)} Q}$$

4. The process term $P \lhd b \rhd Q$ where $P, Q \in \mathcal{P}$, and $b : \mathbb{B}$, behaves as P if b is equal to T (true) and behaves as Q if b is equal to F (false). This operator is called the conditional operator. This gives rise to the following transition rules:

$$\frac{P \xrightarrow{a(e)} P' \quad b = \mathrm{T}}{P \lhd b \rhd Q \xrightarrow{a(e)} P'} \qquad \frac{P \xrightarrow{a(e)} \surd \quad b = \mathrm{T}}{P \lhd b \rhd Q \xrightarrow{a(e)} \surd}$$

$$\frac{Q \xrightarrow{a(e)} Q' \quad b = \mathrm{F}}{P \lhd b \rhd Q \xrightarrow{a(e)} Q'} \qquad \frac{Q \xrightarrow{a(e)} \surd \quad b = \mathrm{F}}{P \lhd b \rhd Q \xrightarrow{a(e)} \surd}$$

Returning to the notions of recursive specification and recursive equation, process terms can be used in these, to describe system behaviour. A recursive specification is defined as follows:

Definition 2 (Recursive specification). *A* recursive specification *is a finite set of* recursive equations

$$\begin{array}{rcl} X_1(x_1 : \mathbb{D}_1) & = & t_1(X_1(d_1),\ldots,X_n(d_n)) \\ & \vdots & \\ X_n(x_n : \mathbb{D}_n) & = & t_n(X_1(d_1),\ldots,X_n(d_n)) \end{array}$$

where $\mathbb{D}_i \in \mathcal{D}$ are data type names, the X_i are recursion variables, the $x_i : \mathbb{D}_i$ are variables, not clashing with a function name of arity zero, nor with a parameter-less recursive equation, or action name. Moreover, the $t_i(X_1(d_1),\ldots,X_n(d_n))$ are μCRL process terms with possible occurrences of the recursion variables $X_1,\ldots,X_n$. Finally, data term $d_i \in \mathbb{D}_i$ may contain occurrences of variable x_i.

We note that actually types $\mathbb{D}_i$ may be Cartesian products of data types. A more restricted form of recursive specification is the guarded recursive specification. It is defined as follows:

Definition 3 (Guarded recursive specification). *A recursive specification*

$$\begin{array}{rcl} X_1(x_1 : \mathbb{D}_1) & = & t_1(X_1(d_1),\ldots,X_n(d_n)) \\ & \vdots & \\ X_n(x_n : \mathbb{D}_n) & = & t_n(X_1(d_1),\ldots,X_n(d_n)) \end{array}$$

is guarded *if the $t_i(X_1(d_1),\ldots,X_n(d_n))$ can be represented in the form*

$$\sum a_k(d_{a_k})\cdot t'_k(X_1(d_1),\ldots,X_n(d_n)) + \sum b_m(d_{b_m})$$

with the $a_k(d_{a_k}) \in \mathcal{A} \cup \{\tau\}$, the $b_m(d_{b_m}) \in \mathcal{A} \cup \{\tau\}$, and the $t'_i(X_1(d_1),\ldots,X_n(d_n))$ being μCRL process terms with possible occurrences of the recursion variables $X_1,\ldots,X_n$, by application of the axioms of μCRL (Table 2.1), and replacing recursion variables by the right-hand sides of their recursive equations. Whenever $k = m = 0$, the sum represents δ.

We note that types $\mathbb{D}_i$, $\mathbb{D}_{a_k}$, $\mathbb{D}_{b_m}$ may be Cartesian products of data types. From this point on, when we refer to a *process*, we often mean a (guarded) recursive equation. Processes in $\mathcal{P}$ can be viewed as components in a μCRL specification. Furthermore, we define $en_{\mathcal{A}}(P)$, with $en_{\mathcal{A}} : \mathbb{P} \to 2^{\mathcal{A}}$, to be the set of *enabled* actions of the process term P. It is defined as follows: $en_{\mathcal{A}}(P) = \{a(e) \mid a : \mathbb{D}_a \in \mathcal{A} \wedge e \in \mathbb{D}_a \wedge \exists P'.P \xrightarrow{a(e)} P'\}$.

A *transition* is a triple $(P, a(e), P')$, which means that from process term P an enabled action $a(e)$ can be fired, leading to a process term P', i.e. $P \xrightarrow{a(e)} P'$. The set of transitions of P is referred to as $en_{\mathcal{M}}(P)$. Of course, $en_{\mathcal{M}}(\delta) = \emptyset$.

Initialisation line $\mathcal{I}$ defines the initial state of a μCRL specification. It is often of the following form:

$$\partial_H(X_0(d_0) \parallel \ldots \parallel X_m(d_m))$$

Here $X_0, \ldots, X_m \in \mathcal{P}$ and $\forall 0 \leq j \leq m . X_j : \mathbb{D}_j \wedge d_j \in \mathbb{D}_j$. Note that if $m = 0$, there is only a single process initialised. The following operators are used here:

1. The parallel composition operator ($\parallel$). A process term $P \parallel Q$ executes the actions of P and Q concurrently in an interleaved fashion. Furthermore, for all actions a, b, c, such that $(a, b, c) \in \mathcal{C}$, if one process can execute a and the other one can execute b, then P and Q can synchronise (i.e. $P \parallel Q$ executes the communication action c). These are the transition rules:

$$\frac{P \xrightarrow{a(e)} P'}{P \parallel Q \xrightarrow{a(e)} P' \parallel Q} \qquad \frac{P \xrightarrow{a(e)} \surd}{P \parallel Q \xrightarrow{a(e)} Q}$$

$$\frac{Q \xrightarrow{a(e)} Q'}{P \parallel Q \xrightarrow{a(e)} P \parallel Q'} \qquad \frac{Q \xrightarrow{a(e)} \surd}{P \parallel Q \xrightarrow{a(e)} P}$$

$$\frac{P \xrightarrow{a(e_1)} P' \quad Q \xrightarrow{b(e_2)} Q' \quad (a, b, c) \in \mathcal{C} \quad e_1 = e_2}{P \parallel Q \xrightarrow{c(e_1)} P' \parallel Q'}$$

$$\frac{P \xrightarrow{a(e_1)} \surd \quad Q \xrightarrow{b(e_2)} Q' \quad (a, b, c) \in \mathcal{C} \quad e_1 = e_2}{P \parallel Q \xrightarrow{c(e_1)} Q'}$$

$$\frac{P \xrightarrow{a(e_1)} P' \quad Q \xrightarrow{b(e_2)} \surd \quad (a, b, c) \in \mathcal{C} \quad e_1 = e_2}{P \parallel Q \xrightarrow{c(e_1)} P'}$$

$$\frac{P \xrightarrow{a(e_1)} \surd \quad Q \xrightarrow{b(e_2)} \surd \quad (a, b, c) \in \mathcal{C} \quad e_1 = e_2}{P \parallel Q \xrightarrow{c(e_1)} \surd}$$

2. The encapsulation operator (∂_H). In $\partial_H(P)$ all actions of P that occur in the set $H \subseteq \mathcal{A}$ are disabled. Typically this operator is used to enforce that certain actions

synchronise. We have $en_{\mathcal{A}}(\partial_H(P)) = en_{\mathcal{A}}(P) \setminus H$, i.e. the transition rules are:

$$\frac{P \xrightarrow{a(e)} P' \quad a \notin H}{\partial_H(P) \xrightarrow{a(e)} \partial_H(P')} \qquad \frac{P \xrightarrow{a(e)} \surd \quad a \notin H}{\partial_H(P) \xrightarrow{a(e)} \surd}$$

3. The renaming operator (ρ_f), with $f : \mathcal{A} \to \mathcal{A}$, is suited for reusing a given specification with different action names. The subscript f signifies that each action a must be renamed to $f(a)$. The behaviour of process term $\rho_f(P)$ is obtained by renaming all actions a which are enabled in P to $f(a)$. The transition rules are as follows:

$$\frac{P \xrightarrow{a(e)} P'}{\rho_f(P) \xrightarrow{f(a)(e)} \rho_f(P')} \qquad \frac{P \xrightarrow{a(e)} \surd}{\rho_f(P) \xrightarrow{f(a)(e)} \surd}$$

While not a requirement in theory, in practice, a μCRL specification should not contain any unguarded recursive equations, nor should any of the recursive equations in a μCRL specification (syntactically) contain successful termination. This is because a recursive specification can only be transformed by the μCRL toolset to a so-called *linear process equation* (LPE) (Bezem and Groote, 1994) if this requirement is met. The LPE form lies at the heart of the μCRL toolset; an LPE can be subjected to a range of analyses, such as state space generation and static analysis.

Linear process equations As mentioned already, $\mathcal{P}$ consists of a finite set of recursive equations. It can be seen as the heart of $\mathcal{M}$, since it declares the behaviour of the specified system. Given that a recursive equation in $\mathcal{P}$ is guarded, and does not (syntactically) contain successful termination, it can be transformed into an LPE. The transformation itself is entirely a topic on its own, and therefore not covered here. Notably Usenko (2002a) has described it. In essence, an LPE is a vector of data parameters, together with a list of summands consisting of a condition, action and effect triple, describing when an action may happen and what its effect is on the vector of data parameters. Its form is given in Definition 4.

Definition 4 (Linear process equation[2]). *A* linear process equation *is a guarded recursive equation of the following form:*

$$X(d : \mathbb{D}) = \sum_{i \in I} \sum_{e_i \in \mathbb{D}_i} a_i(f_i(d, e_i)) \cdot X_i(g_i(d, e_i)) \lhd h_i(d, e_i) \rhd \delta$$

[2] In Section 3.4.1, the definition of LPE will be extended to incorporate successful termination. LPEs with successful termination can be used to express fragments of recursive equations.

where I is a finite index set, $\mathbb{D}, \mathbb{D}_i, \mathbb{D}_{a_i} \in \mathcal{D}$*,* $a_i \in \mathcal{A} \cup \{\tau\}$*,* $a_i : \mathbb{D}_{a_i}$*,* $f_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{D}_{a_i}$*,* $g_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{D}$ *and* $h_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{B}$.

Here, the different states of the process, described by the LPE, are represented by the data parameter $d : \mathbb{D}$. Types $\mathbb{D}$ and $\mathbb{D}_i$ may be Cartesian products of data types. The data parameter e_i can influence the parameter of action a_i, the condition h_i and the resulting state g_i. Parameter e_i is typically used to let a read action range over a data domain (i.e. choice quantification). In the future, when writing I^P, we refer to the index set of process P.

Returning to the notion of the set of enabled actions of a process term P (i.e. $en_{\mathcal{A}}(P)$), note that from a recursive equation $X(d)$ and a value $d_0 \in \mathbb{D}$, we can derive the set of enabled actions $en_{\mathcal{A}}(X(d_0))$ by the fact that an action a_i is in this set iff there exists $e_i \in \mathbb{D}_i$, such that $h_i(d_0, e_i)$.

The behaviour of processes can be compared by means of some form of *bisimilarity*. The basic notion is called *strong bisimulation*. It is defined in Definition 5.

Definition 5 (Strong bisimulation). *A binary relation* $R \subseteq \mathbb{P} \times \mathbb{P}$ *is a* strong bisimulation *if* $p\ R\ q$ *implies:*

- *if* $p \xrightarrow{a(e)} p'$ *then* $q \xrightarrow{a(e)} q'$ *with* $p'\ R\ q'$*;*
- *if* $q \xrightarrow{a(e)} q'$ *then* $p \xrightarrow{a(e)} p'$ *with* $p'\ R\ q'$*;*
- *if* $p \downarrow$ *then* $q \downarrow$*;*
- *if* $q \downarrow$ *then* $p \downarrow$.

Two processes p and q are strong bisimilar*, denoted by* $p \leftrightarrows q$*, if there is a strong bisimulation relation R such that* $p\ R\ q$.

Next, we provide all the axioms of μCRL, minus the ones for the abstraction operator, in Table 2.1. Groote and Reniers (2001) originally presented these axioms, and provided a full explanation of them. Axioms A1 to A5 are the basic axioms, A6 and A7 are the axioms for deadlock, and B1 and B2 concern the internal action τ. The axioms for conditionals are C1 and C2. In order to elegantly describe the axioms for the sum operator, we say that the variables x, y and z may be instantiated with process terms, and X and Y can be instantiated with functions from some data type to $\mathbb{P}$. The sum operator $\sum$ expects a function from a data type to $\mathbb{P}$, and $\sum_{d:\mathbb{D}}$ expects a process term. It should be noted that when substituting, no variable may become bound by any of the sum operators. In order to formulate the axioms concerning the parallel composition operator, the left merge operator ($\mathbin{|\!\lfloor}$) is used. It behaves exactly as the parallel composition operator, except that the first action to be executed must come from the left hand side. The core axiom for parallel composition is CM1, which states that a process term $x \parallel y$ either executes an action coming from x, an action coming from y, or an action resulting from a synchronisation of actions in x and y. The axioms of μCRL are sound modulo strong bisimilarity.

Table 2.1: The axioms of μCRL without abstraction

$x + y = y + x$	A1	$x + (y + z) = (x + y) + z$	A2
$x + x = x$	A3	$(x + y)\cdot z = x\cdot z + y\cdot z$	A4
$(x\cdot y)\cdot z = x\cdot(y\cdot z)$	A5	$x + \delta = x$	A6
$\delta\cdot x = \delta$	A7	$c\cdot\tau = c$	B1
$x\cdot(\tau\cdot(y + z) + y) = x\cdot(y + z)$	B2	$x \lhd \mathrm{T} \rhd y = x$	C1
$x \lhd \mathrm{F} \rhd y = y$	C2	$\sum_{d:\mathbb{D}} x = x$	SUM1
$\sum X = \sum X + Xd$	SUM3	$\sum_{d:\mathbb{D}}(Xd + Yd) = \sum X + \sum Y$	SUM4
$(\sum X)\cdot x = \sum_{d:\mathbb{D}}(Xd\cdot x)$	SUM5	$(\forall d : \mathbb{D}(Xd = Yd)) \implies \sum X = \sum Y$	SUM11
$\partial_H(\delta) = \delta$	DD	$\partial_H(a(d)) = a(d)$ **if** $a \notin H$	D1
$\partial_H(a(d)) = \delta$ **if** $a \in H$	D2	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	D3
$\partial_H(x\cdot y) = \partial_H(x)\cdot\partial_H(y)$	D4	$\partial_H(\sum X) = \sum_{d:\mathbb{D}} \partial_H(Xd)$	SUM8
$\partial_H(\tau) = \tau$	DT	$\rho_f(\delta) = \delta$	RD
$\rho_f(\tau) = \tau$	RT	$\rho_f(a(d)) = f(a)(d)$	R1
$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$	R3	$\rho_f(x\cdot y) = \rho_f(x)\cdot\rho_f(y)$	R4
$\rho_f(\sum X) = \sum_{d:\mathbb{D}} \rho_f(Xd)$	SUM10	$x \parallel y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x \mid y$	CM1
$c \lfloor\!\lfloor x = c\cdot x$	CM2	$c\cdot x \lfloor\!\lfloor y = c\cdot(x \parallel y)$	CM3
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	CM4	$(\sum X) \lfloor\!\lfloor x = \sum_{d:\mathbb{D}}(Xd \lfloor\!\lfloor x)$	SUM6
$a(d) \mid a'(e) = \mathcal{C}(a,a')(d) \lhd d = e \rhd \delta$ **if** $\mathcal{C}(a,a')$ defined δ **otherwise**			CF
$\delta \mid c = \delta$	CD1	$c \mid \delta = \delta$	CD2
$\tau \mid c = \delta$	CT1	$c \mid \tau = \delta$	CT2
$c\cdot x \mid c' = (c \mid c')\cdot x$	CM5	$c \mid c'\cdot x = (c \mid c')\cdot x$	CM6
$c\cdot x \mid c'\cdot y = (c \mid c')\cdot(x \parallel y)$	CM7	$(x + y) \mid z = x \mid z + y \mid z$	CM8
$x \mid (y + z) = x \mid y + x \mid z$	CM9	$(\sum X) \mid x = \sum_{d:\mathbb{D}}(Xd \mid x)$	SUM7
		$x \mid (\sum X) = \sum_{d:\mathbb{D}}(x \mid Xd)$	SUM7'

Finally, we show a small example of a μCRL specification, together with a corresponding linearised version. Let us consider a system, where a sender process repeatedly wishes to send a boolean value to a receiver process. The value of the boolean depends on the value of a counter maintained by the sender process. We define a specification $\mathcal{M}$ with $\mathcal{D} = \{\mathbb{N}, \mathbb{B}\}$, $\geq: \mathbb{N} \times \mathbb{N} \to \mathbb{B}, =: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \in \mathcal{F}$, $\mathcal{A} = \{sa : \mathbb{B}, ra : \mathbb{B}, ca : \mathbb{B}\}$, and $(sa, ra, ca) \in \mathcal{C}$. Furthermore, $\mathcal{P}$ consists of the sender process *Send* and the receiver process *Recv* presented in Figure 2.1. The sender process contains non-determinism, since both $sa(\mathrm{F})$ and $sa(\mathrm{T})$ are enabled whenever $i \geq 2$. In the receiver process, the enumeration over the boolean data type allows the synchronisation with both $sa(\mathrm{T})$ and

$sa(\mathrm{F})$. A possible instantiation is the following: $\mathcal{I} = \partial_{\{sa,ra\}}(Send(2) \parallel Recv)$.

$$\begin{aligned} Send(i : \mathbb{N}) = {} & sa(\mathrm{F}) \cdot Send(i+1) \lhd i \geq 1 \rhd \delta + \\ & sa(\mathrm{T}) \cdot Send(i-1) \lhd i \geq 2 \rhd \delta \end{aligned}$$

$$Recv = \sum_{b:\mathbb{B}} ra(b) \cdot Recv$$

Figure 2.1: A small μCRL example of two processes

Next, we show this specification in a linearised form. In such a form, $\mathcal{P}$ consists of a single LPE and parallelism is completely removed from $\mathcal{I}$. Let us call the linearised specification $\mathcal{M}' = (\mathcal{D}', \mathcal{F}', \mathcal{A}', \mathcal{C}', \mathcal{P}', \mathcal{I}')$. First of all, we have $\mathcal{D}' = \mathcal{D}$, $\mathcal{F}' = \mathcal{F}$, $\mathcal{A}' = \{ca\}$, and $\mathcal{C}' = \emptyset$. The only process in $\mathcal{P}$ is the LPE X, displayed in Figure 2.2. In this case, the linearised form follows directly from $\mathcal{M}$, due to the simplicity of the process *Recv*. When dealing with more complex processes, usually the removal of parallelism involves dealing with all possible interleavings of actions. Finally, the initialisation $\mathcal{I}' = X(2)$.

$$\begin{aligned} X(i : \mathbb{N}) = {} & ca(\mathrm{F}) \cdot X(i+1) \lhd i \geq 1 \rhd \delta + \\ & ca(\mathrm{T}) \cdot X(i-1) \lhd i \geq 2 \rhd \delta \end{aligned}$$

Figure 2.2: A small μCRL example of an LPE

2.2 State Space Generation

Definition 6 (State space). *A* state space *or* labelled transition system (LTS) *is a quadruple $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{I})$, where $\mathscr{S}$ is a set of states, often not fully known a priori, $\mathscr{I} \subseteq \mathscr{S}$ is a set of initial states, $\mathscr{A}$ is a finite set of action labels and $\mathscr{T} \subseteq \mathscr{S} \times \mathscr{A} \times \mathscr{S}$ is the transition relation. A transition $(s, \ell, s') \in \mathscr{T}$, denoted $s \xrightarrow{\ell} s'$, indicates that the system can move from state s to s' by performing action ℓ. Here, s' is referred to as a successor state of s.*

In the rest of the thesis, unless explicitly stated, we consider finite state spaces, i.e. state spaces with finite sets of states and transitions.

A state s' is called *reachable* from state s iff $s \rightarrow^* s'$, where $\rightarrow^*$ is the reflexive transitive closure of $\xrightarrow{\ell}$ for any $\ell \in \mathscr{A}$. When checking a *reachability* property, one searches for an $s \in \mathscr{G}$, where $\mathscr{G} \subseteq \mathscr{S}$ is a given set of goal states, such that there exists an $s' \in \mathscr{I}$ for which $s' \rightarrow^* s$.

The set of enabled transitions in state s of state space $\mathscr{M}$ is defined as $en_{\mathscr{M}}(s) = \{t \in \mathscr{T} \mid \exists s' \in \mathscr{S},\ \ell \in \mathscr{A}.\ t = s \xrightarrow{\ell} s'\}$. For $T \subseteq \mathscr{T}$, we define $nxt_{\mathscr{M}}(s,T) = \{s' \in \mathscr{S} \mid \exists \ell \in \mathscr{A}.\ s \xrightarrow{\ell} s' \in T\}$. Therefore, $nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$ is the set of successor states of s. Whenever $en_{\mathscr{M}}(s) = \emptyset$, we call s a *deadlock* state. We refer to the set of deadlock states as $\mathscr{B} = \{s \mid en_{\mathscr{M}}(s) = \emptyset\}$. Finally, in the state space setting, parameters of an action are considered to be included in its action label ℓ. However, whenever we compare action labels, for instance $\ell = a$, we ignore the parameters. We are aware of this discrepancy, but trying to avoid it would lead to unnecessary complications.

A state space generation algorithm is normally provided with a given specification (e.g. $\mathcal{M}$ in LPE form) as input, and produces the state space that is described by that specification as output. For the sake of presentation, let us say that for all $a \in \mathcal{A}$, $a : \mathbb{D}_{\mathcal{A}}$ (such a data type can be created as a Cartesian product of all the data types of actions in $\mathcal{A}$). The relation between a specification $\mathcal{M}$ and a state space $\mathscr{M}$ can be expressed by means of the functions $mapp : \mathbb{P} \rightarrow \mathscr{S}$ and $mapa : \mathcal{A} \times \mathbb{D}_{\mathcal{A}} \rightarrow \mathscr{A}$, where *mapp* is a mapping of μCRL process terms to states, and *mapa* is a mapping of μCRL actions plus data elements of $\mathbb{D}_{\mathcal{A}}$ to action labels. The definitions of these mappings are established on-the-fly, while generating the state space. Initially, $mapp(\mathcal{I}) \in \mathscr{I}$. Then, for all process terms P such that $\mathcal{I} \xrightarrow{a(e)} P$, we define $mapp(P) \in \mathscr{S}$, $mapa(a,e) \in \mathscr{A}$, and $(mapp(\mathcal{I}), mapa(a,e), mapp(P)) \in \mathscr{T}$. In this manner, we continue to create $\mathscr{S}$, $\mathscr{A}$, and $\mathscr{T}$ by means of $\mathcal{M}$.

There are many different ways to generate a state space. In this section, we describe the two most basic ones, namely *breadth-first* and *depth-first* state space generation. The breadth-first state space generation (BFS) algorithm is described by Algorithm 1. There, each level i of the state space is represented by $\mathscr{L}_i \subseteq \mathscr{S}$. In Part II of this thesis, other state space generation algorithms are presented and proposed, by which it is demonstrated how this basic algorithm can be optimised or accelerated for specific applications.

Whenever, during generation, a state is encountered for the first time, we say that the state has been *visited*. The activity of visiting the successor states of a visited state s is usually referred to as *exploring* or *expanding* state s. Notice that after generation of the successor states, which are placed in the set $\mathscr{L}_{i+1}$, some states are removed again, namely those which have been encountered before, and therefore are also present in the union of the previous levels, i.e. $\bigcup_{j=0}^{i-1} \mathscr{L}_j$. This check is referred to as *duplicate detection*. It is an important step in state space search, since it guarantees termination of state space search when generating finite state spaces, also when they contain cycles. Furthermore, in Chapter 6, this particular step is extended at times to avoid unnecessary exploration in smarter ways.

A *round* i of the algorithm corresponds to a logical level in the state space, which is processed in the i^{th} iteration of a state space generation algorithm. It typically produces $\mathscr{L}_i$. In Algorithm 1, a round corresponds with one traversal through the **while**-loop.

Algorithm 1 Breadth-first state space generation

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{I})$
Ensure: $true$
  $i \leftarrow 0$
  $\mathscr{L}_i \leftarrow \mathscr{I}$
  while $\mathscr{L}_i \neq \emptyset$ do
    $\mathscr{L}_{i+1} \leftarrow \emptyset$
    for all $s \in \mathscr{L}_i$ do
      $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \cup nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathscr{L}_i \leftarrow \mathscr{L}_i \setminus \bigcup_{j=0}^{i-1} \mathscr{L}_j$
  end while
  return $true$
```

Contrary to breadth-first state space generation, depth-first state space generation is not supported by the μCRL toolset. However, it is a prominent algorithm present in other model checkers, such as SPIN and UPPAAL, and we return to it several times in this thesis. Therefore, we describe it in this chapter.

In depth-first state space generation, an individual trace is followed to the end, after which the algorithm backtracks, in order to generate another trace. This may continue until the complete state space is generated. Compared to breadth-first state space generation, it reaches deeper into the state space much faster. If a state space contains cycle-less traces of infinite length, the generation might get trapped into these traces, since the end can never be reached. However, such traces also cause best-first state space generation to go on forever. A common way to ensure that depth-first state space generation terminates is to use a depth upper-bound D; whenever the algorithm has reached this depth, it will backtrack, independent of whether the current trace continues beyond that depth, or not. Algorithms 2 and 3 describe depth-first state space generation recursively, in a set-based way. They do not include the upper-bound D, as this can already be seen as an extension of the most basic version of the algorithms. In Algorithm 2, the set $\hat{\mathscr{L}_i}$ is used to contain a subset of level $\mathscr{L}_i$, consisting of states selected for exploration. The exploration of $\hat{\mathscr{L}_i}$ is then performed by calling the function *dfs*, which is described in Algorithm 3. It explores all the states in the given set of states, places the successor states in $\mathscr{L}_{i+1}$, and repeats the procedure of Algorithm 2 on $\mathscr{L}_{i+1}$, namely it selects a subset $\hat{\mathscr{L}}_{i+1}$ for exploration, and calls the function *dfs*. In this manner, the generation continues until *dfs* is called with an empty set of states to explore, or it finds no successor states. Whenever this happens, the generation backtracks to a point where there are still visited states to explore, and it continues the generation. Note that unlike in best-first state space generation, the value of i can also decrease during a depth-first state space generation. This reflects the fact that a breadth-first state space generation goes deeper and deeper into a state space, without backtracking, but a depth-first state space generation backtracks every now and then. Finally, in case $\hat{\mathscr{L}}_i$ and $\hat{\mathscr{L}}_{i+1}$ are always singleton sets, Algorithms 2 and 3 describe so-called *explicit-state* depth-first state space generation. The set *Closed* is used for duplicate-detection.

Algorithm 2 Depth-first state space generation

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{I})$
Ensure: $true$
  $Closed \leftarrow \emptyset$
  $\mathscr{L}_0 \leftarrow \mathscr{I}$
  while $\mathscr{L}_0 \neq \emptyset$ do
    select $\hat{\mathscr{L}}_0 \subseteq \mathscr{L}_0$
    $\mathscr{L}_0 \leftarrow \mathscr{L}_0 \setminus \hat{\mathscr{L}}_0$
    $Closed \leftarrow Closed \cup \hat{\mathscr{L}}_0$
    $Closed \leftarrow dfs(\hat{\mathscr{L}}_0, Closed)$
  end while
  return $true$
```

Both breadth-first and depth-first state space generation are considered to be *blind* or *uninformed* searches, as there is no additional knowledge concerning the specified problem accessed in order to guide the search in a smart way. In Part II of this thesis, we focus on other searches, which are not blind. Even though the descriptions of the algorithms in Chapter 6 are presented based on Algorithm 1, the depth-first approach is implicitly covered. An explicit usage of the depth upper-bound D in depth-first state space generation can be found in Section 7.3.3.

Due to the state space explosion problem, state spaces may get too big for a single computer to handle. Because of this, *distributed* state space generation is being developed, where multiple computers together generate a state space, interchanging state information whenever needed. As mentioned in Chapter 1, with the μCRL toolset, distributed state space generation, minimisation, and analysis can be performed.

Moving to a distributed setting, we no longer deal with one machine, but one manager and n clients, where $n \in \mathbb{N}$. Contrary to depth-first state space generation, which generates a single trace at a time, breadth-first state space generation lends itself well for a distributed setting. In distributed breadth-first state space generation, ba-

Algorithm 3 Procedure dfs($\mathscr{L}_i$, *Closed*)

```
if $\mathscr{L}_i \neq \emptyset$ then
    $\mathscr{L}_{i+1} \leftarrow \emptyset$
    for all $s \in \mathscr{L}_i$ do
        $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \cup nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
    end for
    $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \setminus Closed$
    while $\mathscr{L}_{i+1} \neq \emptyset$ do
        select $\hat{\mathscr{L}}_{i+1} \subseteq \mathscr{L}_{i+1}$
        $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \setminus \hat{\mathscr{L}}_{i+1}$
        $Closed \leftarrow Closed \cup \hat{\mathscr{L}}_{i+1}$
        $Closed \leftarrow dfs(\hat{\mathscr{L}}_{i+1}, Closed)$
    end while
end if
return $Closed$
```

Algorithm 4 Distributed BFS state space generation - Client Instantiator

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{I})$, client number ID, set of client numbers CIDs, hash function $\# : \mathscr{S} \rightarrow$ CIDs
$i \leftarrow 0$
$\mathscr{L}_i^{\mathtt{ID}} \leftarrow \emptyset$
for all $s \in \mathscr{I}$ do
    if $\#(s) = \mathtt{ID}$ then
        $\mathscr{L}_i^{\mathtt{ID}} \leftarrow \mathscr{L}_i^{\mathtt{ID}} \cup \{s\}$
    end if
end for
repeat
    $\mathscr{L}_{i+1}^{\mathtt{ID}} \leftarrow \emptyset$
    for all $s \in \mathscr{L}_i^{\mathtt{ID}}$ do
        $\mathscr{L}_{i+1}^{\mathtt{ID}} \leftarrow \mathscr{L}_{i+1}^{\mathtt{ID}} \cup nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
    end for
    $SendToMgrNewStatesFound(|\mathscr{L}_{i+1}^{\mathtt{ID}}| > 0)$
    $i \leftarrow i + 1$
    $command \leftarrow RecvFromMgr()$
    if $command \neq \mathtt{finish}$ then
        $SendToClientsNextLevel(\mathscr{L}_i^{\mathtt{ID}})$
        $\mathscr{L}_i^{\mathtt{ID}} \leftarrow RecvFromClientsNextLevel()$
        $\mathscr{L}_i^{\mathtt{ID}} \leftarrow \mathscr{L}_i^{\mathtt{ID}} \setminus \bigcup_{j=0}^{i-1} \mathscr{L}_j^{\mathtt{ID}}$
    end if
until $command = \mathtt{finish}$
```

Algorithm 5 Distributed BFS state space generation - Manager Instantiator

```
Require: Set of client numbers CIDs
Ensure: true
  repeat
    nextlevel ← false
    for all ID ∈ CIDs do
      nextlevel ← nextlevel ∨ RecvFromClientNewStatesFound(ID)
    end for
    if ¬nextlevel then
      SendToClients(finish)
    else
      SendToClients(continue)
    end if
  until ¬nextlevel
  return true
```

sically every client generates part of the state space in a breadth-first manner. After generating state space level $\mathscr{L}_{i+1}$, given level $\mathscr{L}_i$, how the states in $\mathscr{L}_{i+1}$ should be distributed over the n clients is determined by a hash function $\# : \mathscr{S} \rightarrow \{1, \ldots, n\}$. In other words, # assigns to each state a unique owner. Algorithms 4 and 5 present distributed breadth-first state space generation for a client and a manager, respectively, where every client is initialised with a unique client number `ID`. By means of the function *RecvFromClientsNextLevel*(), each client `ID` receives at the end of a round of the generation a part of the current state space level $\mathscr{L}_i$, called $\mathscr{L}_i^{\mathtt{ID}}$, consisting of all the states in $\mathscr{L}_i$ owned by client `ID` according to the # function. With n clients, we have for each level i that $\mathscr{L}_i = \mathscr{L}_i^1 \cup \ldots \cup \mathscr{L}_i^n$. Duplicate detection is now performed by each client after having received the new set of states $\mathscr{L}_i^{\mathtt{ID}}$ to be expanded. This works thanks to the # function, which ensures that a state is always assigned to the same client. Expanding the new states in $\mathscr{L}_i^{\mathtt{ID}}$ results in a part of the next level of the state space $\mathscr{L}_{i+1}$, called $\mathscr{L}_{i+1}^{\mathtt{ID}}$. This part, however, does not constitute the part to be explored by client `ID`. The states in $\mathscr{L}_{i+1}^{\mathtt{ID}}$ need to be sent to their owners, according to #. This is done by calling *SendToClientsNextLevel*($\mathscr{L}_{i+1}^{\mathtt{ID}}$). Once each state in the system has been received by its owner, the generation can continue.

Matching send and receive functions can be identified by their names. During the generation, a client can receive the following commands from the manager, through the function *RecvFromMgr*():

- `continue`: In the next step, receive new states in $\mathscr{L}_i$ and expand them.
- `finish`: Stop the search algorithm. This message is sent by the manager when it observes that in the last round, no client has generated new states.

Via the function *SendToMgrNewStatesFound*(), all clients report whether they have visited new states or not. If at least one client reports that it did, the manager will decide to continue the generation.

For more information on distributed state space generation, the reader is referred to, for instance, Ciardo et al. (1998).

2.3 Verifying Properties With Temporal Logic

Using temporal logics, one can express properties about a state space. A model checker can then be used to check, whether an expressed property holds in that state space or not. Two temporal logics used in this thesis are *Linear Temporal Logic* (LTL) (Pnueli, 1981) and the *regular alternation-free μ-calculus* (Kozen, 1983; Mateescu and Sighireanu, 2003). Here, we give a brief introduction to the μ-calculus, relevant for understanding the formulas in this thesis. E.g. Mateescu and Sighireanu (2003) provide a more detailed description. A full explanation of the syntax of both LTL and μ-calculus is provided by, for instance, Clarke et al. (1999).

We consider here μ-calculus for expressing action-based properties of states. There exists a more elaborate version of this logic which can express temporal properties involving data values, but that one is not supported (yet) by CADP, and therefore not used for the current work.

The state formulas of μ-calculus are defined by the following BNF grammar:

$$\phi ::= \mathrm{F} \mid \mathrm{T} \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \langle\ell\rangle\phi \mid [\ell]\phi \mid Y \mid \mu Y.\phi \mid \nu Y.\phi$$

where ℓ ranges over $\mathscr{A}$, and Y ranges over some collection of propositional variables. A state formula can be built using both actions and variables, combining them with the usual boolean operators, the possibility and necessity modal operators $\langle\ell\rangle\phi$ and $[\ell]\phi$, and the minimal and maximal fixed point operators $\mu Y.\phi$ and $\nu Y.\phi$. The expression $\langle\ell\rangle\phi$ intuitively means "it is possible to fire an ℓ-transition, leading to a state where ϕ holds". Similarly, $[\ell]\phi$ can be interpreted as "ϕ holds in all states reachable by firing an ℓ-transition".

The μ and ν operators act as binders for Y variables, similar to quantifiers in first-order logic. Here, we restrict to *closed* μ-calculus formulas, meaning that all variables occurring in ϕ are bound. A state satisfies $\mu Y.\phi$ iff it belongs to the minimal solution of the fixed point equation $Y = \phi(Y)$. Similarly, a state satisfies $\nu Y.\phi$ iff it belongs to the maximal solution of the fixed point equation $Y = \phi(Y)$.

The *alternation-free* μ-calculus (Emerson and Lei, 1987) consists of μ-calculus formulas with no alternation between minimal and maximal fixed point operators. In practice, this makes a good compromise between expressiveness and efficiency of model checking.

In the *regular* μ-calculus (Mateescu and Sighireanu, 2003), action predicates and regular expressions over action sequences can be used in the state formulas. Here, $\langle\beta\rangle\phi$ and $[\beta]\phi$ can be used, where β is a regular formula. The following BNF grammar defines regular formulas:

$$\beta ::= \alpha \mid \beta_1.\beta_2 \mid \beta_1 \mid \beta_2 \mid \beta^*$$

A regular formula is built from action formulas α (to be defined next), using the concatenation ('.'), choice ('|'), and the reflexive transitive closure ('*') operators. The empty sequence operator ϵ and the transitive closure operator $^+$ are defined as $\epsilon = \mathrm{F}^*$ and $\beta^+ = \beta.\beta^*$.

Finally, action formulas are defined as follows:

$$\alpha ::= \ell \mid \neg\ell \mid \ell_1 \wedge \ell_2$$

Action formulas α can be constructed from action names $\ell \in \mathscr{A}$ using the standard boolean operators. Derivable boolean connectives are e.g. $\mathrm{F} = \ell \wedge \neg\ell$, $\mathrm{T} = \neg\mathrm{F}$, and $\ell_1 \vee \ell_2 = \neg(\neg\ell_1 \wedge \neg\ell_2)$.

Next, we provide some examples. Note that T holds in all states. Therefore, at places in a regular formula, where the name of an action is not important we can write T. Hence, a statement like a.T.b means "first we encounter an action a, then some other action, followed by an action b".

If we write '$\langle R\rangle$ T', then we express that there exists a trace in the state space for which the regular formula R holds. Such a property is often referred to as a *liveness* property. The expression '$[R]$ F' expresses that for all traces in the state space the regular formula R does not hold. This kind of property is called a *safety* property, where R is some undesirable behaviour.

As a final example, the formula '[a.T*.b] F' expresses that for all traces in the state space, we do not find one action a, followed by zero or more other actions, followed again by one action b.

Temporal logic formulas can be used to express a reachability property. As mentioned in Section 2.2, when checking a reachability property, the goal is to find a state $s \in \mathscr{G}$, with $\mathscr{G}$ a given set of goal states. Of course, when checking such a property on-the-fly and $\mathscr{S}$ is not known a priori, $\mathscr{G}$ cannot be given explicitly; however, a temporal logic formula can be provided, by which it is decidable whether a state is in $\mathscr{G}$ or not. This is because a state formula ϕ can be interpreted as a set of states in which ϕ holds, thereby ϕ can be mapped to $\mathscr{G}$.[3] Mateescu and Sighireanu (2003) explain how state formulas relate to sets of states. First of all, action formulas relate to sets of actions as

[3] In practice, a state formula ϕ is often mapped to $\mathscr{S} \setminus \mathscr{G}$, where ϕ actually expresses the negation of the desired property. Then, whenever a state is found which violates ϕ,i.e. it is a goal state, a trace to this state is reported by the model checker.

follows, where the interpretation $[\![\alpha]\!] \subseteq \mathscr{A}$ of action formulas gives the set of actions satisfying α:

$$\begin{array}{rcl} [\![\ell]\!] & = & \{\ell\} \\ [\![\neg\ell]\!] & = & \mathscr{A} \setminus [\![\ell]\!] \\ [\![\ell_1 \wedge \ell_2]\!] & = & [\![\ell_1]\!] \cap [\![\ell_2]\!] \end{array}$$

Next, regular formulas relate to pairs of source and target states. The interpretation $(\!(\beta)\!) \subseteq \mathscr{S} \times \mathscr{S}$ of regular formulas gives a binary relation between source and target states of transition sequences satisfying β. The relation between regular formulas and pairs of states is as follows, where $\circ$, $\cup$, and * are the composition, union, and reflexive transitive closure operators of binary relations, respectively:

$$\begin{array}{rcl} (\!(\alpha)\!) & = & \{(s,s') \in \mathscr{S} \times \mathscr{S} \mid \exists \ell \in \mathscr{A}.s \xrightarrow{\ell} s' \wedge \ell \in [\![\alpha]\!]\} \\ (\!(\beta_1.\beta_2)\!) & = & (\!(\beta_1)\!) \circ (\!(\beta_2)\!) \\ (\!(\beta_1 \mid \beta_2)\!) & = & (\!(\beta_1)\!) \cup (\!(\beta_2)\!) \\ (\!(\beta^*)\!) & = & (\!(\beta)\!)^* \end{array}$$

An action formula relates to one-step sequences $s \xrightarrow{\ell} s'$ such that ℓ satisfies α. Concerning the formula $\beta_1.\beta_2$, a sequence is the concatenation of two sequences satisfying β_1 and β_2, respectively. Regarding the formula $\beta_1 \mid \beta_2$, a sequence can satisfy β_1 or β_2. A sequence satisfying β^* is a concatenation of (zero or more) sequences satisfying β.

Finally, as previously mentioned, state formulas relate to sets of states. The interpretation $[\![\phi]\!]\varrho \subseteq \mathscr{S}$ of state formulas, where ϱ is a so-called *environment* or *propositional context* assigning state sets to propositional variables, gives the set of states satisfying ϕ in the context of ϱ. The relation between state formulas and sets of states is as follows, where we denote by $\varrho[Y \leftarrow S]$ a new environment, identical to ϱ, except that $\varrho[Y \leftarrow S](Y) = S$:

$$\begin{array}{rcl} [\![\mathrm{F}]\!]\varrho & = & \emptyset \\ [\![\mathrm{T}]\!]\varrho & = & \mathscr{S} \\ [\![\phi_1 \vee \phi_2]\!]\varrho & = & [\![\phi_1]\!]\varrho \cup [\![\phi_2]\!]\varrho \\ [\![\phi_1 \wedge \phi_2]\!]\varrho & = & [\![\phi_1]\!]\varrho \cap [\![\phi_2]\!]\varrho \\ [\![\langle\beta\rangle\phi]\!]\varrho & = & \{s \in \mathscr{S} \mid \exists s' \in \mathscr{S}.(s,s') \in (\!(\beta)\!) \wedge s' \in [\![\phi]\!]\varrho\} \\ [\![[\beta]\phi]\!]\varrho & = & \{s \in \mathscr{S} \mid \forall s' \in \mathscr{S}.(s,s') \in (\!(\beta)\!) \implies s' \in [\![\phi]\!]\varrho\} \\ [\![Y]\!]\varrho & = & \varrho(Y) \\ [\![\mu Y.\phi]\!]\varrho & = & \bigcap\{S \subseteq \mathscr{S} \mid [\![\phi]\!]\varrho[Y \leftarrow S] \subseteq S\} \\ [\![\nu Y.\phi]\!]\varrho & = & \bigcup\{S \subseteq \mathscr{S} \mid S \subseteq [\![\phi]\!]\varrho[Y \leftarrow S]\} \end{array}$$

Here, the formulas $\langle\beta\rangle\phi$ and $[\beta]\phi$ relate to the states for which some (all) of the outgoing transition sequences satisfying β lead to states satisfying ϕ. The formulas $\mu Y.\phi$ and $\nu Y.\phi$ correspond with the minimal and maximal solutions (over sets of states) of the fixed point equation $Y = \phi$.

We can now state the following for reachability problems regarding a set of goal states $\mathscr{G}$: Given a state formula ϕ such that $[\![\phi]\!]\varrho = \mathscr{G}$, we can check whether a state s is a goal state, by checking if $s \in [\![\phi]\!]\varrho$. As Part II of this thesis presents a range of state space search algorithms which accept a set of goal states $\mathscr{G}$, the interpretation $[\![\phi]\!]\varrho \subseteq \mathscr{S}$ forms a bridge, allowing us to express a reachability property using the regular alternation-free μ-calculus, and subsequently search for states where the property holds (or not), using these search algorithms.

Part I

Modelling Time

IN WHICH ONE MODELLING LANGUAGE IS TRANSLATED TO ANOTHER, THE VERIFICATION OF A TIMED SYSTEM IS DEMONSTRATED, AND A TIMING MECHANISM IS MODELLED IN AN UNTIMED SETTING

This part is based on the work of Bortnik et al. (2005a), Wijs and Fokkink (2005), and Wijs (2007)

Chapter 3

From χ_t to μCRL: A Translation

> *Je n'ai fait celle-ci plus longue que d'habitude parce que je n'ai pas eu le loisir de la faire plus courte.*
>
> (*Blaise Pascal*)

3.1 Introduction

PERFORMANCE ANALYSIS IS traditionally based on techniques such as simulation, Markov chains and queueing networks. By contrast, important approaches for verifying functional properties are model checking, where temporal formulas are validated by means of an explicit state space search, and theorem proving, which is largely based on axiomatic reasoning at the symbolic level.

In the last few years, these approaches for verifying functional properties have been extended in order to verify performance aspects of systems. Hermanns and Katoen (2001) verified performance properties of a LOTOS specification of a telephone system; LOTOS (Bolognesi and Brinksma, 1987) is a process algebraic language with abstract data types, which is originally meant for functional analysis. Garavel and Hermanns (2002) introduced a general approach to carry out some performance analysis within the framework of LOTOS. They introduce timing information into a LOTOS specification, expressing that certain events are delayable by some random delay, captured by an exponential distribution. From this extended LOTOS specification they generate an interactive Markov chain, which is basically a labelled transition system containing both actions and positive reals as labels, where the positive reals denote delays. They explain how the CADP toolset, which is actually meant for functional verification of LOTOS specifications, can be used to also carry out performance analysis with respect to interactive Markov chains. Although the approach of Garavel and Hermanns is promising, it is difficult if not impossible to apply full-blown performance analysis techniques in a functional verification formalism like LOTOS.

In this chapter we present another approach to bridge the gap between performance and functional analysis. Similar to Garavel and Hermanns, we exploit the fact that specification languages for performance and functional analysis tend to have a lot in common, so that a translation from one specification language to the other is quite feasible. However, we propose to keep the performance and the functional analysis separate, in environments targeted to these analyses. Thus we are in principle able to carry out full-blown performance as well as functional analysis.

χ (Van Beek et al., 2006) is a modelling language for the specification of discrete-event, continuous or combined, so-called *hybrid*, systems. It is based on the process algebra CSP (Hoare, 1985), and contains some predefined data types. It targets performance analysis of timed systems by means of simulation techniques to estimate throughput and cycle time. A subset of the language χ, restricted to specify only discrete-event systems, is called timed χ, or χ_t (Van Beek et al., 2005). Currently, there is only a simulator available for using the current version of the language χ (the toolset is being developed (Toolset of χ, see bibliography)), but predecessors of the language and their toolsets have been successfully applied to a large number of industrial cases, such as an integrated circuit manufacturing plant, a brewery and process industry plants (Van Beek et al., 2002).

In this chapter we present a general translation from χ_t specifications to LPEs. Note that we have to limit ourselves to translating χ_t instead of the complete hybrid χ, because μCRL cannot cope with continuous events. A translation from χ_t to μCRL is feasible, because, although the modelling languages χ_t and μCRL have different aims, there are a number of similarities. Most importantly, their input languages are both based on process algebra, and they are both action-based. The verification of a turntable system in Bortnik et al. (2005a) illustrates how our translation scheme can be used to combine performance and functional analysis. In Bortnik et al. (2005a) the χ_t specification of a turntable (Bos and Kleijn, 2001) was translated to three different specification formalisms: UPPAAL, SPIN and μCRL. The latter translation is also presented here in Section 4.2.

Our work is closest in spirit to TWOTOWERS (Bernardo et al., 1998), which is a tool that combines performance and functional analysis. It has a single input language, based on the stochastic process algebra EMPA (Bernardo and Gorrieri, 1998). Performance analysis is based on simulation and reward Markov chains, while functional analysis is performed by the symbolic model checker NUSMV (Cimatti et al., 2002).

This chapter is set up as follows. The next section provides a short introduction to χ_t; the basics of the language are listed and a brief explanation is given. The approach to model time in μCRL, which is used in the translation scheme, is explained in Section 3.3. Section 3.4 provides a way to translate χ_t specifications to linear process equations. Later, Chapter 4 provides examples of translating a χ_t specification to a μCRL specification, and also shows the verification of an industrial case study.

3.2 The Language χ_{t}

The χ language was designed as a hybrid modelling and simulation language. Since we are interested only in discrete-event specifications and verification, we present here just a part of the language, disregarding features that are used for simulation and to model hybrid behaviour. This (discrete-event) subset of the language is known as timed χ or χ_{t} (Van Beek et al., 2005). For a description of the complete χ, see Van Beek et al. (2006).

Data types Like μCRL, the χ_{t} language is statically strongly typed. Every variable has a type which defines the allowed operations on that variable. The basic data types are boolean, natural, integer, real number, string and enumeration. The language provides mechanisms to build sets, lists, array tuples, record tuples, dictionaries, functions, and distributions (for stochastic specifications). Channels also have a type that indicates the type of data that is communicated via the channel.

Time model Time in χ_{t} is dense, i.e. timing is measured on a continuous time scale. The strong time determinism principle, or sometimes called the time factorisation property (time does not make a choice), and urgent communication (a process can delay only if it cannot do anything else) are implicit. Time additivity (if a process can delay first t_1 and then immediately following t_2 time units, then it can delay $t_1 + t_2$ time units from the start) is not present. Delaying is enforced by the delay operator, but some atomic processes can also delay implicitly.

Communication model Communication in χ_{t} is synchronous, meaning that a *send* and a *receive* action on the same channel cannot happen individually but only together, as one communication action.

Atomic processes The atomic processes of χ_{t} are process constructors which cannot be split into smaller processes. They are:

1. The multi-assignment process ($x_n := e_n$). It assigns the values (which must be defined) of expressions $e_1, ..., e_n$ to the variables $x_1, ..., x_n$, respectively. It does not have the possibility to delay.

2. The skip process. It performs the internal action τ and cannot delay.

3. The send process ($h\,!!\,e_n$). It sends the values of the expressions $e_1, ..., e_n$, for $n \geq 1$, via channel h. For $n = 0$, $h\,!!\,e_n$ becomes $h\,!!$ and nothing is sent via the channel.

4. The send process ($h\,!\,e_n$) is the delayable equivalent of $h\,!!\,e_n$. It is able to delay arbitrarily long before sending.

5. The receive process ($h\,??\,x_n$). It receives values via the channel h and assigns them to the variables $x_1, ..., x_n$. For $n = 0$, $h\,??\,x_n$ becomes $h\,??$ and nothing is received via the channel.

6. The receive process ($h\,?\,x_n$) is the delayable equivalent of $h\,??\,x_n$. It is able to delay arbitrarily long before receiving.

7. The delay process (Δt). It delays a number of time units equal to the value of the expression t. The value of t must be a positive real number.

Operators Atomic processes can be combined by means of the following operators. We present each one of them together with their (informal) semantics. We do not consider operators that are only used for the definition of the semantics of χ_{t}, since those never appear in specifications. Two exceptions to this are the encapsulation operator and the urgent communication operator. These operators are implicitly used in χ_{t}, but should be considered explicitly when translating a specification to μCRL.

1. The delay operator (Δ_t). The process $\Delta_t(p)$ is forced to delay for the amount of time units specified by the value of numerical expression t, after which it can proceed as p.

2. The delay enabling operator ([]). For a process $[p]$, time transitions of arbitrary duration are allowed before the execution of p is initiated.

3. The guard operator ($\rightarrow$). For action behaviour, a process $b \rightarrow p$ behaves as p if the value of the boolean expression (guard) b is *true*. For delay behaviour, $b \rightarrow p$ can delay according to p as long as the boolean expression b evaluates to *true*. While b evaluates to *false*, $b \rightarrow p$ can perform any delay, but p cannot execute.

4. The sequential composition operator (;). A process $p;\ q$ behaves as p followed by q.

5. The alternative composition operator ($[\!]$). A process $p \mathbin{[\!]} q$ represents a (non-deterministic) choice between p and q (if they can proceed).

6. The repetition operator ($*$). A process $*p$ will keep on executing p.

7. The guarded repetition operator ($*$:). A process $*b : p$ can be interpreted as “while b do p”.

8. The parallel composition operator ($\parallel$). A process $p \parallel q$ executes the actions of p and q concurrently in an interleaved fashion. If one of the processes can execute a *send* action and the other one can execute a *receive* action on the same channel, then $p \parallel q$ executes the communication action on this channel.[1]

9. The scope operator ($[\![\mid]\!]$). A process $[\![s \mid p]\!]$ behaves as p in a local state s. The state s is used to define local variables and channels visible only to the process p. It is defined as "$\mathsf{disc}\ \mathbf{s}, \mathsf{chan}\ \mathbf{h}, i, L_R$" with $\mathbf{s}$ a list of local variables, $\mathbf{h}$ a list of local channels, i an initialisation predicate, restricting the allowed values of the variables initially, and L_R a list of recursion definitions. A variable s_i is defined as $s_i : \mathit{type}$. A channel h_i can either be a channel for receiving, defined as $h_i\,? : [\mathit{type}]$ (optionally it can receive data of a given *type*), a channel for sending, defined as $h_i\,! : [\mathit{type}]$, or a channel for both receiving and sending, defined as $h_i\,!? : [\mathit{type}]$. In the initialisation predicate, terms may appear of the form $s_i = e_i$, with s_i a variable defined in $\mathbf{s}$ and e_i a value.[2] Finally, a recursion definition in L_R is of the form $X_i \mapsto p_i$, with X_i a recursion variable and p_i a process.

10. The encapsulation operator (∂_A). A process $\partial_A(p)$ disables all actions of p that occur in the set A. Typically this operator is used to enforce that send and receive actions synchronise.

11. The urgent communication operator (υ_H). Send and receive actions in a process $\upsilon_H(p)$ via channels from set H can only delay when no communication with a corresponding receive or send action on the same channel is possible.

Specification definitions The language χ_{t} provides the possibility to define specifications. A χ_{t} specification is of the following form:

$$\langle\ \mathsf{disc}\ s_1, \ldots s_k, \mathsf{chan}\ h_1, \ldots, h_m, i, X_1 \mapsto p_1, \ldots, X_r \mapsto p_r \mid p \rangle$$

Here, $\mathsf{disc}\ s_1, \ldots s_k, \mathsf{chan}\ h_1, \ldots, h_m, i$ and $X_1 \mapsto p_1, \ldots, X_r \mapsto p_r$ are as described earlier for the scope operator and p is a process.

[1] If $p \parallel q$ can execute a communication action, then its separate component actions cannot be executed, due to implicit encapsulation of all the send and receive actions in χ_{t}.

[2] In χ_{t} it is also possible to provide a range of possible valuations for a variable. We do not consider this here, since it cannot be modelled easily in μCRL.

Finally, an example of a χ_t specification:

$$
\begin{array}{l}
\langle\ \mathsf{chan}\ c\,!?:\mathsf{nat} \\
\quad ,P \mapsto [\![\ \mathsf{disc}\ x:\mathsf{nat}\,, b:\mathsf{bool}\,, b=\mathsf{true} \mid b \to c\,?\,x\ ]\!] \\
\quad ,Q \mapsto [\![\ \mathsf{disc}\ y:\mathsf{nat}\,, y=3 \mid c\,!\,y\ ]\!] \\
\mid P \parallel Q \\
\rangle
\end{array}
$$

Since initially b = true, process P can receive a natural number via channel c. Process Q sends the value 3 over this channel.

3.3 Using Time in μCRL

Delaying for a certain amount of time is impossible in μCRL at first glance. This is because μCRL does not work with time. A later extension of μCRL to *timed* μCRL (Groote, 1997) introduced the notion of time. However, at present, creating a timed μCRL specification is not very practical since the μCRL toolset can only parse timed μCRL code and cannot generate a state space from it.

There is another way, however, to simulate some notion of discrete time. In this part of the thesis, we build on a method introduced by Blom et al. (2003) and Ioustinova (2004). Since it forms an important basis for this and the following two chapters, we describe the method here, adapted to our own notation.

In general, timing can be either *absolute* or *relative* and the time scale can be either *continuous*, also known as *dense*, or *discrete* (Baeten and Middelburg, 2002). In absolute timing, the execution time of an action is expressed in an absolute fashion with respect to the running time of the modelled system, whereas in relative timing, the time of an action is expressed relative to the execution of a previous action. Besides that, one can adopt a setting in which atomic actions take zero time to execute, or some non-zero time to execute. The latter case is often achieved by equipping all actions with a time stamp, indicating when the action has to be fired (e.g. 12:05 PM). When atomic actions may take zero time to execute, time stamps are not needed; instead, we can choose for the so-called *two-phase* model (Nicollin and Sifakis, 1991), where a system alternatingly fires action and time transitions. Practically all combinations of the settings described here exist in the literature; for instance, Baeten and Middelburg (2002) use, among other settings, absolute timing with the two-phase model, whereas in Part III of this thesis, absolute timing is used with time stamps. In the approach for μCRL to be described next, relative timing is modelled in a two-phase manner, since the time unit, in which an action can be fired, is expressed relative to the time unit of actions fired earlier, using a special *tick* action for time transitions. On a discrete time scale, time is divided in finite time units. In a specification using such a scale, a jump in time cannot be smaller than a single time unit. On a continuous or dense time scale, there is no smallest time unit. Using the technique from Blom et al. (2003) and Ioustinova (2004), which we describe next, we get discrete time in μCRL.

For a specification $\mathcal{M}$, an action $tick \in \mathcal{A}$ is used to represent the end of a time slice and the beginning of a new one, i.e. to model a time transition. In order to share this notion of time, all running processes need to synchronise their *tick* actions. If at least one of these processes is busy and therefore unable to perform a *tick*, the *tick* action will not take place. In our notation, $(P_0 \parallel \ \ldots \parallel P_m \xrightarrow{tick} \Leftrightarrow \forall 0 \leq i \leq m.tick \in en_{\mathcal{A}}(P_i))$. This synchronisation aspect is essential if one wants to use global timing. On the top level of a specification, the synchronisation of all processes on *tick* is achieved by using an adapted parallel composition operator, called $|\{tick\}|$. It is defined as follows, given that $tick, tick' \in \mathcal{A}$ and $(tick, tick, tick') \in \mathcal{C}$:

$$P \ |\{tick\}| \ Q \triangleq \rho_{\{tick' \to tick\}}(\partial_{\{tick\}}(P \parallel Q))$$

In $\partial_{\{tick\}}(P \parallel Q)$, *tick* actions in P and Q are forced to communicate with each other. In $P \ |\{tick\}| \ Q$, the resulting $tick'$ transitions are renamed back to *tick*. The action $tick'$ is used for intermediate synchronisation results.

The $|\{tick\}|$ operator is commutative; this follows immediately from the commutativity of $\parallel$. Moreover, one can prove that $|\{tick\}|$ is associative for $tick'$-free processes; that is, if P, Q and R cannot perform any $tick'$-transitions, then

$$(P \ |\{tick\}| \ Q) \ |\{tick\}| \ R \leftrightarrows P \ |\{tick\}| \ (Q \ |\{tick\}| \ R)$$

We only informally argue why this is the case (for the sake of simplicity, we disregard successful termination).

The transitions of $(P \ |\{tick\}| \ Q) \ |\{tick\}| \ R$ can be classified as follows:

1. For actions $a \neq tick, tick'$, $(P \ |\{tick\}| \ Q) \ |\{tick\}| \ R \xrightarrow{a(e)} (P' \ |\{tick\}| \ Q') \ |\{tick\}| \ R'$ iff $(P \parallel Q) \parallel R \xrightarrow{a(e)} (P' \parallel Q') \parallel R'$;

2. Since P, Q, R are $tick'$-free, $(P \ |\{tick\}| \ Q) \ |\{tick\}| \ R \xrightarrow{tick} (P' \ |\{tick\}| \ Q') \ |\{tick\}| \ R'$ iff $P \xrightarrow{tick} P'$, $Q \xrightarrow{tick} Q'$, and $R \xrightarrow{tick} R'$;

3. Clearly, $(P \ |\{tick\}| \ Q) \ |\{tick\}| \ R$ cannot perform any $tick'$-transitions.

Likewise for the transitions of $P \ |\{tick\}| \ (Q \ |\{tick\}| \ R)$. Now, associativity of $|\{tick\}|$ follows in a straightforward fashion from associativity of $\parallel$.

Besides the introduction of *tick* and $|\{tick\}|$, there are specification disciplines given for the construction of a process. This is done to distinguish on the one hand *receive* and on the other hand *send* and internal actions, which differ in their ability to delay.

Therefore, two patterns are provided; one for a receive state and one for a send state. The receive state (or process) is, slightly adapted to fit our notation, as presented in Figure 3.1.

$$
\begin{aligned}
&A(d:\mathbb{D}, t_1:\mathit{Timer}, \ldots, t_m:\mathit{Timer}) = \\
&\quad a_1 \cdot X_1(d_{a_1}, t_1, \ldots, t_m) \lhd \mathit{expired}(t_1) \rhd \delta + \\
&\quad \vdots \\
&\quad a_m \cdot X_m(d_{a_m}, t_1, \ldots, t_m) \lhd \mathit{expired}(t_m) \rhd \delta + \\
&\quad \mathit{tick} \cdot A(d, t_1 - 1, \ldots, t_m - 1) \lhd \neg \bigvee_{j=1}^{m} \mathit{expired}(t_j) \rhd \delta + \\
&\quad \sum_{e_1:\mathbb{D}_1} \mathit{in}_1(f_1(d,e_1)) \cdot Y_1(g_1(d,e_1), t_1, \ldots, t_m) \lhd h_1(d,e_1) \rhd \delta + \\
&\quad \vdots \\
&\quad \sum_{e_k:\mathbb{D}_k} \mathit{in}_k(f_k(d,e_k)) \cdot Y_k(g_k(d,e_k), t_1, \ldots, t_m) \lhd h_k(d,e_k) \rhd \delta
\end{aligned}
$$

Figure 3.1: Receive process pattern

Timers $t_1, \ldots, t_m$ are here of the special type *Timer*. The data domain of time has the same structure as $\mathbb{N}$. A number of functions are applicable on a timer; its value can be increased, set and reset, and there is a check *expired*, which tells whether the value of the timer has already reached 0. In A, there are actions $a_1, \ldots, a_m$, which can be fired whenever one of the m timers has expired. The *tick* action can be fired when no timer has expired. Finally, there are receive actions $\mathit{in}_1, \ldots, \mathit{in}_k$.

A send (or, more precisely, non-receive) process is of the form presented in Figure 3.2. Note that there is no delay alternative here, meaning that the actions $b_1, \ldots, b_l$ are not delayable. The reason for this will become clear in the upcoming text.

When achieving time through modelling, one of the important issues is how communication should be dealt with. When two processes can communicate with each other, the communication should have priority over the passage of time. However, when only one process can communicate, it should be able to postpone this activity until the other process can do so. Blom et al. (2003) and Ioustinova (2004) resolve this issue by introducing asymmetry in communication, namely by allowing only receive actions to delay. Their argumentation for this is that otherwise communication cannot be seen as synchronous. Looking at χ_t, we see that communication there is symmetrical, so if we want a translation scheme from χ_t to μCRL, we would like to achieve symmetrical communication in μCRL. Considering the *tick*-mechanism described here, this seems

$$\begin{aligned}
&B(d : \mathbb{D}, t_1 : \mathit{Timer}, \ldots, t_m : \mathit{Timer}) = \\
&\quad \sum_{e_1 : \mathbb{D}_1} b_1(f_1(d, e_1)) \cdot Z_1(g_1(d, e_1), t_1, \ldots, t_m) \lhd\ h_1(d, e_1) \rhd \delta + \\
&\quad \vdots \\
&\quad \sum_{e_l : \mathbb{D}_l} b_l(f_l(d, e_l)) \cdot Z_l(g_l(d, e_l), t_1, \ldots, t_m) \lhd\ h_l(d, e_l) \rhd \delta
\end{aligned}$$

Figure 3.2: Non-receive process pattern

to be not possible.

However, the synchronisity of communication is not at issue in the *tick*-mechanism; in fact, by the very nature of communication in process algebras such as μCRL, communication cannot be anything but synchronous. Communication only occurs, when a send and a corresponding receive action can both be fired at the same time. Process P sending a message and, at a later time, process Q receiving it can actually not be modelled (unless we would, for example, additionally model some message buffer). What is at issue here is the so-called *maximal progress* of communication actions, i.e. communications as a whole need to have priority over the passage of time. In χ_{t}, this is achieved by the implicit use of the urgent communication operator. If we simply provide *tick* alternatives to both send and receive actions we will not achieve this. But this problem can be dealt with in other ways than making communication asymmetrical. Next, we present a way to avoid many situations which violate maximal progress; to be more exact, we avoid unforced delays in the system as a whole, i.e. the system only delays if one of its components has to delay. Full maximal progress can be achieved in a number of ways, though, which will be explained in the upcoming section.

Avoiding unforced delays can be achieved like this: We introduce a second time action, called *tock*. The differences between *tock* and *tick* are the following:

- The action *tick* is used for translating delays, while *tock* is used to make an action delayable (which means adding a *tock* self-loop as an alternative to this action);

- A *tick* action can synchronise with any number of *tick* or *tock* actions, but a *tock* action cannot synchronise with only *tock* actions (at least one *tick* action is needed for going from one time unit to the next).

Now, several delayable processes can delay together if there is a *tick* action enabled in at least one process.

In order to achieve this timing mechanism in μCRL, first of all we define *tick*, *tick*′, *tock*, *tock*′ ∈ $\mathcal{A}$ and (*tick*, *tock*, *tick*′), (*tock*, *tick*, *tick*′), (*tick*, *tick*, *tick*′), (*tock*, *tock*, *tock*′)∈

$\mathcal{C}$. Here, the primed actions are used for intermediate communication results. Note at this point that if at least one *tick* action is involved in communication, the result is *tick'*.

Second, we define a parallel composition operator $\overset{\top}{\|}$ in the following way:

$$P \overset{\top}{\|} Q \triangleq \rho_{\{tick' \to tick, tock' \to tock\}}(\partial_{\{tick,tock\}}(P \parallel Q))$$

In $\partial_{\{tick,tock\}}(P \parallel Q)$, *tick* and *tock* actions in P and Q are forced to communicate with each other. In $P \overset{\top}{\|} Q$, the resulting *tick'* and *tock'* transitions are renamed back to *tick* and *tock*. Commutativity and associativity of $\overset{\top}{\|}$ can be argued in a similar fashion as commutativity and associativity of $|\{tick\}|$.

By putting *tock* in the set H of the encapsulation operator used in the initialisation $\mathcal{I} = \partial_H(P_1 \overset{\top}{\|} \ldots \overset{\top}{\|} P_n)$, we remove all *tock* actions remaining in the final system.

Using this method we can avoid unforced delays in a system. Say we have a system consisting of two processes P and Q, and both send and receive actions are made delayable by means of *tock* self-loops. If process P is waiting for process Q to send a message, process Q may not be able to send it yet (in other words, cannot send it within the current time unit). Process Q may have to delay for a number of time units, by means of a (sequence of) *tick* action(s). This is possible using the new method, because process P can delay (can perform a *tock* self-loop). Now, the moment process Q is able to send the message, communication will take place immediately, even though both the send and the receive action are delayable, because two *tock* actions are not allowed to synchronise.

This, however, does not always coincide with maximal progress. Let us introduce a third process R, which wants to delay at the moment P and Q can communicate; then the latter two processes again have the possibility to perform either the communication or the synchronised *tick*, not preferring one above the other. Maximal progress can be fully enforced however by either post-processing the linearised system, or by dealing with it at the state space level, either on-the-fly or afterwards. These different approaches are described in the upcoming section. They may at the moment give the impression that the usage of *tock* is unnecessary; however, in Chapter 5, we further extend the functionality of *tock* such that it becomes an essential part of our mechanism.

In our approach we will use a special data type $\mathbb{T}$, which, as *Timer* in the approach of Blom et al. (2003) and Ioustinova (2004), functions as the type for timers. The data type $\mathbb{T}$, however, has the same structure as $\mathbb{Z}$, and the same typical functions, such as equality, apply on it. The inclusion of negative time values is not needed in this chapter, but will be essential in Chapter 5.

3.3.1 Maximal Progress

Timed systems often use a concept called *maximal progress*. It allows actions to have priority over the passage of time, as, for instance, explained by Baeten and Middelburg (2002), Nicollin and Sifakis (1991), and Ulidowski and Yuen (1997). The application of this technique differs between languages. Baeten and Middelburg (2002), for instance, define it as an operator, applicable on processes, giving actions from a given set H priority over time. In χ_t, on the other hand, a similar concept called *urgent communication* is *always* applied globally on a system, giving all actions priority (Van Beek et al., 2005). While maximal progress for a single process can certainly be imagined for μCRL with *tick* actions, here we focus on the possibility to achieve maximal progress on a μCRL specification as a whole. Therefore, in our approach, we choose a form between the ones used by Baeten and Middelburg (2002) and Van Beek et al. (2005); maximal progress can be applied once, globally on a specification $\mathcal{M}$. It is, however, not mandatory, and it applies for a given subset of actions $H \subseteq \mathcal{A} \setminus \{tick, tock, tock', tick'\}$ (clock actions can, of course, not be elements of H). The challenge here is that a global view of the system is needed; if, for instance, communication between two system components can occur, it should have priority over the passage of time. However, if this communication cannot yet take place, a corresponding send or receive action which is enabled within either of these system components should be delayable. This cannot be decided at the level of a system component, since it involves the conditions of other system components, i.e. it involves the interaction between system components. Therefore, we need to apply a global maximal progress once the whole system has already been defined.

There are a number of ways to achieve global maximal progress. We present three approaches here.

Transformation of an LPE First of all, we can enforce a prioritisation of actions within a μCRL specification after the linearisation of that specification. Once a specification is linearised, we have a single process X making up the specification $\mathcal{M}$, in which all possible communications appear as single actions. At that stage, it becomes possible to transform X such that maximal progress is enforced. Say that the finite index set $I = I_N \cup I_C$ with $I_N \cap I_C = \emptyset$, I_N being the set of indices of all actions in X which are neither *tick* nor *tock* actions (i.e. they are 'normal' actions) and I_C being the set of indices of all actions in X which are either *tick* or *tock* actions (i.e. they are 'clock' actions). Furthermore, given the set of actions to prioritise H, we say that $I_H = \{i \in I_N \mid a_i \in H\}$. Figure 3.3 shows how to achieve a transformation of an LPE X (in the form given in Definition 4), such that maximal progress holds in it. In X, there are no *tock* actions, since these are encapsulated.

This approach achieves maximal progress for a system by transforming the LPE. As the LPE resulting from linearising a μCRL specification is usually quite large and complex, this transformation can be rather difficult in practice. Next, we describe how

$$\begin{aligned}
&X(d:\mathbb{D}) = \\
&\quad \sum_{i\in I_N}\sum_{e_i:\mathbb{D}_i} a_i(f_i(d,e_i))\cdot X(g_i(d,e_i)) \lhd h_i(d,e_i) \rhd \delta \; + \\
&\quad \sum_{i\in I_C}\sum_{e_i:\mathbb{D}_i} \mathit{tick}\cdot X(g_i(d,e_i)) \lhd h_i(d,e_i) \wedge \neg \bigvee_{j\in I_H} h_j(d,e_i) \rhd \delta
\end{aligned}$$

Figure 3.3: An LPE X with maximal progress

we can achieve maximal progress by transforming a state space.

A maximal progress state space reduction tool We observe that globally applied maximal progress can be straightforwardly enforced on a state space. There, a system is already considered as a whole, and it turns out that maximal progress can be applied on a state by state basis. At first, this approach may seem unconventional, since traditionally, maximal progress is added to a timed language through an operator. We point out, however, that conceptually, maximal progress has a lot in common with partial order reduction (Peled et al., 1996), many forms of which could in fact also be achieved with an extra operator; but in that field, it is custom to embed it in state space generation.

The benefit of having a specific maximal progress reduction tool is that the existing state space generation toolset can remain as is. In Figure 3.4, it is shown how maximal progress can be achieved in a state space; at each state s, outgoing *tick* transitions are only allowed if there are no outgoing non-*tick* transitions; therefore, if there are outgoing non-*tick* transitions, all possible outgoing *tick* transitions must be pruned away. The maximal progress reduction of a state space is formally defined in Definition 7. There, $s \rightarrow_T^* s'$ denotes that s' is reachable from s *through the set of transitions* T, i.e. there are $s_0,\ldots,s_n \in \mathscr{S}$ and $\ell_0,\ldots,\ell_{n+1} \in \mathscr{A}$, with $n \geq 0$, such that $s \xrightarrow{\ell_0} s_0 \in T$, $s_i \xrightarrow{\ell_{i+1}} s_{i+1} \in T$ for $0 \leq i \leq n-1$, and $s_n \xrightarrow{\ell_{n+1}} s' \in T$.

Definition 7 (Maximal progress reduction of a state space). *Given a state space* $\mathscr{M} = (\mathscr{S},\mathscr{A},\mathscr{T},\mathscr{I})$*, we call* $\mathscr{M}' = (\mathscr{S}',\mathscr{A}',\mathscr{T}',\mathscr{I}')$ *the* maximal progress reduced form *of* $\mathscr{M}$ *iff with* $\hat{\mathscr{T}} = \{(s,\ell,s') \in \mathscr{T} \mid \ell \neq \mathit{tick}\} \cup \{(s,\mathit{tick},s') \in \mathscr{T} \mid \neg\exists \ell \in H, s'' \in \mathscr{S}.(s,\ell,s'') \in \mathscr{T}\}$ *we have*

1. $\mathscr{A}' = \mathscr{A}$;

2. $\mathscr{I}' = \mathscr{I}$;

3. $\mathscr{S}' = \{s \in \mathscr{S} \mid \exists s' \in \mathscr{I}.s' \rightarrow_{\hat{\mathscr{T}}}^* s\}$;

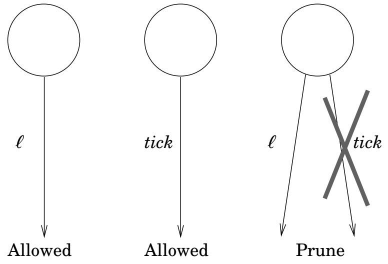

Figure 3.4: Applying maximal progress on a state space

4. $\mathscr{T}' = \{(s, \ell, s') \in \hat{\mathscr{T}} \mid s \in \mathscr{S}' \wedge s' \in \mathscr{S}'\}$.

In requirement 3 of Definition 7, we do not simply state $\mathscr{S}' = \mathscr{S}$ in order to avoid the inclusion of states that have become unreachable through transitions which do not violate maximal progress.

If so desired, maximal progress can now be dealt with using a specialised tool. The drawback of this approach, however, is that it means that the whole state space, including all undesired behaviour violating maximal progress, must be generated before the reduction can be done. In practice, if we deal with large problems, the undesired portion of the state space can be very big. To avoid generating all this extra behaviour, we can move to a third possibility to enforce maximal progress: on-the-fly pruning while generating the state space.

On-the-fly pruning of the state space As maximal progress can be applied in a state space on a state by state basis, it is very well possible to design a specialised state space generation algorithm which applies maximal progress on-the-fly. This approach avoids generating unreachable parts of the state space.

Algorithm 6 shows a breadth-first search with maximal progress. It cuts away undesired parts of the state space, i.e. those which are reached through unnecessary delays. The connection between a specification $\mathcal{M}$ and this algorithm is achieved through linearising $\mathcal{M}$ to a single LPE X. The initial set $\mathscr{I}$ is derived from X and $\mathcal{I}$, and furthermore, X is used to find successor states (since the set of successors of a process $X(d)$ can be obtained by determining $en_{\mathcal{M}}(X(d))$.

3.4 The Translation Scheme

In the current section we present a scheme for the translation of χ_t specifications to μCRL specifications. Here, a χ_t specification is mapped to a μCRL specification and

Algorithm 6 Maximal progress breadth-first state space generation

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{I}), H \subseteq \mathscr{A} \setminus \{tick, tock, tock', tick'\}$
Ensure: $true$
  $i \leftarrow 0$
  $\mathscr{L}_i \leftarrow \mathscr{I}$
  while $\mathscr{L}_i \neq \emptyset$ do
    $\mathscr{L}_{i+1} \leftarrow \emptyset$
    for all $s \in \mathscr{L}_i$ do
      if $H \cap \{\ell \in \mathscr{A} \mid \exists s' \in \mathscr{S}. s \xrightarrow{\ell} s'\} = \emptyset$ then
        $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \cup \{s' \in \mathscr{S} \mid \exists \ell \in \mathscr{A}. s \xrightarrow{\ell} s'\}$
      else
        $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \cup \{s' \in \mathscr{S} \mid \exists \ell \in \mathscr{A}. s \xrightarrow{\ell} s' \wedge \ell \neq tick\}$
      end if
    end for
    $i \leftarrow i + 1$
    $\mathscr{L}_i \leftarrow \mathscr{L}_i \setminus \bigcup_{j=0}^{i-1} \mathscr{L}_j$
  end while
  return $true$
```

a χ_{t} process is mapped to an LPE. We start by extending the definition of an LPE, to include successful termination.

3.4.1 Linear Process Equations

In this chapter we use a slightly extended version of the LPE definition as stated in Chapter 2. Here, an LPE is of the form as defined in Definition 8.

Definition 8 (LPE with successful termination). *A* linear process equation with successful termination *is of the following form:*

$$\begin{aligned} X(d:\mathbb{D}) \quad = \quad & \sum_{i\in I} \sum_{e_i:\mathbb{D}_i} a_i(f_i(d,e_i)) \cdot X(g_i(d,e_i)) \lhd h_i(d,e_i) \rhd \delta \; + \\ & \sum_{i\in I'} \sum_{e_i:\mathbb{D}'_i} a'_i(f'_i(d,e_i)) \cdot \surd(g'_i(d,e_i)) \lhd h'_i(d,e_i) \rhd \delta \end{aligned}$$

where I, I' are finite index sets, $\mathbb{D}, \mathbb{D}_i, \mathbb{D}'_i, \mathbb{D}_{a_i}$ and $\mathbb{D}_{a'_i}$ are data types, $a_i, a'_i \in \mathcal{A} \cup \{\tau, \delta\}$, $a_i : \mathbb{D}_{a_i}$, $a'_i : \mathbb{D}_{a'_i}$, $f_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{D}_{a_i}$, $f'_i : \mathbb{D} \times \mathbb{D}'_i \to \mathbb{D}_{a'_i}$, $g_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{D}$, $g'_i : \mathbb{D} \times \mathbb{D}'_i \to \mathbb{D}$, $h_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{B}$ and $h'_i : \mathbb{D} \times \mathbb{D}'_i \to \mathbb{B}$.

The extension to Definition 4 is the usage of $\surd$, which introduces successful termination. The notation $\surd(g'_i(d, ei))$ should be read as "the process enters state $g'_i(d, e_i)$

after which it successfully terminates" (the process has reached an end state).[3]

In this chapter, we present a translation from χ_t processes to μCRL LPEs with successful termination. The translation is given as a function $T : \chi_\mathrm{t} \rightarrow \mu$CRL by induction on the term structure of the χ_t process, where T takes a χ_t process as input and gives an LPE with successful termination as output.

In general, when translating a χ_t process p to an LPE, the variables s_i in the scope operator of p (if used) should be translated to parameters of the LPE (in other words, should become part of the data parameter $d : \mathbb{D}$). Local channels of p also appear in its scope, but these should not be included in the LPE. Instead, they will be represented on a global level in the μCRL specification (see the upcoming section).

In both χ_t and μCRL one can use data types. For every data type in χ_t, one can easily define a corresponding abstract data type in μCRL.

3.4.2 A χ_t Specification

The parallel composition operator and the encapsulation operator are here placed in one section, since both of them are used in a particular way within a μCRL specification. More specifically, in the μCRL toolset, these operators are only allowed to be used in the initialisation line. What follows are guidelines to translate these operators and which assumptions are made during the remainder of this chapter.

Let us consider the definition of a χ_t specification again, with $p = p_0 \parallel \ldots \parallel p_n$, $n \geq 0$. The variables s_i declared in a χ_t specification are shared among the p_i. These cannot be translated in a straightforward fashion, since μCRL does not have shared variables. It is possible however to achieve the same results by having all the processes working with these variables maintain their own local copies of these variables and using broadcasts whenever an assignment takes place. More on this in Section 3.4.13.

Each channel a which is declared in a χ_t specification should be translated to a send action sa, a corresponding receive action ra, and a corresponding communication action ca, with $sa, ra, ca \in \mathcal{A}$ of the μCRL specification $\mathcal{M}$. Furthermore, we need $(sa,ra,ca) \in \mathcal{C}$.

As already stated, the initialisation predicate i consists of statements of the form $s_i = e_i$. For μCRL, these initialisations need to appear in $\mathcal{I}$. In order to keep track of the initialisations, we construct a set V while translating, containing pairs of (translated) variable names and their values (s_i, e_i). Next, we describe how to construct $\mathcal{I}$. In practice, this is done as the final step in translating a χ_t specification. Once $\mathcal{I}$ needs to be constructed, we consult the initialisation set V for the correct initial values of the variables to be set in $\mathcal{I}$.

In the remainder of this chapter, we restrict the use of parallel composition to the

[3] We are aware of the fact that such an extension calls for a new structural operational semantics and a new set of axioms. These can be created by extending the structural operational semantics and the axioms presented by Groote and Ponse (1995) and Groote and Reniers (2001). Why we refrain from doing so here is explained in Section 4.8.

top level of a specification. In other words, looking at the definition of a χ_{t} specification, we say that p is of the form $p_0 \parallel \ldots \parallel p_n$ with $n \geq 0$ and the p_i being processes without parallel composition. This is not a harsh restriction, since there exists a χ_{t} lineariser (created by Theunissen (2006)) which can remove nested parallel composition from the p_i. Moreover, in case studies nested parallelism is hardly ever encountered. The usage of the parallel composition operator at the top level of a χ_{t} specification can be translated in a straightforward fashion. Recall $\mathcal{I}$ of a μCRL specification $\mathcal{M}$ to be $\partial_H(X_0(d_0) \parallel \ldots \parallel X_m(d_m))$. We translate the p_i directly to $X_i(d_i)$ using the translation function T. Now, we have the following:

$$\mathcal{I} = \partial_H(T(p_0) \overset{\top}{\parallel} \ldots \overset{\top}{\parallel} T(p_n))$$

with $H = \{tock\} \cup \{sa, ra \mid (sa, ra, ca) \in \mathcal{C}\}$. Consider that $T(p_i) = X_i(d_i)$. Once the p_i are translated, the initial values of the d_i can be obtained from V. Variables not appearing in V are set to a default value.

The encapsulation operator of χ_{t} (∂_A) is implicitly used at the top level of a χ_{t} specification, applied on a given set of actions A. It can be translated by using the encapsulation operator of μCRL (∂_H), making the set H equal to the set A. Also, the urgent communication operator is used at the top level of a specification. For more on this operator, see Section 3.3.1.

3.4.3 Atomic Processes

The multi-assignment process In μCRL, assignments take place by using recursion or calling a new process in which the new value of the changed variable is given as a parameter. Therefore, process $x_n := e_n$ is translated to (i.e. $T(x_n := e_n)$ is equal to) $X(d : \mathbb{D}) = \tau \cdot \surd(d[{}^{e_1}/_{x_1}, \ldots, {}^{e_n}/_{x_n}])$, in which $\surd(d[{}^{e_1}/_{x_1}, \ldots, {}^{e_n}/_{x_n}])$ means that you end up in a state where the values $e_1, \ldots, e_n$ have been substituted for the variables $x_1, \ldots, x_n$, respectively, while the other variables in the state remain unchanged.

The skip process The process skip performs the internal action τ. This can be translated into an LPE by using the τ action. The translation T(skip) then becomes $X(d : \mathbb{D}) = \tau \cdot \surd(d)$.

The send process In μCRL, channels are not available as a concept, like they are in χ_{t}. Instead, a similar functionality can be obtained by defining actions and having them synchronise with each other. Traditionally, sending a command like e.g. *test* can be done by using an action *stest* (the *s* stands for *send*). This command can be received by another process with the action *rtest*, where the *r* means *receive*. The actions *stest* and *rtest* must be defined in the specification, together with a communication rule, saying that a send over the test 'channel' together with a receive over this 'channel'

Table 3.1: Translation of send processes

χ_t term p	μCRL LPE $T(p)$
$h\,!\,e_n$	$X(d:\mathbb{D}) = sh(e_1,\ldots,e_n)\cdot\surd(d) + tock\cdot X(d)$
$h\,!!\,e_n$	$X(d:\mathbb{D}) = sh(e_1,\ldots,e_n)\cdot\surd(d)$
$h\,!$	$X(d:\mathbb{D}) = sh\cdot\surd(d) + tock\cdot X(d)$
$h\,!!$	$X(d:\mathbb{D}) = sh\cdot\surd(d)$

leads to a communication (an action called *ctest*). It is important that when describing the initial situation one encapsulates the send and receive actions in order to force communication between the two.

Taking into account that a send process $h\,!\,e_n$ should be delayable, $T(h\,!\,e_n)$ equals $X(d:\mathbb{D}) = sh(e_1,\ldots,e_n)\cdot\surd(d) + tock\cdot X(d)$ with $e_i \in \mathbb{D}_i$ and $sh:\mathbb{D}_1\times\ldots\times\mathbb{D}_n$ being the action of sending something over the channel h. The *tock* alternative is used to express delayability.

All variants of the χ_t send process can be found in Table 3.1. Each of them is accompanied by a μCRL translation. The translations should be evident, given that all variants of the basic send process $h\,!\,e_n$ can only differ in delayability and/or the sending of data.

In χ_t, communications have priority over the passage of time. This behaviour is enforced by using the urgent communication operator implicitly. Having translated a χ_t specification, it is therefore necessary to process the translation in the way specified in Section 3.3.1.

The receive process As mentioned in the previous paragraph, μCRL does not work with channels, but for this, one can define send and receive actions and force them to communicate. Receive actions traditionally begin with the letter r.

Therefore, for process $h\,?\,x_n$, $T(h\,?\,x_n)$ equals

$$X(d:\mathbb{D}) = \sum_{y_1:\mathbb{D}_1},\cdots,\sum_{y_n:\mathbb{D}_n}(rh(y_1,\ldots,y_n)\cdot\surd(d[^{y_1}/_{x_1},\ldots,^{y_n}/_{x_n}])) + tock\cdot X(d)$$

with $\mathbb{D}_i$ being the type of the variables x_i and y_i, for all i $(1\le i\le n)$. As in the translation of the send processes, the *tock* alternative is used to express delayability.

All variants of the χ_t receive process can be found in Table 3.2. Each of them is accompanied by a μCRL translation. The translations should be evident, given that all variants of the basic receive process $h\,?\,e$ can only differ in delayability and/or the receiving of data.

Concerning communications having priority over the passage of time, a remark similar to the one in the previous paragraph holds for the receive process.

Table 3.2: Translation of receive processes

χ_t term p	μCRL LPE $T(p)$
$h\,?\,e_n$	$X(d:\mathbb{D}) = \sum_{y_1:\mathbb{D}_1}\cdots\sum_{y_n:\mathbb{D}_n}(rh(y_1,\ldots,y_n)\cdot\surd(d[^{y_1}/_{x_1},\ldots,^{y_n}/_{x_n}]))$ $+tock\cdot X(d)$
$h\,??\,e_n$	$X(d:\mathbb{D}) = \sum_{y_1:\mathbb{D}_1}\cdots\sum_{y_n:\mathbb{D}_n}(rh(y_1,\ldots,y_n)\cdot\surd(d[^{y_1}/_{x_1},\ldots,^{y_n}/_{x_n}]))$
$h\,?$	$X(d:\mathbb{D}) = rh\cdot\surd(d) + tock\cdot X(d)$
$h\,??$	$X(d:\mathbb{D}) = rh\cdot\surd(d)$

The delay process The translation of the delay process Δt is highly dependent on the timing mechanism used here in μCRL. Therefore the reader should be aware of this timing mechanism as described in Section 3.3. Note that while χ_t uses continuous time, this timing mechanism only considers discrete time. Therefore, only the “discrete time part” of χ_t can be translated.

If we restrict the possible values of t in Δt to the natural numbers, then $T(\Delta t)$ is as follows, where t_0 and t_1 are timers:

$$X(d:\mathbb{D}, t_0:\mathbb{T}, t_1:\mathbb{T}) = tick\cdot X(d, t_0, t_1 - 1) \lhd t_1 > 0 \rhd \delta + \tau\cdot\surd(d, t_0, t_1) \lhd t_1 = 0 \rhd \delta$$

We cannot set the initial values of t_0 and t_1 in process X. We have to do that in $\mathcal{I}$. Therefore, we add (t_0, t) and (t_1, t) to V, thereby storing that both variables t_0 and t_1 initially have value t. Timer t_0 is used to be able to reset timer t_1 to its initial value, if so desired. This actually occurs in case repetition is present in a process (see Section 3.4.9).

3.4.4 Delay Operator

When discussing the translation of χ_t operators in the upcoming sections, two LPEs P and Q are used:

$$P(d:\mathbb{D}) = \sum_{i\in I}\sum_{e_i:\mathbb{D}_i} a_i(f_{P_i}(d,e_i))\cdot P(g_{P_i}(d,e_i)) \lhd h_{P_i}(d,e_i) \rhd \delta +$$
$$\sum_{i\in I'}\sum_{e_i\in\mathbb{D}'_i} a'_i(f'_{P_i}(d,e_i))\cdot\surd(g'_{P_i}(d,e_i)) \lhd h'_{P_i}(d,e_i) \rhd \delta$$

$$Q(d':\mathbb{D}') = \sum_{j\in J}\sum_{e_j:\mathbb{D}_j} a_j(f_{Q_j}(d',e_j))\cdot Q(g_{Q_j}(d',e_j)) \lhd h_{Q_j}(d',e_j) \rhd \delta +$$
$$\sum_{j\in J'}\sum_{e_j:\mathbb{D}'_j} a'_j(f'_{Q_j}(d',e_j))\cdot\surd(g'_{Q_j}(d',e_j)) \lhd h'_{Q_j}(d',e_j) \rhd \delta$$

We avoid name clashes of variables in d and d' when it is necessary to combine the two LPEs.

Consider a process $\Delta_t(p)$ with the LPE $P = T(p)$. Then $T(\Delta_t(p))$ is defined as in Figure 3.5, where t_0 and t_1 are timers.

$$\begin{aligned}
&X(d : \mathbb{D}, t_0 : \mathbb{T}, t_1 : \mathbb{T}) = \\
&\quad tick{\cdot}X(d, t_0, t_1 - 1) \lhd t_1 > 0 \rhd \delta\ + \\
&\quad \sum_{i \in I} \sum_{e_i : \mathbb{D}_i} a_i(f_{P_i}(d, e_i)){\cdot}X(g_{P_i}(e_i), t_0, t_1) \lhd h_{P_i}(d, e_i) \wedge t_1 = 0 \rhd \delta\ + \\
&\quad \sum_{i \in I'} \sum_{e_i : \mathbb{D}'_i} a_i(f'_{P_i}(d, e_i)){\cdot}\surd(g'_{P_i}(d, e_i), t_0, t_1) \lhd h'_{P_i}(d, e_i) \wedge t_1 = 0 \rhd \delta
\end{aligned}$$

Figure 3.5: Translation of $\Delta_t(p)$

Basically the following things have been done to combine the delay and the LPE P:

1. The counters t_0 and t_1 have been introduced. They are used in the same way as they are in the delay process. We add (t_0, t) and (t_1, t) to the initialisation set V.
2. The guards of the lines that originate from LPE P have been extended with the boolean expression $t = 0$.

3.4.5 Delay Enabling Operator

Assume we have a process $[p]$ with the LPE $P = T(p)$. Then $T([p])$ is defined as in Figure 3.6.

$$\begin{aligned}
&X(n : \mathbb{N}, d : \mathbb{D}) = \\
&\quad \sum_{i \in I} \sum_{e_i : \mathbb{D}_i} a_i(f_{P_i}(d, e_i)){\cdot}X(1, g_{P_i}(d, e_i)) \lhd h_{P_i}(d, e_i) \rhd \delta\ + \\
&\quad \sum_{i \in I'} \sum_{e_i : \mathbb{D}'_i} a_i(f'_{P_i}(d, e_i)){\cdot}\surd(0, g'_{P_i}(d, e_i)) \lhd h'_{P_i}(d, e_i) \rhd \delta\ + \\
&\quad tock{\cdot}X(n, d) \lhd n = 0 \rhd \delta
\end{aligned}$$

Figure 3.6: Translation of $[p]$

A counter n has been introduced. It has type $\mathbb{N}$, but in practise $n \in \{0, 1\}$. This counter is set to 0 before executing X ($(n, 0)$ is added to the initialisation set V) and is used here to initially allow the LPE to delay. As soon as an action originally from the LPE P has been executed, and if after this execution an end state has not been reached, counter n is set to 1, resulting in the added *tock* action being disabled.

3.4.6 Guard Operator

Consider a process $b \to p$ with b being a boolean expression. Say LPE $P = T(p)$. As mentioned earlier, we say that the finite index set $I = I_N \cup I_C$ with $I_N \cap I_C = \emptyset$, I_N being the set of indices of actions in P which are neither *tick* nor *tock* actions (i.e. they are 'normal' actions) and I_C being the set of indices of actions in P which are either *tick* or *tock* actions (i.e. they are 'clock' actions). For I' we do not have to do a similar thing since I'_C will always be empty. A process never terminates after executing a *tick* or *tock* action; after a *tick* action there is always eventually a normal action (see the translations of the delay process and the delay operator) and *tock* actions only occur in self-loops.

Now $T(b \to p)$ is defined as stated in Figure 3.7.

$$
\begin{aligned}
&X(n:\mathbb{N}, d:\mathbb{D}) = \\
&\quad \sum_{i \in I_N} \sum_{e_i:\mathbb{D}_i} a_i(f_{P_i}(d,e_i)) \cdot X(1, g_{P_i}(d,e_i)) \lhd h_{P_i}(d,e_i) \wedge (n = 1 \vee b) \rhd \delta \,+ \\
&\quad \sum_{i \in I_C} \sum_{e_i:\mathbb{D}_i} a_i \cdot X(n, g_{P_i}(d,e_i)) \lhd h_{P_i}(d,e_i) \wedge (n = 1 \vee b) \rhd \delta \,+ \\
&\quad \sum_{i \in I'} \sum_{e_i:\mathbb{D}'_i} a_i(f'_{P_i}(d,e_i)) \cdot \surd(0, g'_{P_i}(d,e_i)) \lhd h'_{P_i}(d,e_i) \wedge (n = 1 \vee b) \rhd \delta \,+ \\
&\quad \mathit{tock} \cdot X(n,d) \lhd n = 0 \wedge \neg b \rhd \delta
\end{aligned}
$$

Figure 3.7: Translation of $b \to p$

Basically the following things have been done to combine the boolean expression b and the LPE P:

1. A counter n has been introduced. It has type $\mathbb{N}$, but in reachable states $n \in \{0, 1\}$. This counter is initially 0 ($(n, 0)$ is added to V) and is used here to regulate that only initially the value of b is important.

2. Notice the difference between the first and the second line: Instead of being set to 1 the counter n is unchanged. This is very important when $n = 0$, since this means that the value of the boolean expression b remains important in the next time unit.

3. In the third line n is reset to 0. Why this is done can be read in Section 3.4.9 on the repetition operator.

4. In the fourth line it is expressed that if b does not hold and no 'normal' action has been executed yet (i.e. $n = 0$) this process can delay one time unit without changing the current state.

5. In all lines the guard has been expanded with equations concerning n and b to express that one may only start executing 'normal' actions if b holds.

3.4.7 Sequential Composition Operator

Assume we have the χ_t process $p;\ q$ with the LPE $P = T(p)$ and the LPE $Q = T(q)$. Now we define $T(p;\ q)$ as in Figure 3.8. A counter n has been introduced to regulate the order of execution. Initially this counter has value 0 ($(n,0)$ is added to V), thereby enabling the execution of the actions originally from the LPE P. At those points where P terminates successfully, n is set to 1, disabling the execution of actions from P and enabling the execution of actions from Q.

$$\begin{aligned}
&X(n : \mathbb{N}, d : \mathbb{D}, d' : \mathbb{D}') = \\
&\quad \sum_{i \in I} \sum_{e_i : \mathbb{D}_i} a_i(f_{P_i}(d, e_i)) \cdot X(0, g_{P_i}(d, e_i), d') \lhd h_{P_i}(d, e_i) \wedge n = 0 \rhd \delta \, + \\
&\quad \sum_{i \in I'} \sum_{e_i \in \mathbb{D}'_i} a_i(f'_{P_i}(d, e_i)) \cdot X(1, g'_{P_i}(d, e_i), d') \lhd h'_{P_i}(d, e_i) \wedge n = 0 \rhd \delta \, + \\
&\quad \sum_{j \in J} \sum_{e_j : \mathbb{D}_j} a_j(f_{Q_j}(d', e_j)) \cdot X(1, d, g_{Q_j}(d', e_j)) \lhd h_{Q_j}(d', e_j) \wedge n = 1 \rhd \delta \, + \\
&\quad \sum_{j \in J'} \sum_{e_j : \mathbb{D}'_j} a_j(f'_{Q_j}(d', e_j)) \cdot \surd(0, d, g'_{Q_j}(d', e_j)) \lhd h'_{Q_j}(d', e_j) \wedge n = 1 \rhd \delta
\end{aligned}$$

Figure 3.8: Translation of $p;\ q$

3.4.8 Alternative Composition Operator

In this section we give a translation of the χ_t process $p \,[]\, q$, which non-deterministically chooses between the processes p and q. At first glance providing a translation for this does not seem to be more difficult than providing one for the sequential composition. This, however, turns out to be untrue, due to the timing mechanism of χ_t; if both alternatives p and q can delay, then they delay *together* and no choice is made. If only one alternative can delay and furthermore no actions can be executed at all, then there is a deadlock. This constitutes strong time determinism.

Say we have the χ_t process $p \mathbin{[]} q$ with the LPE $P = T(p)$ and the LPE $Q = T(q)$. Furthermore we say that the finite index set $I = I_N \cup I_C$ (similar to Section 3.4.6). Finally, we say that $I_C = I_{C1} \cup I_{C2}$ with $I_{C1} \cap I_{C2} = \emptyset$, I_{C1} being the set of indices of occurrences of *tick* in P and I_{C2} being the set of indices of occurrences of *tock* in P. In a similar way we define $J = J_N \cup J_C$ and $J_C = J_{C1} \cup J_{C2}$. Now we define $T(p \mathbin{[]} q)$ as shown in Figure 3.9.

$$
\begin{aligned}
&X(n : \mathbb{N}, d : \mathbb{D}, d' : \mathbb{D}') = \\
&\quad \sum_{i \in I_N} \sum_{e_i : \mathbb{D}_i} a_i(f_{P_i}(d, e_i)) \cdot X(1, g_{P_i}(d, e_i), d') \lhd h_{P_i}(d, e_i) \land (n = 0 \lor n = 1) \rhd \delta + \\
&\quad \sum_{i \in I_C} \sum_{e_i : \mathbb{D}_i} a_i(f_{P_i}(d, e_i)) \cdot X(n, g_{P_i}(d, e_i), d') \lhd h_{P_i}(d, e_i) \land n = 1 \rhd \delta + \\
&\quad \sum_{i \in I'} \sum_{e_i : \mathbb{D}'_i} a_i(f'_{P_i}(d, e_i)) \cdot \surd(0, g'_{P_i}(d, e_i), d') \lhd h'_{P_i}(d, e_i) \land (n = 0 \lor n = 1) \rhd \delta + \\
&\quad \sum_{j \in J_N} \sum_{e_j : \mathbb{D}_j} a_j(f_{Q_j}(d', e_j)) \cdot X(2, d, g_{Q_j}(d', e_j)) \lhd h_{Q_j}(d', e_j) \land (n = 0 \lor n = 2) \rhd \delta + \\
&\quad \sum_{j \in J_C} \sum_{e_j : \mathbb{D}_j} a_j(f_{Q_j}(d', e_j)) \cdot X(n, d, g_{Q_j}(d', e_j)) \lhd h_{Q_j}(d', e_j) \land n = 2 \rhd \delta + \\
&\quad \sum_{j \in J'} \sum_{e_j : \mathbb{D}'_j} a_j(f'_{Q_j}(d', e_j)) \cdot \surd(0, d, g'_{Q_j}(d', e_j)) \lhd h'_{Q_j}(d', e_j) \land (n = 0 \lor n = 2) \rhd \delta + \\
&\quad \sum_{i \in I_{C1}} \sum_{j \in J_{C2}} \sum_{e_i : \mathbb{D}_i} \sum_{e_j : \mathbb{D}_j} \mathit{tick} \cdot X(n, g_{P_i}(d, e_i), g_{Q_j}(d', e_j)) \lhd h_{P_i}(d, e_i) \land h_{Q_j}(d', e_j) \land n = 0 \rhd \delta + \\
&\quad \sum_{i \in I_{C2}} \sum_{j \in J_{C1}} \sum_{e_i : \mathbb{D}_i} \sum_{e_j : \mathbb{D}_j} \mathit{tick} \cdot X(n, g_{P_i}(d, e_i), g_{Q_j}(d', e_j)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \lhd h_{P_i}(d, e_i) \land h_{Q_j}(d', e_j) \land n = 0 \rhd \delta + \\
&\quad \sum_{i \in I_{C1}} \sum_{j \in J_{C1}} \sum_{e_i : \mathbb{D}_i} \sum_{e_j : \mathbb{D}_j} \mathit{tick} \cdot X(n, g_{P_i}(d, e_i), g_{Q_j}(d', e_j)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \lhd h_{P_i}(d, e_i) \land h_{Q_j}(d', e_i) \land n = 0 \rhd \delta + \\
&\quad \sum_{i \in I_{C2}} \sum_{j \in J_{C2}} \sum_{e_i : \mathbb{D}_i} \sum_{e_j : \mathbb{D}_j} \mathit{tock} \cdot X(n, d, d') \lhd h_{P_i}(d, e) \land h_{Q_j}(d', e) \land n = 0 \rhd \delta
\end{aligned}
$$

Figure 3.9: Translation of $p \mathbin{[]} q$

Basically the following things have been done to combine the LPEs P and Q:

1. A counter n has been introduced. It has type $\mathbb{N}$, but in reachable states $n \in \{0, 1, 2\}$. Initially this counter has value 0 ($(n, 0)$ is added to V).
2. In the first line we find the 'normal' actions that originally are not at the end of process P (i.e. P does not terminate after performing one of these actions). Since

n initially equals 0, some of these actions can be performed in the beginning of executing X (where $h_{P_i}(d, e_i)$ holds).

3. In the second line we find all occurrences of *tick* and *tock* in LPE P. It is very important to note that the usage of n in the guard (only considering $n = 1$) leads to guards which are always false in cases where *tick* and *tock* actions are enabled in the beginning of executing P. This is because initially n does not equal 1, but 0 and after that, when n does equal 1, $h_{P_i}(d)$ does not hold. This results in *tick* and *tock* occurrences at the beginning of P (in terms of execution order) being effectively removed from process X.

4. In the third line we find the 'normal' actions as we did in the first line, only after executing these actions P originally terminates. As we see here, X terminates as well.

5. In the fourth, fifth and sixth line we find situations similar to the first, second and third line respectively, only now they concern actions from process Q.

6. In line seven we combine all occurrences of *tick* in process P with all occurrences of *tock* in process Q. Together, these form *tick* occurrences in X, where the new state is defined by using the two functions g_{P_i} and g_{Q_j} and the guard is the conjunction of the guards of the occurrences being combined together with the expression $n = 0$. This last expression $n = 0$ effectively makes all guards equal to F, except in those cases where both the *tick* and the *tock* occurrence are at the beginning (execution-wise) of P and Q, respectively. The reason for this is similar to the one given for the second line.

7. In the same way as is done in the previous line, the remaining lines combine *tock* occurrences in P with *tick* occurrences in Q, *tick* occurrences in P and Q and *tock* occurrences in P and Q respectively.

So in line two and five the *tick* and *tock* occurrences from the beginning of P and Q are practically removed, only to appear in a combined form in lines seven, eight and nine. This reflects what happens in the χ_t process $p \mathbin{[]} q$, where p and q delay together if they can both delay and no delay will happen if one of them cannot.

3.4.9 Repetition Operator

When executing the χ_t process $*p$, the process p gets executed in sequence infinitely often. This construction needs to be translated using recursion.

Say we have a χ_t process $*p$ where the LPE $P = T(p)$. Now we define $T(*p)$ as in Figure 3.10. In the LPE the termination ($\surd$) has been replaced by X, resulting in executing X from the beginning again every time X has executed the final action in

$$
\begin{aligned}
&X(d : \mathbb{D}) = \\
&\quad \sum_{i \in I} \sum_{e_i \in \mathbb{D}_i} a_i(f_{P_i}(d, e_i)) \cdot X(g_{P_i}(d, e_i)) \lhd h_{P_i}(d, e_i) \rhd \delta + \\
&\quad \sum_{i \in I'} \sum_{e_i \in \mathbb{D}'_i} a_i(f'_{P_i}(d, e_i)) \cdot X(reset(g'_{P_i}(d, e_i))) \lhd h'_{P_i}(d, e_i) \rhd \delta
\end{aligned}
$$

Figure 3.10: Translation of $*p$

the LPE. When repeating the execution the LPE automatically begins with the first action, which is ensured by the translation of p (note that in all translations of the operators, counters get their initial value back at termination). However, note that the new state, when the process starts repeating, is subject to the function $reset : \mathbb{D} \to \mathbb{D}$. We define this function as follows, but first note that the state of a process as presented in this chapter is referred to as d and that it is of the form $d' : \mathbb{D}, \overrightarrow{(t_0 : \mathbb{T}, t_1 : \mathbb{T})}$ with $\overrightarrow{(t_0 : \mathbb{T}, t_1 : \mathbb{T})}$ a vector of pairs $(t_0 : \mathbb{T}, t_1 : \mathbb{T})$. The function $reset : \mathbb{D} \to \mathbb{D}$ is now defined as:

$$reset(d, \overrightarrow{(t_0, t_1)}) = d, \overrightarrow{(t_0, t_0)}$$

This function is applied to ensure that timers are reset to their initial value. Although timers are often already reset at the termination of a process (see the translations of the other processes), in an alternative composition it may occur that not all timers are reset if this function is not applied.

3.4.10 Guarded Repetition Operator

Say we have the χ_t process $*b : p$ with the LPE $P = T(p)$ and b is translated. Now we define $T(*b : p)$ as presented in Figure 3.11.

Basically the following things have been done to get the process X:

1. A counter n has been introduced. It has type $\mathbb{N}$ but in reachable states $n \in \{0, 1\}$. Initially this counter has value 0 ($(n, 0)$ is added to V).

2. In the first line, the process can do a τ action if the boolean expression b evaluates to T. After this, the actions of LPE P can be executed.

3. In the second line, $P(g_{P_i}(d))$ is replaced by $X(1, g_{P_i}(d))$.

4. In the third line, $\surd(g'_{P_i}(d))$ is replaced by $X(0, reset(g'_{P_i}(d, e_i)))$. The function $reset : \mathbb{D} \to \mathbb{D}$ is as described in Section 3.4.9.

$$
\begin{aligned}
&X(n:\mathbb{N}, d:\mathbb{D}) = \\
&\quad \tau{\cdot}X(1,d) \lhd n = 0 \wedge b \rhd \delta\ + \\
&\quad \sum_{i\in I}\sum_{e_i:\mathbb{D}_i} a_i(f_{P_i}(d,e_i)){\cdot}X(n, g_{P_i}(d,e_i)) \lhd h_{P_i}(d,e_i) \wedge n = 1 \rhd \delta\ + \\
&\quad \sum_{i\in I'}\sum_{e_i:\mathbb{D}'_i} a_i(f'_{P_i}(d,e_i)){\cdot}X(0, \mathit{reset}(g'_{P_i}(d,e_i))) \lhd h'_{P_i}(d,e_i) \wedge n = 1 \rhd \delta\ + \\
&\quad \tau{\cdot}\surd(n,d) \lhd n = 0\ \wedge \neg b \rhd \delta
\end{aligned}
$$

Figure 3.11: Translation of $*b : p$

5. The fourth line allows the process to finish execution. Once the guard is false when trying to begin executing the actions of the original P again, the process should finish with a τ step.

3.4.11 Scope Operator

The process algebra μCRL does not have a scope operator, but the functionality of this can be found implicitly in the algebra. Note that in the χ_t process $[\![\ s \mid p\ ]\!]$, the state s is used to define local programming variables, local channels, an initialisation predicate and local recursion definitions. So in a way, a state is a quadruple consisting of variables, channels, initialisations and recursion definitions. First of all, in μCRL, the programming variables of a process can be found in its parameter d. For practical reasons, we rename local variables, if necessary, to ensure that they are globally uniquely named. Local channels are not considered in this chapter; we assume that p does not contain parallel composition, hence local channels are useless. Translation of the initialisations should, like the initialisation predicate of a χ_t specification (Section 3.4.2), be dealt with in $\mathcal{I}$. Hence, we add the appropriate tuples of variable names and their values to V. Finally, recursion definitions in s should be directly applied on p, i.e. p should be rewritten, such that it incorporates the recursion definitions. Concluding, s is captured in μCRL by the process definition $T(p)$, its recursion parameters, and the initialisation of them in $\mathcal{I}$. These techniques together translate the functionality of the scope operator.

3.4.12 Urgent Communication Operator

Finally, the urgent communication operator achieves globally applied maximal progress for all actions in χ_t specifications (see Section 3.3.1). In Section 3.3.1, we described three methods to deal with maximal progress for μCRL with *tick* actions. Suffice it to say here, that when, for instance, using on-the-fly state space generation with maximal progress, as presented in Algorithm 6, the action set H must contain all communication actions of the specification.

3.4.13 Shared Variables

In χ_t, processes in a specification can share variables defined at the top level of the specification; if one process changes the value of such a variable, the other processes may be affected by this. In μCRL there are no shared variables, but it is possible to get similar results.

Say two χ_t processes p and q share a variable named x. We translate process p to process $T(p)$ and process q to $T(q)$. Both processes maintain a local copy of the (translated) variable x. The processes can read the value of their local copy at all times, but if one of them changes the value of its copy the other one should be aware of this (and change the value of its own copy of x likewise). To make this possible in μCRL, a new action *assignx* $\in \mathcal{A}$ is introduced, which is called by a process if it changes the value of x. As a parameter the new value should be given. This action communicates with another action *updatex* $\in \mathcal{A}$, which can be executed by the other process **at all times**. This last thing is very important, since an assignment should proceed as soon at it is invoked. Once the other process can communicate via *updatex* it receives the new value for x and assigns this to its local copy.

In case there are multiple μCRL processes which have to share x, once an *assignx* action has communicated with an *updatex* action the resulting action communicates immediately with the *updatex* action of another process, such that in the end all processes are aware of the assignment. More specifically, for every x in a set of shared variables S, we define *assignx*, *assignx*$'$, *updatex*, *updatex*$'$ $\in \mathcal{A}$ and (*assignx*, *updatex*, *assignx*$'$), (*updatex*, *updatex*, *updatex* $'$) $\in \mathcal{C}$. The approach is similar to the time approach as described in Section 3.3. Again, the primed actions are used for intermediate synchronisation results, and we define an extended parallel composition operator $|^{\top}_{S}|$, which deals with both time and shared variables. The definition of $|^{\top}_{S}|$ depends on the set of shared variables S in $\mathcal{M}$. We define it as follows:

$$P \;|^{\top}_{S}|\; Q \triangleq \rho_f(\partial_H(P \parallel Q))$$

with $f = \{tick' \to tick, tock' \to tock\} \cup \{assignx' \to assignx, updatex' \to updatex \mid x \in S\}$ and $H = \{tick, tock\} \cup \{assignx, updatex \mid x \in S\}$

3.4.14 On the Correctness of the Translation Scheme

We remark here that there is no correctness proof of the translation scheme. This is addressed in more detail in Section 4.8, which provides the conclusions for both this and the subsequent chapter. Here we note that the creation of a correctness proof was outside of the scope of the work at the time of designing the translation scheme, and that it may be considered as future work. In line with this, the structural operational semantics and the set of axioms of μCRL has also not been formally extended to deal with successful termination.

Chapter 4

Modelling and Verifying Timed Systems

Youth would be an ideal state if it came later in life.

(*H.H. Asquith*)

IN THIS CHAPTER WE PROVIDE some examples of applying the translation scheme from χ_t to μCRL explained in Chapter 3 on concrete cases. First, we present a small χ_t specification P, which consists of two processes in parallel. Next we look at an industrial system called a turntable, which we have translated in order to verify a given set of properties.

4.1 A Small χ_t System

Consider the following χ_t specification P:

```
$\langle$ chan $a$ !? : bool
$|$  $[\![$ disc $i$ : nat , $i = 2$
     $|$ $*($ $i \geq 1 \rightarrow \Delta 3.0$; $a$ ! false; $i := i + 1$
     $[\,]$ $i \geq 2 \rightarrow a$ ! true; $i := i - 1$ $)]\!]$
$\|$  $[\![$ disc $b$ : bool
     $|$ $*($ $\Delta 2.0$; $a$ ? $b$ $)]\!]$
$\rangle$
```

Next we translate this specification to a μCRL specification $\mathcal{M}$. After that we linearise the translation using the μCRL toolset and introduce urgent communication.

We start by noting that the only data types used are the ones for the natural numbers and the booleans. For μCRL this means that $\mathbb{N}, \mathbb{B} \in \mathcal{D}$. Since these are standard we do not display their definitions here.

Now we detect the channels used and define appropriate actions for them in μCRL. In the χ_{t} specification we see the channel a. For this we define the actions $sa, ra, ca \in \mathcal{A}$ with sa, ra, ca: $\mathbb{B}$. Here sa stands for sending a value over channel a, while ra is used for receiving a value and ca represents communication over the channel. Furthermore, we define (sa, ra, ca), $(ra, sa, ca) \in \mathcal{C}$.

We observe that P first of all consists of two processes in parallel composition. We label these p_0 and p_1. Concerning process p_0, we know how to translate the individual actions. Using a single counter, for readability purposes, we can place the actions in the right structure (following the translation scheme we would end up with a list of counters, but here we use only one counter, which can range over all natural numbers). Furthermore we see two guards placed in an alternative composition. Finally the whole construction is subject to the repetition operator. Translating all this (and simplifying it by removing those actions which will never be executed due to their guards never being true) we get process X_0 as displayed in Figure 4.1.

$$
\begin{aligned}
&X_0(i : \mathbb{N}, n : \mathbb{N}, t_0 : \mathbb{T}, t_1 : \mathbb{T}) = \\
&\qquad \mathit{tick}\cdot X_0(i, n, t_0, t_1 - 1) \lhd t_1 > 0 \land n = 0 \land i \geq 1 \rhd \delta\, + \\
&\qquad \tau\cdot X_0(i, 1, t_0, t_0) \lhd t_1 = 0 \land n = 0 \land i \geq 1 \rhd \delta\, + \\
&\qquad \mathit{sa}(\mathrm{F})\cdot X_0(i, 2, t_0, t_1) \lhd n = 1 \rhd \delta\, + \\
&\qquad \mathit{tock}\cdot X_0(i, n, t_0, t_1) \lhd n = 1 \rhd \delta\, + \\
&\qquad \tau\cdot X_0(i + 1, 0, t_0, t_0) \lhd n = 2 \rhd \delta\, + \\
&\qquad \mathit{sa}(\mathrm{T})\cdot X_0(i, 3, t_0, t_1) \lhd n = 0 \land i \geq 2 \rhd \delta\, + \\
&\qquad \tau\cdot X_0(i - 1, 0, t_0, t_0) \lhd n = 3 \rhd \delta\, + \\
&\qquad \mathit{tock}\cdot X_0(i, n, t_0, t_1) \lhd i < 1 \land n = 0 \rhd \delta
\end{aligned}
$$

Figure 4.1: Translation of the process p_0

Finally we translate p_1. This is a very small process which is translated to the LPE given in Figure 4.2. The translation is completed when we create $\mathcal{I}$:

$$\mathcal{I} = \partial_{\{sa,ra,tock\}} X_0(2, 0, 3, 3) \mathbin{|^{\!\mathrm{T}}\!|} X_1(\mathrm{F}, 0, 2, 2)$$

Notice that we encapsulate *tock* here, which results in the fact that the synchronisation of a number of *tock* actions (without any *tick* action) will not lead to an action in the system.

$$
\begin{aligned}
&X_1(b:\mathbb{B}, n:\mathbb{N}, t_0:\mathbb{T}, t_1:\mathbb{T}) = \\
&\qquad \mathit{tick}\cdot X_1(b,n,t_0,t_1-1) \lhd t_1 > 0 \wedge n = 0 \rhd \delta \; + \\
&\qquad \tau\cdot X_1(b,1,t_0,t_0) \lhd t_1 = 0 \wedge n = 0 \rhd \delta \; + \\
&\qquad \sum_{\hat{b}:\mathbb{B}} \mathit{ra}(\hat{b})\cdot X_1(\hat{b},0,t_0,t_0) \lhd n = 1 \rhd \delta \; + \\
&\qquad \mathit{tock}\cdot X_1(b,n,t_0,t_1) \lhd n = 1 \rhd \delta
\end{aligned}
$$

Figure 4.2: Translation of the process p_1

Now that this translation is finished, we move on to linearise and post-process it to introduce urgent communication. After linearisation we have LPE X as presented in Figure 4.3, where the original parameters n, t_0 and t_1 of LPE p_1 have been renamed to n', t'_0 and t'_1 in LPE X to avoid name clashes. Post-processing LPE X for urgent communication leads to extended guards of the three lines beginning with *tick*. More specifically, the guards of these lines are extended with an extra conjunct, which is the negation of a disjunction of the guards of all the other lines (lines 4 to 9). In lines 1 and 3 this does not lead to new behaviour; when the original guards of lines 1 and 3 are true, the extra conjunct is always true as well, because in that case not one guard from lines 4 to 9 holds. In line 2 however, this is different; when the original guard of line 2 holds, the guard of line 4 may hold as well. For readability purposes, we will not show the fully post-processed LPE X here. We only add an extra conjunct to line 2, for reasons stated above. Now we conclude by providing the final LPE X in Figure 4.4,

$$
\begin{aligned}
&X(i:\mathbb{N}, n:\mathbb{N}, t_0:\mathbb{T}, t_1:\mathbb{T}, b:\mathbb{B}, n':\mathbb{N}, t'_0:\mathbb{T}, t'_1:\mathbb{T}) = \\
&\qquad \mathit{tick}\cdot X(i,n,t_0,t_1-1,b,n',t'_0,t'_1-1,b) \lhd t_1 > 0 \wedge n = 0 \wedge i \geq 1 \wedge t'_1 > 0 \wedge n' = 0 \rhd \delta \; + \\
&\qquad \mathit{tick}\cdot X(i,n,t_0,t_1-1,b,n',t'_0,t'_1) \lhd t_1 > 0 \wedge n = 0 \wedge i \geq 1 \wedge n' = 1 \rhd \delta \; + \\
&\qquad \mathit{tick}\cdot X(i,n,t_0,t_1,b,n',t'_0,t'_1-1) \lhd ((i < 1 \wedge n = 0) \vee n = 1) \wedge t'_1 > 0 \wedge n' = 0 \rhd \delta \; + \\
&\qquad \mathit{ca}(\mathrm{T})\cdot X(i,3,t_0,t_1,\mathrm{T},0,t'_0,t'_0) \lhd n = 0 \wedge i \geq 2 \wedge n' = 1 \rhd \delta \; + \\
&\qquad \mathit{ca}(\mathrm{F})\cdot X(i,2,t_0,t_1,\mathrm{F},0,t'_0,t'_0) \lhd n = 1 \wedge n' = 1 \rhd \delta \; + \\
&\qquad \tau\cdot X(i,n,t_0,t_1,b,1,t'_0,t'_0) \lhd t'_1 = 0 \wedge n' = 0 \rhd \delta \; + \\
&\qquad \tau\cdot X(i-1,0,t_0,t_0,b,n',t'_0,t'_1) \lhd n = 3 \rhd \delta \; + \\
&\qquad \tau\cdot X(i+1,0,t_0,t_0,b,n',t'_0,t'_1) \lhd n = 2 \rhd \delta \; + \\
&\qquad \tau\cdot X(i,1,t_0,t_0,b,n',t'_0,t'_1) \lhd t_1 = 0 \wedge n = 0 \wedge i \geq 1 \rhd \delta
\end{aligned}
$$

Figure 4.3: Linearisation of $\partial_{\{sa,ra,tock\}} X_0(i,n,t_0,t_1) \mathbin{\overset{\mathsf{T}}{\|}} X_1(b,n,t_0,t_1)$

which is a translation of the χ_{t} specification P with urgent communication. In the linearised specification, we have $\mathcal{I} = X(2, 0, 3, 3, \mathrm{F}, 0, 2, 2)$.

$$
\begin{array}{l}
X(i : \mathbb{N}, n : \mathbb{N}, t_0 : \mathbb{T}, t_1 : \mathbb{T}, b : \mathbb{B}, n' : \mathbb{N}, t'_0 : \mathbb{T}, t'_1 : \mathbb{T}) = \\
\qquad \mathit{tick}{\cdot}X(i, n, t_0, t_1 - 1, b, n', t'_0, t'_1 - 1) \lhd t_1 > 0 \wedge n = 0 \wedge i \geq 1 \wedge t'_1 > 0 \wedge n' = 0 \rhd \delta + \\
\qquad \mathit{tick}{\cdot}X(i, n, t_0, t_1 - 1, b, n', t'_0, t'_1 - 1) \\
\qquad\qquad\qquad \lhd t_1 > 0 \wedge n = 0 \wedge i \geq 1 \wedge n' = 1 \wedge \neg(\boldsymbol{n} = \mathbf{0} \wedge \boldsymbol{i} \geq \mathbf{2} \wedge \boldsymbol{n}' = \mathbf{1}) \rhd \delta + \\
\qquad \mathit{tick}{\cdot}X(i, n, t_0, t_1, b, n', t'_0, t'_1 - 1) \lhd ((i < 1 \wedge n = 0) \vee n = 1) \wedge t'_1 > 0 \wedge n' = 0 \rhd \delta + \\
\qquad \mathit{ca}(\mathrm{T}){\cdot}X(i, 3, t_0, t_1, \mathrm{T}, 0, t'_0, t'_0) \lhd n = 0 \wedge i \geq 2 \wedge n' = 1 \rhd \delta + \\
\qquad \mathit{ca}(\mathrm{F}){\cdot}X(i, 2, t_0, t_1, \mathrm{F}, 0, t'_0, t'_0) \lhd n = 1 \wedge n' = 1 \rhd \delta + \\
\qquad \tau{\cdot}X(i, n, t_0, t_1, b, 1, t'_0, t'_0) \lhd t'_1 = 0 \wedge n' = 0 \rhd \delta + \\
\qquad \tau{\cdot}X(i - 1, 0, t_0, t_0, b, n', t'_0, t'_1) \lhd n = 3 \rhd \delta + \\
\qquad \tau{\cdot}X(i + 1, 0, t_0, t_0, b, n', t'_0, t'_1) \lhd n = 2 \rhd \delta + \\
\qquad \tau{\cdot}X(i, 1, t_0, t_0, b, n', t'_0, t'_1) \lhd t_1 = 0 \wedge n = 0 \wedge i \geq 1 \rhd \delta
\end{array}
$$

Figure 4.4: LPE X with urgent communication

4.2 The Turntable System

The turntable system is an example of a real-life manufacturing system representing the application domain of (real-time) control research. It has appeared in the work of e.g. Bos and Kleijn (2001, 2002) and Hofkamp and Van Rooy (2003).

The turntable system consists of a round turntable, a clamp, a drill and a testing device (Figure 4.5). The turntable transports products to the drill and the testing device. The drill drills holes in the products. After drilling a hole the products are delivered to the tester, where the depth of the hole is measured, since it is possible that drilling went wrong. To control the turntable system, sensors and actuators are used. A sensor detects a physical phenomenon, and changes its state. The controller reads the state of the sensor, and sends output to actuators. The actuators translate output from the controller to a physical change in the machine.

The turntable has four slots that can hold a product. Each slot can hold at most one product and can be in input, drill, test or output position. There are three sensors attached to the turntable: the sensor $s1$ at the input position (to detect if a product has been added by the environment), the sensor $s3$ in the output position (to detect if a product has been removed by the environment) and the sensor $s2$ that detects whether the turntable has completed the turn.

The drilling module consists of the drill and the clamp. Every product should be locked before drilling and unlocked afterwards. To detect whether the clamp is locked

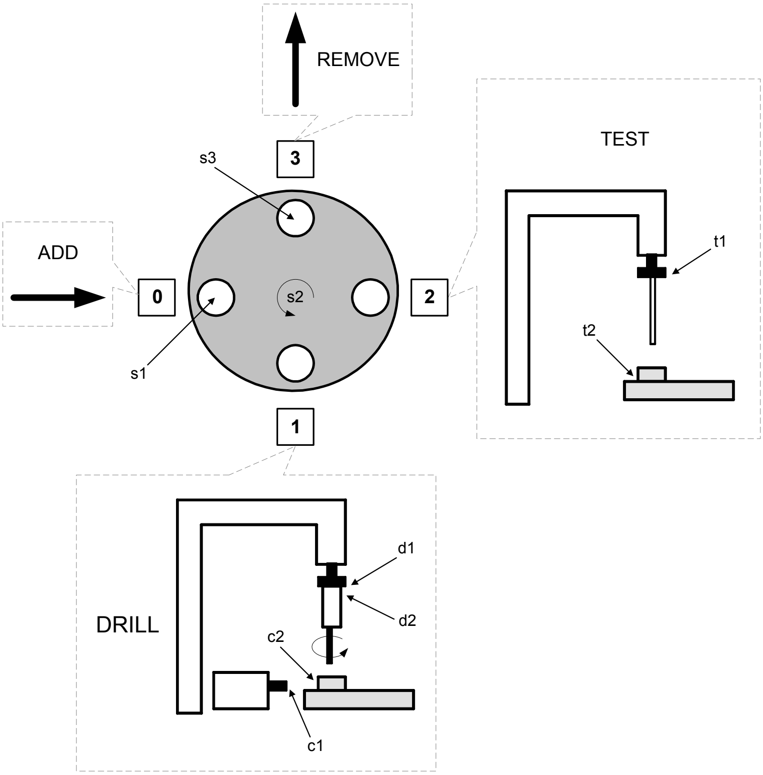

Figure 4.5: The turntable system

or not two sensors are used ($c1$ and $c2$ respectively). The drill also has two sensors to detect whether the drill is in its up ($d1$) or down ($d2$) position. These sensors are located above the surface of the turntable, so it is not possible to say whether the product has been drilled successfully or not.

In the testing position there are two sensors to detect whether the tester has reached its up ($t1$) or down ($t2$) position. If the tester has reached its down position the test result of the product is good and if the sensor at the down position did not send a signal during a certain amount of time the test result of the product is bad.

The turntable control system consists of the main controller, turntable controller, drill controller, and tester controller. The main controller supervises the other controllers and the environment. It stores current information about products and operations being performed and based on this information it issues commands to the other controllers and the environment to start operations. When operations are completed the main controller updates the information about the products.

The turntable controller gets signals from the turntable sensors and passes them to the main controller. It also starts rotation of the turntable at the command of the main controller.

The drill controller supervises the drill and the clamp. It switches the drill on/off and commands to lock/unlock the clamp or to start or stop drilling. The drill controller also gets signals from the drill and clamp sensors.

The test controller sends a signal to the tester to start the operation. Then it waits for a signal from the sensor at the down position. If the hole is not deep enough, the sensor is not activated and the current product should be rejected.

The operation-routing sequence of each product is as follows: add a product to the input position, make a turn (now the product is in the drilling position), lock the clamp, switch on the drill, drill, switch off the drill, unlock the clamp, make another turn (now the product is in the test position), test, and make a turn again (the product is in the removing position).

No product can be added if the adding slot is not empty. No drilling, testing or removing can be performed if the corresponding slot is empty. The turntable can treat up to four products at the same time, that means that the operations can be done in parallel.

Design rules and assumptions Creating the specification, we only consider "good weather" behaviour, i.e. the assumption is that the system works without faults and there is no product loss. The initial state is defined as follows: all slots are empty and no operation is started.

For reasons of simplicity, we decide to concentrate on the control system. That means that we do not model material flow as this information can be obtained from the information stored by the main controller.

We assume that the main controller sends messages to the environment to allow adding and removing of products and the environment informs the main controller when the operations are completed. The environment can skip the adding or removing operations. A product can be removed from the removing position only if it has been drilled properly. If a product has a good test result and it has not been removed, it should not be drilled and tested again. If a product has a bad test result it must be drilled and tested again. That means that information on the adding and removing of products is only necessary after the rotation of the turntable.

When the other sensors change their states, the control system must be notified immediately. For instance, if the clamp sensor does not report that the clamp is locked, the drill cannot start drilling. Therefore, the turntable sensor states are checked by the control system just before a turn, while the other sensors inform the control system about their state changes immediately.

We also assume that the order of starting and ending of the adding, drilling, testing and removing operations is not known in advance.

The execution of each turntable operation requires a certain amount of time. Because the duration of the turntable operations has not been defined anywhere, we have decided to use the delays which have been defined in other turntable specifications, like the one of Bos and Kleijn (2002). We assume that the environment needs 2 time units to perform adding or removing of a product. The clamp needs 2 time units to lock or unlock a product. The drilling operation takes 3 time units and returning the drill to its up position takes 2 time units. Testing and returning the tester to its initial (up) position require 2 time units each.

Verification properties Traditionally, verification properties have been classified into safety and liveness properties. Safety is usually defined as a set of properties that the system may not violate, while liveness is defined as the set of properties that the system must satisfy (Holzmann, 2004). Safety, then, defines that something bad will never happen, and liveness defines that eventually something good will happen. It can be argued what kind of property the absence of deadlock is, but here it is considered a liveness property, due to the fact that a deadlock situation trivially satisfies all safety properties, but does not satisfy deadlock freedom (Godefroid and Wolper, 1991).

Given those assumptions we want to verify the following properties:

1. The system does not contain a deadlock, i.e. it cannot come to a state from which it cannot continue operating (liveness).

2. If drilling (testing, adding or removing) is started then it is also finished and the turntable does not rotate in the meantime (liveness and safety).

3. If the product has a bad test result then the product remains on the table and is drilled again (liveness).

4. If the product has a good test result then the remover will be called to remove the product (liveness).

5. No drilling (testing or removing) takes place if there is no product in the slot and no adding is performed if there is a product in the slot (safety).

6. Every added product is drilled in the next rotation (liveness).

7. Every product eventually leaves the table (liveness).

8. When a product is added it takes between 25 and 39 time units to get its test result (liveness).

Property 7 is a liveness property that requires a *fairness* principle, which makes this property the most complicated one.

First, a product can be removed only if it has a good test result. However, the remover can always decide not to remove and the tester can always generate bad test results. Theoretically, this can happen, because the choices whether the product will be removed and whether the test result of the product is good or bad are non-deterministic. In order to verify this property we must put some notion of fairness to the verification process, i.e. exclude unfair paths, in which a product yields a bad test result infinitely often.

Second, since there are at most four products on the table, it can happen that one of the products stays on the table while the other ones are drilled properly and removed. In order to verify that every product will eventually be removed we must identify them in some way. The most common solution is to give colors to the products, for instance, *red* and *white*, and change the adder such that it adds (non-deterministically) *zero or more* white products, then *one* red, and then again *zero or more* white ones. We want to make sure that if a red product is added then a red one will leave the table eventually. Another solution would be to assign unique identifiers to products or use some other way to distinguish them, but one should be aware of the fact that an introduction of unique identifiers often leads to a large increase of the resulting state space.

The fairness constraints can be expressed syntactically in linear temporal logic (LTL), but not in branching temporal logic (like CTL).[1] In μ-calculus, fairness properties can be expressed very efficiently, as shown by Mateescu and Sighireanu (2003).

The last property (so-called *bounded liveness*) also requires identification of the products. First, we calculate manually the time interval within which a test result of a product is known based on the assumptions. After that we check this interval automatically.

4.3 The Turntable Specification in χ_t

The turntable system architecture is depicted in Figure 4.6. The mechanical components are represented by means of the processes *Tester*, *DrillSwitch*, *DrillMove*, *Clamp* and *Table*. These components are controlled by switching commands: the command *cDrillSwitch* switches the drill on/off, *cDrillMove* instructs the drill to start or stop drilling, *cClampSwitch* instructs the clamp to lock or unlock the product, and *cTesterMove* instructs the tester to start or stop testing. The other signals used are *cTurn* (which commands the turntable to start turning) and *cAdd, cRemove* (which inform the environment that it can perform adding or removing operations respectively). As already mentioned, the sensors are implemented in several ways (more explanation is given in the descriptions of the corresponding processes).

[1] For more on temporal logics, see e.g. Clarke et al. (1999).

The control system specification consists of the main controller, the drill and clamp controller and the tester controller, which are modelled by means of the processes *MainControl*, *DrillControl* and *TesterControl* respectively. The processes *EnvAdd* and *EnvRemove* represent the environment.

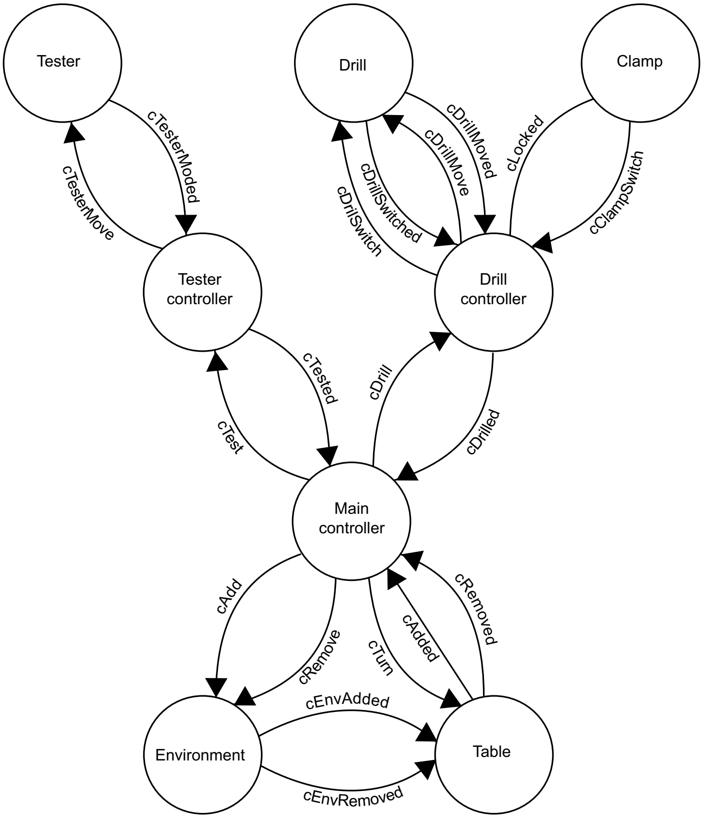

Figure 4.6: The turntable specification architecture

The architecture presented in Figure 4.6 is modelled with the χ_t specification called *Turntable*, which is as shown in Figure 4.7.

Next, we explain all processes appearing in *Turntable* in detail. Of each process, a description is given, followed by the χ_t code that models the component.

The *Table* process In the *Table* process (Figure 4.8) we work with variables *added* and *removed*, in order to keep track of adding and removing actions undertaken in

```
⟨ chan cAdd !?, cRemove !?, cAdded !?, cRemoved !? : bool ,
    cClampSwitch !?, cDrillSwitch !? : void , cLocked !?, cDrillSwitched !? : bool ,
    cDrillMove !?, cTurn !? : void , cDrillMoved !? : bool , cTurned !? : void ,
    cTesterMove !?, cTest !? : void , cTesterMoved !?, cTested !? : bool ,
    cDrill !?, cDrilled !?, cEnvAdded !?, cEnvRemoved !? : void
|   Table ∥ Clamp ∥ DrillSwitch ∥ DrillMove ∥ Tester ∥ MainControl
∥ DrillControl ∥ TesterControl ∥ EnvAdd ∥ EnvRemove
⟩
```

Figure 4.7: The χ_t turntable specification

the current position of the table. Once the *Table* process receives the message that a product has been added (or removed) it sets the *added* variable (or *removed* variable) to true. The value of these variables can be sent via channels *cAdded* and *cRemoved*. As can be seen later, the *MainControl* process regularly issues requests via these channels. Besides that, *MainControl* keeps track of all four slots and all their possible states (in total 5 per slot). When *Table* gets the signal *cTurn* it performs a delay to represent the turning of the table. Reading and updating are 'atomic' and instantaneous actions. The control system is modelled in such a way that it is not possible to perform those actions in parallel. This allows us to use alternative composition instead of parallel composition, resulting in a reduction of the state space.

```
Table = ⟦ disc added, removed : nat
          , added = false, removed = false
        | ∗( cEnvAdded ?; added = true
           [] cEnvRemoved ?; removed = true
           [] cAdded ! added
           [] cRemoved ! removed
           [] cTurn ?; Δ4.0; added := false; removed := false; cTurned !
           )
        ⟧
```

Figure 4.8: The χ_t process *Table*

The *Clamp* process The clamp process (Figure 4.9) has one actuator that is used to switch it on/off (*cClampSwitch*). The clamp also has a sensor to detect whether it is locked or unlocked. When the state of the sensor changes, the process *Clamp* reports to the drill control system via the channel *cLocked*.

```
Clamp = ⟦ disc clamp_on : bool
        , clamp_on = false
        | ∗( cClampSwitch ?
           ; ( clamp_on → clamp_on := false
             [] ¬clamp_on → clamp_on := true
             )
           ; Δ2.0; cLocked ! clamp_on
           )
        ⟧
```

Figure 4.9: The χ_t process *Clamp*

The *Drill* processes The drill is controlled by two independent actuators. One of the actuators is used to switch the drill on/off (*cDrillSwitch*). The other one (*cDrillMove*) instructs the drill to start drilling or to return in its initial (up) position. The states of the sensors are detected through the channels *cDrillSwitched* and *cDrillMoved*. The commands are handled independently, therefore we model these functionalities in two independent processes, *DrillSwitch* and *DrillMove* (Figures 4.10 and 4.11).

The *Tester* process The tester is controlled by one actuator, *cTesterMove*, which is used to start or stop testing. It has two sensors. One of them is used to detect a test result of a product, the other one detects whether the tester is in its initial (up) position. Possible test results are modelled by non-deterministic choice. When the test result of a product is good the process *Tester* (Figure 4.12) sends a signal via the channel *cTesterMoved*. Otherwise, it does not send any message. Of course, one could imagine modelling it differently, namely having the tester either send a success or a failure message, but the approach chosen here better reflects the real system.

The *MainControl* process The *MainControl* process (Figure 4.13) keeps track of the table slots and operates the other controllers. We use four local variables (*p0, p1, p2, p3*) to describe the state of every slot. The variable values range from 0 to 4

```
DrillSwitch = |[ disc drill_on : bool
                ,drill_on = false
                | *( cDrillSwitch ?
                   ; ( drill_on → drill_on := false
                     [] ¬drill_on → drill_on := true
                     )
                   ; Δ2.0; cDrillSwitched ! drill_on
                   )
                ]|
```

Figure 4.10: The χ_t process *DrillSwitch*

(0 means that there is no product in the slot, 1 - there is a product in the slot and it is not drilled, 2 - a product has been drilled, 3 - a product has been tested and has a bad test result, and 4 - a product has been tested and has a good test result). First, the *MainControl* process checks the states of the slots and starts corresponding processes (adding, drilling, testing and removing). The operations (*cAdd*, *cDrill*, *cTest* and *cRemove*) are started according to the following rules:

- The environment is allowed to add a product if there is no product in the slot.
- Drilling can be performed if there is a product in the slot and it has not been drilled yet or it has a bad test result.
- Testing is allowed if there is a product in the slot and it has been drilled.
- The environment is allowed to remove a product if there is a product in the slot and it has a good test result.

If these operations have been started the *MainControl* process waits until they are completed (*cAdded, cRemoved, cTested, cDrilled*). After that, it gives the command to the process *Table* to read the states of the sensors at the adding and removing slots and gets their current states. If their states have been changed (i.e. products have been added or removed), *MainControl* updates the information about current slot states. Then, it sends the command to turn the turntable (*cTurn*) to *Table* and waits until the turn is completed (*cTurned*). After this, the loop is repeated. In the real system the states of the sensors at adding and removing positions are automatically updated during the turn. To achieve this in the specification we send new states of the turntable sensors over the channel *cTurn*. In our specification, *MainControl* sends the value

```
DrillMove = ⟦ disc drill_down : bool
                ,drill_down = false
              | ∗( cDrillMove?
                 ; ( drill_down → drill_down := false; Δ2.0
                   [] ¬drill_down → drill_down := true; Δ3.0
                   )
                 ; cDrillMoved ! drill_down
                 )
              ⟧
```

Figure 4.11: The χ_t process *DrillMove*

of the sensors after the turn over the channel *cTurn* (the information is coded as an integer in following way: $p = 0$ means that there is no product in the adding and removing positions, $p = 1$ means that there is no product in the adding position and there is a product in the removing slot, $p = 2$ means that there is a product in the adding position and there is no product in the removing position, $p = 3$ means that there are products in both slots). Another approach to update the sensor states is to duplicate the information about all slots in the *Table* process (Bos and Kleijn, 2002). This approach allows one to separate the physical and control systems more easily and more simply but leads to a larger state space.

The *DrillControl* process The process *DrillControl* (Figure 4.14) gets the command to start drilling from the *MainControl* process over the channel *cDrill*. When this happens, it sends a signal to lock the clamp (via *cClampSwitch*) and waits for the reply from the clamp sensor (via *cLocked*). When the clamp is locked *DrillControl* uses the other switching command (via *cDrillSwitch*) to turn on the drill and waits for the confirmation (via *cDrillSwitched*). Next, it gives a signal to start drilling (via *cDrillMove*), waits for confirmation from the sensor (via *cDrillMoved*), sends a signal to return the drill to its initial (up) position (via *cDrillMove*), and waits for confirmation from the sensor (via *cDrillMoved*). After that, *DrillControl* switches the drill off (via *cDrillSwitch*), and waits for confirmation (via *cDrillSwitched*). Next, *DrillControl* unlocks the clamp (via *cClampSwitch*), waits for the signal from the clamp sensor (via *cLocked*) and reports to *MainControl* that drilling is completed (via *cDrilled*). Note that if during the procedure an unexpected reply is given by another process, *DrillControl* facilitates a livelock.

```
Tester =
    |[ disc tester_down : bool
     , tester_down = false
    | ∗( cTesterMove ?
       ; ( tester_down → tester_down := false; Δ2.0; cTesterMoved ! tester_down
         [] ¬tester_down → tester_down := true; Δ2.0; cTesterMoved ! tester_down
         [] ¬tester_down → tester_down := true; Δ2.0
         )
       )
    ]|
```

Figure 4.12: The χ_{t} process *Tester*

The *TesterControl* process *TesterControl* (Figure 4.15) gets a command to perform testing from *MainControl* (via *cTest*) and moves the tester down (via *cTesterMove*). To perform the testing operation, *Tester* needs 2 time units. If the tester has reached its down position within 2 time units, the test result of the product is good, but if the sensor does not react in 2 time units, the test result of the product is bad. The *TesterControl* process, however, waits for the signal from *Tester* for 3 time units instead of 2. The reason for this is that if *Tester* and *TesterControl* delay for the same amount of time, there is a possibility that *TesterControl* would make its choice before *Tester*. So, in order to ensure that *Tester* always makes its choice before *TesterControl* the latter delays longer. Now, if *Tester* makes a choice within 2 time units, *TesterControl* has no choice anymore. *TesterControl* stores the test result, moves the tester up (via *cTesterMove*), and sends the test result to *MainControl* over the channel *cTested*. Note that if during the procedure an unexpected reply is given by the tester, *TesterControl* facilitates a livelock.

The Environment processes There are two environment processes in the specification (Figures 4.16 and 4.17): one for adding and one for removing products. They get appropriate signals from *MainControl* to add or remove a product (via *cAdd* and *cRemove*). Once an environment process has decided to add or remove a product (it can also ignore a request), it sends the appropriate message to the *Table* process through the channels *cAdded* and *cRemoved*.

```
MainControl = ⟦ disc p1,p2,p3,p4,ps : nat , m : bool
                  ,p1 = 0,p2 = 0,p3 = 0,p4 = 0,ps = 0, m = false
                | ∗( ( p1 = 0 → cAdd ! true [] p1 ≠ 0 → skip )
                   ; ( p4 = 3 → cRemove ! true [] p4 ≠ 3 → skip )
                   ; ( p2 = 1 → cDrill [] p2 ≠ 1 → skip )
                   ; ( p3 = 2 → cTest; cTested ? m
                       ; ( ¬m → p3 := 1 [] m → p3 := 3 )
                     [] p3 ≠ 2 → skip )
                   ; ( p2 = 1 → cDrilled ?; p2 := 2 [] p2 ≠ 1 → skip )
                   ; cAdd ! false; cRemove ! false
                   ; cAdded ? m; ( m → p1 := 1 [] ¬m → skip )
                   ; cRemoved ? m; ( m → p4 := 0 [] ¬m → skip )
                   ; cTurn !; ps := p4; p4 := p3; p3 := p2; p2 := p1; p1 := ps
                   ; cTurned ?
                   )
                ⟧
```

Figure 4.13: The χ_{t} process *MainControl*

4.4 The Turntable Specification in μCRL

In the next few paragraphs we look at the μCRL specification of the turntable which resulted from translating the original χ_{t} specification. Instead of explicitly providing all LPE definitions though, we look at a representative number of them. For instance, the χ_{t} processes *EnvAdd* and *EnvRemove* are structurally so much alike that we omit the latter one. Translating the turntable specification was done using an implemented automatic translator using the scheme described in Section 3.4. However, for readability purposes, we limit the number of process counters in a process to one, such that this one counter n, which ranges over the natural numbers, takes over the functionality of the entire list of counters resulting from the translation scheme. Furthermore, we omit an alternative in a resulting LPE if its condition obviously never holds.

The definition of actions needed to translate the χ_{t} channels is not given here. Suffice it to say that each channel shown in Figure 4.6 is translated according to the scheme to an action triple, accompanied by an appropriate communication rule.

```
DrillControl = ⟦ disc b : bool
                 , b = false
                 | ∗( cDrill ?; cClampSwitch !; cLocked ? b
                    ; b → cDrillSwitch !; cDrillSwitched ? b
                    ; b → cDrillMove !; cDrillMoved ? b
                    ; b → cDrillMove !; cDrillMoved ? b
                    ; ¬b → cDrillSwitch !; cDrillSwitched ? b
                    ; ¬b → cClampSwitch !; cLocked ? b
                    ; ¬b → cDrilled !
                    )
                 ⟧
```

Figure 4.14: The χ_{t} process *DrillControl*

The *Table* process The first process we look at is the *Table* process. Figure 4.18 shows the μCRL specification of the corresponding LPE *TABLE*. In $\mathcal{I}$, the process *TABLE* is called with the process counter n equal to 0. Therefore the program can start execution only by performing one of the actions for which the accompanying guard includes the disjunct $n = 0$. The table initially has several alternative options:

- It can receive the message that the environment has received a new product or that a product has been removed. This will change the state in the obvious way.
- It can send the information whether a product has been added to position 1 and whether a product has been removed from position 4.
- It can receive the request to make a turn. Turning takes four time units, which is set in $\mathcal{I}$ by calling *TABLE* there with both t_0 and t_1 equal to 4.
- The lines beginning with τ are in fact translations of χ assignment actions.
- The *tock* alternative ensures the delayability of all send and receive actions.

Finally, note that the main loop in the process is achieved by resetting n to 0 at the appropriate moments, and that at those times t_1 is reset to its initial value t_0.

The *Clamp* process The χ_{t} process *Clamp* is translated to the LPE *CLAMP* displayed in Figure 4.19. Compared to the LPE *TABLE*, here we see the translation of

```
TesterControl = ⟦ disc t, test_result : bool
                   , t = false, test_result = false
                  | ∗( cTest?; cTesterMove!
                     ; ( cTesterMoved? t
                         ; t → test_result := true
                       [] Δ3.0; test_result := false
                       )
                     ; cTesterMove!; cTesterMoved? t
                     ; ¬t → cTested! test_result
                     )
                  ⟧
```

Figure 4.15: The χ_t process *TesterControl*

some χ_t guards. Their conditions are incorporated into the conditions of the corresponding μCRL actions. Note that although an individual guard construction can delay if its condition does not hold, the two guard constructions here together in alternative composition cannot delay, since they negate each other. This dealt with in the translation by the translation of alternative composition.

Due to having structures almost identical to the one of *CLAMP*, the LPEs *DRILLSWITCH*, *DRILLMOVE*, *ENVADD* and *ENVREMOVE*, which are the translations of the χ_t processes *DrillSwitch*, *DrillMove*, *EnvAdd* and *EnvRemove*, respectively, are skipped in our discussion.

The *Tester* process The χ_t *Tester* process represents the test device on the turntable. Figure 4.20 presents it in its translated form as the LPE *TESTER*. In the LPE *TESTER*, we see an example of several delays being present and how this is dealt with using pairs of timers.

The *DrillControl* process The χ_t *DrillControl* process is the process that controls the overall order of execution concerning the usage of the clamp and the drill. The LPE *DCONTROL* is the translation of this process, presented in Figure 4.21. The *DrillControl* process runs through a specific sequence of actions which is dealt with in the LPE by the usage of process counter n. Receive actions are here equipped with a boolean parameter, therefore summations over the boolean domain are added to allow the reception of both T and F. Finally, note that the delayability alternative is always

```
EnvAdd = |[ disc add_allowed : bool
           , add_allowed = true
           | *( cAdd ? add_allowed
              ; ( add_allowed → cEnvAdded !
                [] skip
                )
              )
           ]|
```

Figure 4.16: The χ_t process *EnvAdd*

```
EnvRemove = |[ disc remove_allowed : bool
              , remove_allowed = true
              | *( cRemove ? remove_allowed
                 ; ( remove_allowed → cEnvRemoved !
                   [] skip
                   )
                 )
              ]|
```

Figure 4.17: The χ_t process *EnvRemove*

enabled; this is due to the fact that no actions other than send and receive actions are present. It also facilitates livelocks in situations where a received boolean value is not as expected.

The *TesterControl* process The χ_t *TesterControl* process controls the testing procedure. The LPE *TCONTROL*, as shown in Figure 4.22, is the translation of this process. In this process we identify a time-out construction; when the action *rcTesterMoved* is enabled, a delay of t_1 time units is also enabled. However, once this delay is fully finished, a τ action can be fired, disabling the *rcTesterMoved* possibility. Depending on whether a time-out happens or not, variable *test_result* is set to F or T. Again, the *tock* alternative ensures livelocks whenever t has an unexpected value.

$$
\begin{aligned}
&\mathit{TABLE}(n : \mathbb{N}, \mathit{added} : \mathbb{B}, \mathit{removed} : \mathbb{B}, t_0 : \mathbb{T}, t_1 : \mathbb{T}) = \\
&\quad \mathit{rcEnvAdded}\cdot\mathit{TABLE}(1, \mathit{added}, \mathit{removed}, t_0, t_1) \lhd n = 0 \rhd \delta + \\
&\quad \tau\cdot\mathit{TABLE}(0, \mathrm{T}, \mathit{removed}, t_0, t_0) \lhd n = 1 \rhd \delta + \\
&\quad \mathit{rcEnvRemoved}\cdot\mathit{TABLE}(2, \mathit{added}, \mathit{removed}, t_0, t_1) \lhd n = 0 \rhd \delta + \\
&\quad \tau\cdot\mathit{TABLE}(0, \mathit{added}, \mathrm{T}, t_0, t_0) \lhd n = 2 \rhd \delta + \\
&\quad \mathit{scAdded}(\mathit{added})\cdot\mathit{TABLE}(0, \mathit{added}, \mathit{removed}, t_0, t_0) \lhd n = 0 \rhd \delta + \\
&\quad \mathit{scRemoved}(\mathit{removed})\cdot\mathit{TABLE}(0, \mathit{added}, \mathit{removed}, t_0, t_0) \lhd n = 0 \rhd \delta + \\
&\quad \mathit{rcTurn}\cdot\mathit{TABLE}(3, \mathit{added}, \mathit{removed}, t_0, t_1) \lhd n = 0 \rhd \delta + \\
&\quad \mathit{tick}\cdot\mathit{TABLE}(n, \mathit{added}, \mathit{removed}, t_0, t_1 - 1) \lhd n = 3 \land t_1 > 0 \rhd \delta + \\
&\quad \tau\cdot\mathit{TABLE}(4, \mathit{added}, \mathit{removed}, t_0, t_1) \lhd n = 3 \land t_1 = 0 \rhd \delta + \\
&\quad \tau\cdot\mathit{TABLE}(5, \mathrm{F}, \mathit{removed}, t_0, t_1) \lhd n = 4 \rhd \delta + \\
&\quad \tau\cdot\mathit{TABLE}(6, \mathit{added}, \mathrm{F}, t_0, t_1) \lhd n = 5 \rhd \delta + \\
&\quad \mathit{scTurned}\cdot\mathit{TABLE}(0, \mathit{added}, \mathit{removed}, t_0, t_0) \lhd n = 6 \rhd \delta + \\
&\quad \mathit{tock}\cdot\mathit{TABLE}(n, \mathit{added}, \mathit{removed}, t_0, t_1) \lhd n = 0 \lor n = 6 \rhd \delta
\end{aligned}
$$

Figure 4.18: The μCRL process *TABLE*

The *MainControl* process Finally the most important and complicated χ_t process is the *MainControl* process. It is translated to the LPE presented in Figure 4.23, called *MCONTROL*.

$$
\begin{aligned}
&\mathit{CLAMP}(n:\mathbb{N}, \mathit{clamp\_on}:\mathbb{B}, t_0:\mathbb{T}, t_1:\mathbb{T}) = \\
&\qquad \mathit{rcClampSwitch}\cdot \mathit{CLAMP}(1, \mathit{clamp\_on}, t_0, t_1) \triangleleft n = 0 \triangleright \delta \;+ \\
&\qquad \tau\cdot \mathit{CLAMP}(2, \mathrm{F}, t_0, t_1) \triangleleft n = 1 \wedge \mathit{clamp\_on} \triangleright \delta \;+ \\
&\qquad \tau\cdot \mathit{CLAMP}(2, \mathrm{T}, t_0, t_1) \triangleleft n = 1 \wedge \neg \mathit{clamp\_on} \triangleright \delta \;+ \\
&\qquad \mathit{tick}\cdot \mathit{CLAMP}(n, \mathit{clamp\_on}, t_0, t_1 - 1) \triangleleft n = 2 \wedge t_1 > 0 \triangleright \delta \;+ \\
&\qquad \tau\cdot \mathit{CLAMP}(3, \mathit{clamp\_on}, t_0, t_1) \triangleleft n = 2 \wedge t_1 = 0 \triangleright \delta \;+ \\
&\qquad \mathit{scLocked}(\mathit{clamp\_on})\cdot \mathit{CLAMP}(0, \mathit{clamp\_on}, t_0, t_0) \triangleleft n = 3 \triangleright \delta \;+ \\
&\qquad \mathit{tock}\cdot \mathit{CLAMP}(n, \mathit{clamp\_on}, t_0, t_1) \triangleleft n = 0 \vee n = 3 \triangleright \delta
\end{aligned}
$$

Figure 4.19: The μCRL process *CLAMP*

$$
\begin{aligned}
&\mathit{TESTER}(n:\mathbb{N}, \mathit{tester\_down}:\mathbb{B}, t_{00}:\mathbb{T}, t_{01}:\mathbb{T}, t_{10}:\mathbb{T}, t_{11}:\mathbb{T}, t_{20}:\mathbb{T}, t_{21}:\mathbb{T}) = \\
&\qquad \mathit{rcTesterMove}\cdot \mathit{TESTER}(1, \mathit{tester\_down}, t_{00}, t_{01}, t_{10}, t_{11}, t_{20}, t_{21}) \triangleleft n = 0 \triangleright \delta \;+ \\
&\qquad \tau\cdot \mathit{TESTER}(2, \mathrm{F}, t_{00}, t_{01}, t_{10}, t_{11}, t_{20}, t_{21}) \triangleleft n = 1 \wedge \mathit{tester\_down} \triangleright \delta \;+ \\
&\qquad \tau\cdot \mathit{TESTER}(3, \mathrm{T}, t_{00}, t_{01}, t_{10}, t_{11}, t_{20}, t_{21}) \triangleleft n = 1 \wedge \neg \mathit{tester\_down} \triangleright \delta \;+ \\
&\qquad \tau\cdot \mathit{TESTER}(4, \mathrm{T}, t_{00}, t_{01}, t_{10}, t_{11}, t_{20}, t_{21}) \triangleleft n = 1 \wedge \neg \mathit{tester\_down} \triangleright \delta \;+ \\
&\qquad \mathit{tick}\cdot \mathit{TESTER}(n, \mathit{tester\_down}, t_{00}, t_{01} - 1, t_{10}, t_{11}, t_{20}, t_{21}) \triangleleft n = 2 \wedge t_{01} > 0 \triangleright \delta \;+ \\
&\qquad \tau\cdot \mathit{TESTER}(5, \mathit{tester\_down}, t_{00}, t_{01}, t_{10}, t_{11}, t_{20}, t_{21}) \triangleleft n = 2 \wedge t_{01} = 0 \triangleright \delta \;+ \\
&\qquad \mathit{tick}\cdot \mathit{TESTER}(n, \mathit{tester\_down}, t_{00}, t_{01}, t_{10}, t_{11} - 1, t_{20}, t_{21}) \triangleleft n = 3 \wedge t_{11} > 0 \triangleright \delta \;+ \\
&\qquad \tau\cdot \mathit{TESTER}(6, \mathit{tester\_down}, t_{00}, t_{01}, t_{10}, t_{11}, t_{20}, t_{21}) \triangleleft n = 3 \wedge t_{11} = 0 \triangleright \delta \;+ \\
&\qquad \mathit{tick}\cdot \mathit{TESTER}(n, \mathit{tester\_down}, t_{00}, t_{01}, t_{10}, t_{11}, t_{20}, t_{21} - 1) \triangleleft n = 4 \wedge t_{21} > 0 \triangleright \delta \;+ \\
&\qquad \tau\cdot \mathit{TESTER}(0, \mathit{tester\_down}, t_{00}, t_{00}, t_{10}, t_{10}, t_{20}, t_{20}) \triangleleft n = 4 \wedge t_{21} = 0 \triangleright \delta \;+ \\
&\qquad \mathit{scTesterMoved}(\mathit{tester\_down})\cdot \mathit{TESTER}(0, \mathit{tester\_down}, t_{00}, t_{00}, t_{10}, t_{10}, t_{20}, t_{20}) \triangleleft n = 5 \triangleright \delta \;+ \\
&\qquad \mathit{scTesterMoved}(\mathit{tester\_down})\cdot \mathit{TESTER}(0, \mathit{tester\_down}, t_{00}, t_{00}, t_{10}, t_{10}, t_{20}, t_{20}) \triangleleft n = 6 \triangleright \delta \;+ \\
&\qquad \mathit{tock}\cdot \mathit{TESTER}(n, \mathit{tester\_down}, t_{00}, t_{01}, t_{10}, t_{11}, t_{20}, t_{21}) \triangleleft n = 0 \vee n = 5 \vee n = 6 \triangleright \delta
\end{aligned}
$$

Figure 4.20: The μCRL process *TESTER*

$$
\begin{aligned}
&\mathit{DCONTROL}(n:\mathbb{N}, b:\mathbb{B}) = \\
&\quad \mathit{rcDrill}\cdot\mathit{DCONTROL}(1,b) \triangleleft n = 0 \triangleright \delta + \\
&\quad \mathit{scClampSwitch}\cdot\mathit{DCONTROL}(2,b) \triangleleft n = 1 \triangleright \delta + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcLocked}(\hat{b})\cdot\mathit{DCONTROL}(3,\hat{b}) \triangleleft n = 2 \triangleright \delta + \\
&\quad \mathit{scDrillSwitch}\cdot\mathit{DCONTROL}(4,b) \triangleleft n = 3 \wedge b \triangleright \delta + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcDrillSwitched}(\hat{b})\cdot\mathit{DCONTROL}(5,\hat{b}) \triangleleft n = 4 \triangleright \delta + \\
&\quad \mathit{scDrillMove}\cdot\mathit{DCONTROL}(6,b) \triangleleft n = 5 \wedge b \triangleright \delta + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcDrillMoved}(\hat{b})\cdot\mathit{DCONTROL}(7,\hat{b}) \triangleleft n = 6 \triangleright \delta + \\
&\quad \mathit{scDrillMove}\cdot\mathit{DCONTROL}(8,b) \triangleleft n = 7 \wedge b \triangleright \delta + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcDrillMoved}(\hat{b})\cdot\mathit{DCONTROL}(9,\hat{b}) \triangleleft n = 8 \triangleright \delta + \\
&\quad \mathit{scDrillSwitch}\cdot\mathit{DCONTROL}(10,b) \triangleleft n = 9 \wedge \neg b \triangleright \delta + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcDrillSwitched}(\hat{b})\cdot\mathit{DCONTROL}(11,\hat{b}) \triangleleft n = 10 \triangleright \delta + \\
&\quad \mathit{scClampSwitch}\cdot\mathit{DCONTROL}(12,b) \triangleleft n = 11 \wedge \neg b \triangleright \delta + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcLocked}(\hat{b})\cdot\mathit{DCONTROL}(13,\hat{b}) \triangleleft n = 12 \triangleright \delta + \\
&\quad \mathit{scDrilled}\cdot\mathit{DCONTROL}(0,b) \triangleleft n = 13 \wedge \neg b \triangleright \delta + \\
&\quad \mathit{tock}\cdot\mathit{DCONTROL}(n,b)
\end{aligned}
$$

Figure 4.21: The μCRL process *DCONTROL*

$$
\begin{aligned}
&\mathit{TCONTROL}(n:\mathbb{N}, t:\mathbb{B}, \mathit{test\_result}:\mathbb{B}, t_0:\mathbb{T}, t_1:\mathbb{T}) = \\
&\quad \mathit{rcTest}\cdot\mathit{TCONTROL}(1, t, \mathit{test\_result}, t_0, t_1) \lhd n = 0 \rhd \delta + \\
&\quad \mathit{scTesterMove}\cdot\mathit{TCONTROL}(2, t, \mathit{test\_result}, t_0, t_1) \lhd n = 1 \rhd \delta + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcTesterMoved}(\hat{b})\cdot\mathit{TCONTROL}(3, \hat{b}, \mathit{test\_result}, t_0, t_1) \lhd n = 2 \rhd \delta + \\
&\quad \mathit{tick}\cdot\mathit{TCONTROL}(n, t, \mathit{test\_result}, t_0, t_1 - 1) \lhd n = 2 \wedge t_1 > 0 \rhd \delta + \\
&\quad \tau\cdot\mathit{TCONTROL}(4, t, \mathit{test\_result}, t_0, t_1) \lhd n = 2 \wedge t_1 = 0 \rhd \delta + \\
&\quad \tau\cdot\mathit{TCONTROL}(5, t, \mathrm{T}, t_0, t_1) \lhd n = 3 \wedge t \rhd \delta + \\
&\quad \tau\cdot\mathit{TCONTROL}(5, t, \mathrm{F}, t_0, t_1) \lhd n = 4 \rhd \delta + \\
&\quad \mathit{scTesterMove}\cdot\mathit{TCONTROL}(6, t, \mathit{test\_result}, t_0, t_1) \lhd n = 5 \rhd \delta + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcTesterMoved}(\hat{b})\cdot\mathit{TCONTROL}(7, \hat{b}, \mathit{test\_result}, t_0, t_1) \lhd n = 6 \rhd \delta + \\
&\quad \mathit{scTested}(\mathit{test\_result})\cdot\mathit{TCONTROL}(0, t, \mathit{test\_result}, t_0, t_0) \lhd n = 7 \wedge \neg t \rhd \delta + \\
&\quad \mathit{tock}\cdot\mathit{TCONTROL}(n, t, \mathit{test\_result}, t_0, t_1) \lhd n \leq 1 \vee (n = 3 \wedge \neg t) \vee n \geq 5 \rhd \delta
\end{aligned}
$$

Figure 4.22: The μCRL process *TCONTROL*

$$
\begin{aligned}
&\mathit{MCONTROL}(n:\mathbb{N},\mathit{p1}:\mathbb{N},\mathit{p2}:\mathbb{N},\mathit{p3}:\mathbb{N},\mathit{p4}:\mathbb{N},\mathit{ps}:\mathbb{N},m:\mathbb{B}) = \\
&\quad \mathit{scAdd}(\mathrm{T})\cdot\mathit{MCONTROL}(1,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=0 \wedge \mathit{p1}=0 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(1,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=0 \wedge \mathit{p1}\neq 0 \rhd \delta\ + \\
&\quad \mathit{scRemove}(\mathrm{T})\cdot\mathit{MCONTROL}(2,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=1 \wedge \mathit{p4}=3 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(2,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=1 \wedge \mathit{p4}\neq 3 \rhd \delta\ + \\
&\quad \mathit{scDrill}\cdot\mathit{MCONTROL}(3,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=2 \wedge \mathit{p2}=1 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(3,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=2 \wedge \mathit{p2}\neq 1 \rhd \delta\ + \\
&\quad \mathit{scTest}\cdot\mathit{MCONTROL}(4,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=3 \wedge \mathit{p3}=2 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(6,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=3 \wedge \mathit{p3}\neq 2 \rhd \delta\ + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcTested}(\hat{b})\cdot\mathit{MCONTROL}(5,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},\hat{b}) \lhd n=4 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(6,\mathit{p1},\mathit{p2},1,\mathit{p4},\mathit{ps},m) \lhd n=5 \wedge \neg m \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(6,\mathit{p1},\mathit{p2},3,\mathit{p4},\mathit{ps},m) \lhd n=5 \wedge m \rhd \delta\ + \\
&\quad \mathit{rcDrilled}\cdot\mathit{MCONTROL}(7,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=6 \wedge \mathit{p2}=1 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(8,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=6 \wedge \mathit{p2}\neq 1 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(8,\mathit{p1},2,\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=7 \rhd \delta\ + \\
&\quad \mathit{scAdd}(\mathrm{F})\cdot\mathit{MCONTROL}(9,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=8 \rhd \delta\ + \\
&\quad \mathit{scRemove}(\mathrm{F})\cdot\mathit{MCONTROL}(10,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=9 \rhd \delta\ + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcAdded}(\hat{b})\cdot\mathit{MCONTROL}(11,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},\hat{b}) \lhd n=10 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(12,1,\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=11 \wedge m \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(12,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=11 \wedge \neg m \rhd \delta\ + \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcRemoved}(\hat{b})\cdot\mathit{MCONTROL}(13,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},\hat{b}) \lhd n=12 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(14,\mathit{p1},\mathit{p2},\mathit{p3},0,\mathit{ps},m) \lhd n=13 \wedge m \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(14,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=13 \wedge \neg m \rhd \delta\ + \\
&\quad \mathit{scTurn}\cdot\mathit{MCONTROL}(15,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=14 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(16,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{p4},m) \lhd n=15 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(17,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p3},\mathit{ps},m) \lhd n=16 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(18,\mathit{p1},\mathit{p2},\mathit{p2},\mathit{p4},\mathit{ps},m) \lhd n=17 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(19,\mathit{p1},\mathit{p1},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=18 \rhd \delta\ + \\
&\quad \tau\cdot\mathit{MCONTROL}(20,\mathit{ps},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=19 \rhd \delta\ + \\
&\quad \mathit{rcTurned}\cdot\mathit{MCONTROL}(0,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \lhd n=20 \rhd \delta\ + \\
&\quad \mathit{tock}\cdot\mathit{MCONTROL}(n,\mathit{p1},\mathit{p2},\mathit{p3},\mathit{p4},\mathit{ps},m) \\
&\qquad\qquad \lhd (n=0 \wedge \mathit{p1}=0) \vee (n=1 \wedge \mathit{p4}=3) \vee (n=2 \wedge \mathit{p2}=1) \vee (n=3 \wedge \mathit{p3}=2) \\
&\qquad\qquad\ \ \vee n=4 \vee (n=6 \wedge \mathit{p2}=1) \vee 8 \le n \le 10 \vee n=12 \vee n=14 \vee n=20 \rhd \delta
\end{aligned}
$$

Figure 4.23: The μCRL process *MCONTROL*

The only part which might need additional explanation is the delayability alternative; since in the χ_{t} process, most of the communication actions have a skip action as an alternative, we observe that these guarded communication actions are not delayable if their guards do not hold (remember that in χ_{t} the strong time determinism principle holds and that skip is undelayable). Therefore, in the delayability alternative, the guards of these actions are explicitly stated here.

The initialisation Now that all the processes have been translated, all that remains is the construction of $\mathcal{I}$. It is as follows:

$$\begin{aligned}
&\partial_H(\mathit{TABLE}(0,\mathrm{F},\mathrm{F},4,4) \mathbin{|\overset{\top}{|}} \mathit{CLAMP}(0,\mathrm{F},2,2) \mathbin{|\overset{\top}{|}} \mathit{DRILLSWITCH}(0,\mathrm{F},2,2)\\
&\quad \mathbin{|\overset{\top}{|}} \mathit{DRILLMOVE}(0,\mathrm{F},2,2,3,3)) \mathbin{|\overset{\top}{|}} \mathit{TESTER}(0,\mathrm{F},2,2,2,2,2,2)\\
&\quad \mathbin{|\overset{\top}{|}} \mathit{DCONTROL}(0,\mathrm{F}) \mathbin{|\overset{\top}{|}} \mathit{TCONTROL}(0,\mathrm{F},\mathrm{F},3,3)\\
&\quad \mathbin{|\overset{\top}{|}} \mathit{MCONTROL}(0,0,0,0,0,0,\mathrm{F}) \mathbin{|\overset{\top}{|}} \mathit{ENVADD}(0,\mathrm{T})\\
&\quad \mathbin{|\overset{\top}{|}} \mathit{ENVREMOVE}(0,\mathrm{T}))
\end{aligned}$$

In $\mathcal{I}$, the set H contains all send and receive actions of the specification, and *tock* to block partial delays of the whole system.

4.5 Verifying the Properties

From the μCRL specification a state space can be generated using the μCRL toolset. This state space can then be analysed with the verification tool CADP. Using this tool, one can express properties in the regular alternation-free logic μ-calculus (see Section 2.3). These properties can be verified on the state space generated from the specification. The CADP tool, together with μ-calculus, was used for the verification of the properties of the turntable specification.

The state space Having translated the χ_{t} specification, the state space of the μCRL specification was generated using the μCRL toolset. This took about 20 seconds, resulting in a state space of 70,324 states and 119,306 transitions, which contained 72 deadlocks. This state space, however, still contained undesired system behaviour due to the lack of maximal progress. For instance, deadlocks resulted from the time-out construction in the *TCONTROL* process. Without maximal progress, the *Tester* process may be fully ignored by *TCONTROL*, even though it is ready to communicate. If it is ignored it cannot participate in later communications, resulting in deadlocks. Applying maximal progress using the state space reduction tool resulted in a state space of 24,496 states and 41,494 transitions. After reduction modulo branching bisimulation (Van Glabbeek and Weijland, 1996b) we ended up with a state space consisting of 8,919 states and 12,871 transitions. Table 4.1 shows the results mentioned. We also experienced that the order in which maximal progress and reduction modulo branching bisimilation are applied does not influence the final result.

Table 4.1: State spaces of the μCRL turntable specification

Type / **Size**	Initial	max. prog.	max. prog. + red.
# States	70,324	24,496	8,919
# Transitions	119,306	41,494	12,871

Verifying properties Using the state space obtained after applying maximal progress and reduction modulo branching bisimilation, we can start verifying system properties expressed in μ-calculus. Here we look again at the properties initially given in Section 4.2.

1. **The system does not contain a deadlock.** The absence of deadlock was verified in the CADP tool, which has this functionality built-in.

2. **If drilling (testing, adding or removing) is started then it is also finished. The turntable does not rotate in the meantime.** This property is checked in two steps. First we check the liveness property "if drilling is started then it is also finished" by using this formula:

   $$[\mathrm{T}^*.\texttt{ccDrill}]\ \mu X.(\langle \mathrm{T}\rangle\ \mathrm{T} \land [\neg\texttt{ccDrilled}]\ X)$$

   Then the safety property "it never occurs during drilling that the turntable rotates" is checked with this formula:

   $$[\mathrm{T}^*.\texttt{ccDrill}.\neg\texttt{ccDrilled}^*.\texttt{ccTurn}]\ \mathrm{F}$$

3. **If the product has a bad test result it remains on the table and is drilled again (when it comes to the drilling position).** This property is checked in two steps. The first formula is used to check, that it never occurs that a product with a bad test result is removed from the table after one turn of the table:

   $$[\mathrm{T}^*.\texttt{ccTested(F)}.\neg\texttt{ccTurn}^*.\texttt{ccTurn}.\neg\texttt{ccTurn}^*.\texttt{ccEnvRemoved}]\ \mathrm{F}$$

   It expresses that in no execution path of the state space a failed test of a product (`ccTested(F)`) is followed (after having done one rotation) by the removal of that product.

In the second formula it is checked that it never occurs that a product with a bad test result is not drilled again after three rotations:

$$[\mathrm{T}^*.\texttt{ccTested(F)}.\neg\texttt{ccTurn}^*.\texttt{ccTurn}.\neg\texttt{ccTurn}^*.\texttt{ccTurn}.\\ \neg\texttt{ccTurn}^*.\texttt{ccTurn}.\neg\texttt{ccDrill}^*.\texttt{ccTurn}]\,\mathrm{F}$$

4. **If the product has a good test result then the remover will be called to remove the product.** In other words, it will never happen that for a product with a good test result the remover is not called to remove the product. This is represented in μ-calculus by the formula:

   $$[\mathrm{T}^*.\texttt{ccTested(T)}.\neg\texttt{ccTurn}^*.\texttt{ccTurn}.\neg\texttt{ccRemove(T)}^*.\texttt{ccTurn}]\,\mathrm{F}$$

5. **No drilling (testing or removing) takes place if there is no product in the slot and no adding can be performed if there is a product in the slot.** Because of the usage of the regular alternation-free μ-calculus one can only verify action-based properties, but state-based properties can in general be checked in an action-based way by first changing the specification slightly, as shown by De Nicola and Vaandrager (1995). In essence, what happens is that if you want to check the value of parameter x of process X you extend the specification of X such that it can always perform a special action with x as a parameter which does not have a real effect on the system. More specifically, after executing this action the process returns to the state where it was when starting the action, i.e. the process performs a self-loop. Although this does not have an effect on the behaviour of the system, it does provide the ability to see the value of x at any given time in the state space, since all states are now equipped with a self-loop of this action with the value of x visible as a parameter.

   In this case, one of the subprocesses of the main control has been equipped with a self-loop executing the action *inslot2*, which tells whether a product is currently in slot 2 or not. Now we can state in μ-calculus:

   $$[\mathrm{T}^*.\texttt{inslot2(F)}.\texttt{ccDrill}]\,\mathrm{F}$$

   This expresses that you can never encounter an *inslot2*(F) action (there is no product in slot 2) just before *ccDrill*.

   In a similar way one can equip *MCONTROL* with a self-loop containing the action *inslot1*, which provides info on whether slot 1 contains a product or not. Then we can express in μ-calculus the property that no adding can be performed if there is a product in slot 1:

   $$[\mathrm{T}^*.\texttt{inslot1(T)}.\texttt{ccAdd(T)}]\,\mathrm{F}$$

6. **Every added product is drilled in the next rotation.** In other words, it never occurs that an added product is not drilled in the next rotation. In μ-calculus this leads to:

   $$[\mathrm{T}^*.\texttt{ccEnvAdded}.\neg\texttt{ccTurn}^*.\texttt{ccTurn}.\neg\texttt{ccDrill}^*.\texttt{ccTurn}]\ \mathrm{F}$$

7. **Every product eventually leaves the table.** This is a fairness property in the sense that the reachability of the action *cEnvRemoved* is checked in *fair* execution sequences. In μ-calculus the notion of "fair reachability of predicates" is used for fairness, as already stated in Chapter 4.5. To prove this property, the specification needs to be slightly changed so it supports coloured products; a product can be either red or white. By adding a boolean parameter for each position of the table to the parameter list of *MCONTROL*, this process can keep track of the colours of products, where T represents red and F represents white. In addition to that the *ENVADD* process is extended such that it ensures that it will constantly add white products except for at most one instance, in which it adds a red product. This red product can now easily be tracked through the turntable device. Using this changed specification, it is possible to prove the fairness property. The extension however, leads to an increase of the state space size, as Table 4.2 shows.

Table 4.2: State spaces of the μCRL turntable specification with coloured products

Type / **Size**	Initial	reduced	red. + max. prog.
# States	761,495	278,768	101,467
# Transitions	1,297,469	450,428	146,183

As an intermediate property, first we prove that in an execution sequence one can never encounter more than one red product:

$$[\mathrm{T}^*.\texttt{ccEnvAdded(T)}.\mathrm{T}^*.\texttt{ccEnvAdded(T)}]\ \mathrm{F}$$

Next, using this transformed specification, one can express the desired property as follows:

$$[\mathrm{T}^*.\texttt{ccEnvAdded(T)}.\neg\texttt{ccEnvRemoved(T)}^*]\ \langle \mathrm{T}^*.\texttt{ccEnvRemoved(T)}\rangle\ \mathrm{T}$$

This states that once a *ccEnvAdded*(T) action is encountered it is always possible to execute the action *ccEnvRemoved*(T) down the line.

8. **When a product is added it takes between 25 and 39 time units to get its test result.** This property is checked by using the specification with coloured products, since we need to track a single product to know how many time units it takes to get from having been added to having been tested. This specification is changed such that the actions *ccEnvAdded* and *ccTested* provide us the information (as an argument) what the colour is of the product that has been added or tested respectively. In μ-calculus we can then write a number of formulas.

   First of all, we check the lower-bound of 25 time units. We express in μ-calculus that all traces do not contain n *tick* transitions or less as follows, where the notation R^n is not actual μ-calculus notation, but expresses that R is written n times in sequence:

   $$[\mathrm{T}^*.\mathtt{ccEnvAdded(T)}.\neg\mathtt{tick}^*.((\mathrm{T} \mid \epsilon).\neg\mathtt{tick}^*)^n.\\ (\mathtt{ccTested(T,T)} \mid \mathtt{ccTested(F,T)})]\,\mathrm{F}$$

   The construction $((\mathrm{T} \mid \epsilon).\neg\mathtt{tick}^*)^n$ allows up to *n tick* actions to appear in a trace. This is because $(\mathrm{T} \mid \epsilon)$ accepts zero or one action, whatever its label.

   When $n = 25$, we get a counter-example containing exactly 25 *tick* transitions. Then, when we check the property with $n = 24$, we are told it holds. This tells us that 25 time units is indeed the lower-bound.

   Similarly, we investigate the upper-bound, using the following μ-calculus formula:

   $$\neg\langle \mathrm{T}^*.\mathtt{ccEnvAdded(T)}.(\neg(\mathtt{tick} \mid \mathtt{ccTested(T,T)} \mid \mathtt{ccTested(F,T)})^*.\mathtt{tick})^n.\\ \neg(\mathtt{ccTested(T,T)} \mid \mathtt{ccTested(F,T)})^*.(\mathtt{ccTested(T,T)} \mid \mathtt{ccTested(F,T)})\rangle\,\mathrm{T}$$

   Here, if we use $n = 39$, we get a counter-example containing exactly 39 *tick* transitions. If we check the property with $n = 40$, we are told it holds. Therefore, 39 time units is the upper-bound.

4.6 Results and Comparisons

In this section we present three tables to give an impression of the level of difficulty concerning different aspects on translating the χ_t specification and verifying its properties. Bortnik et al. (2005a) not only translated a (somewhat different) χ_t specification of the turntable to a μCRL specification, but also to a PROMELA specification (the input for the SPIN model checker) and a specification consisting of timed automata (the input for the UPPAAL model checker). We discuss their conclusions here, since it provides a good insight into how these three modelling techniques relate to each other. For more on the other translations, see the original paper by Bortnik et al. (2005a). The translation scheme from χ_{t} to PROMELA used there is described by Trčka (2006) and the translation scheme from χ_{t} to timed automata for UPPAAL is provided by Bortnik et al. (2005b). We do not present the sizes of the state spaces of the other two specifications here, due to the fact that no conclusions can be derived from such a comparison; the way in which e.g. the μCRL toolset and SPIN count states is entirely different. A comparison of state spaces generated by these two tools concerning another case study can be found in Section 10.3.

Tables 4.3 and 4.4 use a grading system which should be read as follows:

- **0: Impossible**. Due to differences between the two modelling languages or the limitations of the temporal logic it is impossible to do this.
- **1: Difficult**. For Table 4.3, translation is not straightforward but can be done using special techniques; for Table 4.4, verification cannot be done straightaway, it involves changing the specification a lot.
- **2: Needs some work**. For Table 4.3, translation is not completely straightforward, but it does not require special techniques; for Table 4.4, only slight changes in the specification are needed to verify this property.
- **3: Easy**. Translation or verification can be done straightaway.

Table 4.3 tells us how difficult it is to translate certain χ_{t} constructions in our case study (possibilities not mentioned here do not pose problems for any of the translations). Both translating to PROMELA and μCRL can be difficult under some circumstances, but translating to UPPAAL on the other hand never gets really difficult in our case. We added experience with translating shared variables and nested parallelism in this table, since at that time, we considered a χ_{t} specification which contained them. Shared variables were treated as described in Section 3.4.13, while nested parallelism, i.e. parallel composition inside a process, was translated to μCRL by introducing extra synchronisation actions. These results tell us that, at least concerning the turntable specification, UPPAAL is the best choice when selecting a language based on the difficulty to translate to this language.

Table 4.4 finally shows how difficult it is to express and verify the properties of the turntable using the tools. In this table, for each property (using the numbering presented earlier) the type (**s**afety, **l**iveness, **l**iveness + **s**afety, **l**iveness + **f**airness, **l**iveness + **t**ime) is given. What stands out is that property 7 is difficult or even impossible to verify in either tool. Property 8 is very hard to verify in SPIN due to the fact that time is hard to be referred to when expressing properties. In UPPAAL and CADP verifying this property still needs some work. This is not due to problems with time, but because in order to prove the property at least some products need to be uniquely identifiable. Overall we conclude that CADP provides the least number of problems concerning the verification of the properties.

Table 4.3: Comparison of translation problems

Problems Language	Assignments	Delays	Guards	Nested parallelism	Shared variables
Promela	3	2	1	1	3
μCRL	2	1	3	2	1
Uppaal timed aut.	3	3	3	2	3

Table 4.4: Comparison of the verification

Property Tool	1(l)	2(l+s)	3(l)	4(l)	5(s)	6(l)	7(l+f)	8(l+t)
Spin	3	2	2	2	3	2	1	1
Cadp (μCRL)	3	3	3	3	2	3	1	2
Uppaal	3	3	3	2	3	3	0	2

All three formalisms are suitable for the analysis of systems that are originally modelled in χ_{t}. When translating, we discovered that certain statements have straightforward translations in one language while they do not in another. For example, assignments exist in the same way in χ_{t}, Promela and Uppaal, while in μCRL they are represented differently. In addition, some χ_{t} constructs like, for example, the parallel composition operator with shared variables inside the process definition, are very hard to achieve in any of the target languages.

To reason about the values of variables (state values) in Cadp (using regular, alternation-free μ-calculus) one must extend the specification with additional actions that make these values visible. The linear temporal logic, built in Spin, and the timed computation tree logic, built in Uppaal, are more appropriate when reasoning about states than the regular, alternation-free μ-calculus used in Cadp, even though the latter is more powerful. Action-based properties could be verified in Spin as well, by the trace-assertion mechanism. In Uppaal proving such properties can be done using test automata or a decoration technique. When it comes to the fairness principle, μ-calculus and Spin can express it in a very effective way while in Uppaal the use of the fairness principle is impossible.

Finally, the graphical user interface makes modelling and verifying in Uppaal more comfortable than in μCRL. We also find XSpin, a graphical user interface for Spin, very useful.

4.7 Related Work

- Bos and Kleijn (2001) describe one of the first attempts to verify a specification written in an earlier version of χ by manual translation to DTPROMELA and applying the model checker DTSPIN is described. It dealt with a specification of an industrial system and had three objectives. The first objective was to investigate the ability to verify translated χ specifications with the model checker DTSPIN. The second objective was to find out whether there are opportunities to automatically translate χ specifications into DTPROMELA. Finally, the third objective was to verify formally some properties of a manufacturing system specification.

- In an article by Usenko (2002b), SPIN and μCRL are compared using the HAVi leader election protocol. Concerning the generation of a state space for a specification of this protocol, it was concluded that SPIN generates states faster, but the resulting state space has more states. On the other hand, according to the article, the state space generation capabilities of SPIN and the μCRL toolset cannot be compared due to the differences in the underlying languages. Furthermore, the results may be misleading, so the author tells us, due to the fact that the PROMELA code was derived from the μCRL code instead of from the informal description. The article ends by saying that a better comparison may be achieved using much smaller case studies.

- The article by Jensen et al. (1996) is on comparing SPIN and UPPAAL using a collision avoidance protocol as a case study. In the paper it is indicated that it is possible to model real-time systems and their broadcast behavior in UPPAAL and it cannot be done in SPIN. The kind of properties expressible in the UPPAAL requirement specification language are restricted to invariance and possibility properties. It is possible to verify bounded liveness properties in UPPAAL, though they need to be expressed as separate test automata.

- Garavel and Hermanns (2002) present a practical methodology for studying the performance of a concurrent system, starting from an already verified functional specification of this system. They do not design a new formalism, but instead reuse a non-stochastic process algebra called LOTOS, which they adapt to the stochastic framework. The CADP toolset is used for the minimisation of state spaces, resulting in Markov chains. These Markov chains can then be used as input for the TIPPTOOL, in order to find answers for performance questions.

- Mateescu (2006) provides two LOTOS specifications of the turntable system, one with a sequential and one with a parallel main controller. For both, the state space is generated and minimised. The two resulting state spaces are compared and visualisation techniques are applied on the smallest state space, the one with the sequential main controller, to get more insight into the system. Finally, a list of properties is checked using CADP.

- The article by Trčka (2006) discusses the general translation from χ_t to PROMELA in more detail.

- In Bortnik et al. (2005b), a general translation scheme from χ_t to timed automata for UPPAAL is described.

4.8 Conclusions

In the last two chapters, we presented a scheme to translate χ_t specifications to μCRL specifications, and applied it in practice. In this section, we comment on the lessons learned during the design and use of this scheme.

First of all, in creating the scheme, most complications arose due to the fact that χ_t is a timed language, and μCRL is not. For instance, when considering the alternative composition operator of χ_t, the basic (untimed) concept can be straightforwardly translated, since μCRL also has alternative composition. But when timing behaviour of an alternative composition in χ_t is analysed, one notices that it is hard to mimic this behaviour in a general way using *tick* actions in μCRL; whether a χ_t process term $p \,[]\, q$ can delay or not depends first of all on whether both p and q can delay or not. And whether p (and q) can delay or not depends again on the structure of p (and q).

Second of all, it should be stressed, that specifications resulting from automatic translation usually are not as intuitive as ones specified by hand. Since χ_t and μCRL are different languages, naturally the structures of specifications written in χ_t and μCRL differ slightly, even when describing the same system behaviour. Thus, when enforcing the structure of a χ_t specification on a μCRL specification, the latter becomes different from usual μCRL specifications. A small example of these differences in structure is the way by which values are assigned to variables; in χ_t, this can be done by means of the assignment action, while in μCRL there is no action associated with it. Instead, as variables are parameters of a recursion equation, they receive new values whenever the equation is recursively called. However, in order to mimic the semantics of χ_t, in a μCRL specification resulting from the translation of a χ_t specification, the τ action is always fired whenever an assignment takes place. Another example is the usage of (often many) counters in μCRL specifications resulting from translation. These are largely needed due to the inclusion of time actions; a sequential composition in a χ_t specification cannot be translated to a sequential composition in μCRL. This is because delayability of actions is achieved in μCRL using *tock* self-loops. If we consider a χ_t process term $a;\ b$ where both a and b are delayable, then a translation to $X = a{\cdot}b{\cdot}\surd + \mathit{tock}{\cdot}X$ does not suffice, since after firing a, the *tock* alternative is gone; more specifically, we have $X \xrightarrow{a} X'$, where $X' = b{\cdot}\surd$, hence b is not delayable here.

Looking at the experiments of the current chapter, our main conclusion concerning the verification of specifications via a translation step to another formalism, is that it can be very useful for checking action-based properties. Since both the original χ_{t} specification and the translation express the same behaviour, a state space resulting from the translation contains all possible traces of the χ_{t} specification. Hence, when a trace to a bad state is found, usually it is not necessary to consult the translation; instead, one can directly relate the trace to the original χ_{t} specification. Since the automatic translation is often hard to interpret, it is highly desirable for a modeller not having to read it.

We are aware of the fact that there is no correctness proof of the translation scheme included in the work. The main reason for this is that the research reported in Chapter 3 has first of all been done to get some insight into the relation between timed and untimed process algebras. Once we had designed the scheme, we tested it on a number of case studies. These experiments showed the applicability of the scheme, and its correctness in these specific cases. To prove the correctness of the scheme in general was outside of the scope of the work at the time; instead, the main lessons learned in the whole process led to a first design of a scheme to achieve discrete relative timing in the untimed μCRL, which would be more compatible with the structure of μCRL specifications, and therefore, would still allow the intuitive usage of μCRL's operators. This scheme is presented in Chapter 5. A correctness proof of the translation scheme may, of course, still be considered as possible future work. Because of the absence of a correctness proof here, we have also not provided a new structural operational semantics and a new set of axioms for μCRL with successful termination.

Chapter 5

Achieving Discrete Relative Timing with Untimed Process Algebra

El hoy fugaz es tenue y es eterno;
Otro Cielo no esperes, ni otro Infierno.

(*Jorge Luis Borges*)

MODEL CHECKING HAS PROVEN to be very useful in finding bugs in embedded system specifications. μCRL, for instance, has been used to verify properties of many systems and protocols. Many cases, however, are time-critical, meaning that time should also play a role in specifications of those systems, in order to be able to check relevant properties. Over the years, the inclusion of time in modelling languages has been shown to be complex, both on a theoretical and on a practical level. As can be found in the literature, in theory, subjects like the extension of modelling languages with time (Baeten, 2003; Baeten and Middelburg, 2002; Nicollin and Sifakis, 1991; Ulidowski and Yuen, 1997) and the design of relations between systems such as timed branching bisimilarity[1] (see, e.g., Fokkink et al. (2005) and Van der Zwaag (2002)) are very complicated, and at times difficult to get, and prove, correct. On the practical side, as mentioned earlier in Chapter 1, a major problem in model checking is the state space explosion problem, meaning that a linear growth of the number of processes placed in parallel in a specification leads to an exponential growth of the resulting state space, and adding time to a specification makes this problem even more difficult.

As already mentioned in Section 3.3, in the past, based on the modelling language μCRL, a timed language, called *timed* μCRL, has been developed by Groote (1997). For practical reasons, one of which is the aforementioned tendency of state spaces to be infinite, there are currently no tools for that language yet.

Also in Section 3.3, we saw that another approach can be taken. Blom et al. (2003)

[1] For more on this subject, see Part III of this thesis.

and Ioustinova (2004) investigated how to model time in regular, untimed μCRL, which would enable modellers to use the fully developed and highly optimised μCRL toolset when dealing with timed systems. Moreover, existing relations between systems, such as branching bisimilarity, can then be checked on timed systems. What they finally presented was a framework, a recipe, to express processes involving some notion of time. It is very important that a modeller follows this recipe faithfully, otherwise unwanted and bizarre timing behaviour might occur, such as the violation of principles like time determinism (Baeten and Middelburg, 2002) and maximal progress (Baeten and Middelburg, 2002).[2] We will take a closer look at these principles later on in this chapter. Besides that, a delay of one time unit corresponds directly with one transition in the system, resulting in large sequences of delays whenever a big time jump has to be made. One can imagine the inconvenience of this when confronted with a specification containing large delays.

In Chapters 3 and 4, we saw how this recipe is used as an inspiration for a translation scheme from the timed language χ_{t} to μCRL. By this, it is again shown that time can be modelled using an untimed modelling language, but in some cases, most notably when using alternative composition, it leads to complex process terms. The complexity of the resulting process is not so problematic, since it is the result of an automatic translation, but a high complexity cannot be demanded when a modeller has to create the term directly.

The work on modelling time by Blom et al. (2003) and Ioustinova (2004) and the translation scheme used in Chapters 3 and 4 inspired us to investigate the possibilities of modelling time in an untimed process algebra, such that:

1. The resulting timing mechanism makes sense, i.e. principles such as time determinism and maximal progress hold.

2. The modeller can use the process algebra as freely as when modelling untimed systems.

3. Arbitrarily big time jumps can be made in a single transition, provided that the system cannot fire any action during the time interval.

4. The existing tools can be applied on the timed systems, i.e. relations, such as branching bisimilarity, and properties, expressed using temporal logics, can be checked.

The creation of this chapter was a back and forth of designing a timing mechanism on the one hand and trying to model it in μCRL on the other. On a number of occasions the mechanism had to be changed somewhat due to the limitations of modelling, while still making sure that the mechanism remained reasonable from the perspective of the theory of timed systems.

[2] Maximal progress has already been described earlier in this thesis in Section 3.3.1.

In this chapter, we start with a discussion on the timing mechanism we wish to have, leading to, as we call it, the extended language μCRL^{tick}. The language μCRL^{tick} differs from, as we call it, 'μCRL with *tick* actions' (described in Section 3.3) in the following facts: First of all, when using μCRL with *tick* actions, a modeller needs to ensure that each system component in a specification must be in one of the two forms described in Figures 3.1 and 3.2, but this requirement is absent for μCRL^{tick}, which tends to make μCRL^{tick} specifications more readable than μCRL specifications with *tick*-actions (some examples of this are shown in Section 5.3). Second of all, μCRL^{tick} allows time transitions which jump ahead in time further than one time unit, while μCRL with *tick*-actions does not have this feature.

Next, we explain how the timing mechanism is achieved for a μCRL^{tick} specification through the transformation to a μCRL specification. The correctness proofs can be found in Appendix A. Finally, we show some examples, briefly discuss related work, and provide a conclusion, in which we also describe possible extensions.

Contributions We propose a way to achieve time through modelling in an untimed process algebra. Contrary to earlier attempts, we focus on ease of modelling and time jumps of arbitrary size, the latter since it often practically leads to smaller state spaces. We achieve this by the introduction of a timed extension of the algebra and a transformation to the original. Through doing so, this chapter offers more insight into the relation between untimed and timed process algebras.

5.1 A Timing Mechanism for μCRL

5.1.1 The Concepts

For timed process algebras, usually a number of time properties hold (see e.g. Baeten (2003), Baeten and Middelburg (2002), Van Beek et al. (2005), Nicollin and Sifakis (1991), Ulidowski and Yuen (1997)). Which hold and which do not differs from one process algebra to another. By only adding time actions to μCRL, we do not achieve all necessary properties. Because of this, we introduce an extension of μCRL, called μCRL^{tick}. The idea is that in the end, a μCRL^{tick} specification can be transformed into a μCRL specification.

Nicollin and Sifakis (1991) provide a list of main points in which timing mechanisms can differ from one another. We list these properties here and explain our choices. The choices depend both on whether the properties would be achievable through our approach or not, and whether it makes sense in relation to existing literature. By doing so, this section sketches the main setting of subsequent sections.

- **Time determinism.** The progress of time should be deterministic. This property is essential (see e.g. Nicollin and Sifakis (1991)). It has the largest influence on alternative composition; if two alternatives can delay, then they delay together. A further distinction can be made between *strong choice* and *weak choice*. In strong choice an undelayable alternative prevents all delays in the alternative composition, in weak choice it does not and the passage of time can therefore result in making a choice. We choose weak choice for our mechanism, since it more naturally fits in our approach (for an explanation and consideration of the alternative, see Section 5.6 or Baeten (2003)).

- **Time additivity.** If a process can delay $t + t'$ time units, then it can delay for t and then for t' time units. The behaviour of these two cases is the same. This property very often holds in a timing mechanism, but not in ours. Since we achieve time steps through regular action steps, as we will see later, two delays in sequence result in at least two steps, while one delay might be done in one step. It may seem a serious lack of our mechanism, but there are timed languages known to lack time additivity in certain situations, such as χ_{t}, in the case where one uses the Δt construction for a delay of t time units. As a positive note, not having time additivity allows us to use standard bisimilarity for the comparison of systems.

- **Deadlock-freeness.** Time can always pass, even though nothing else can be done. This practically allows for livelocks instead of deadlocks. This property holds in our mechanism. Besides that, in line with Baeten and Middelburg (2002) and Baeten (2003), we are able to introduce an undelayable deadlock (see Section 5.6).

- **Action urgency.** Often referred to as maximal progress, it allows actions to have priority over the passage of time. Already in Section 3.3.1, our version of it is explained, which is a mix of the ones described by Baeten and Middelburg (2002) and Van Beek et al. (2005).

- **Persistency.** The passage of time cannot suppress the ability to perform an action. As in many timed process algebras, also in our case, this property does not hold, due to weak time determinism.

- **Finite variability and bounded variability.** Also known as *non-Zenoness*, i.e. in every time unit only a finite number of actions can occur. Nicollin and Sifakis (1991) report that only in the process algebra TCSP these properties hold. In our mechanism they do not.

- **Bounded Control.** There exists a time period d, such that the enabled set of actions of a process only changes in time, if the delay is bigger than or equal to d. In a discrete time domain this automatically holds, therefore also in our setting (take $d = 1$).

Furthermore, we extend our usage of time in another way. In Chapter 3, time was modelled using *tick* and *tock* actions. The latter was used to make actions delayable. In this chapter, the time action *tock* is also used to perform *partial delays*, i.e. parts of specified delays, as a major extension introduced in this chapter is to allow time jumps of more than one time unit to be performed in a single transition. This is modelled by parameterising *tick* with a delay duration.

In the next section, a transformation procedure is presented, which transforms a μCRL^{tick} specification $\mathcal{M}_\mathsf{T}$ to a μCRL specification $\mathcal{M}$, in which practically the chosen set of time properties is achieved.

5.1.2 The Axioms and Transition Rules

We will present some axioms and transition rules, which we would like to hold in our timed setting in order to achieve the mechanism chosen in the previous section. In Appendix A, we prove that they indeed hold in our setting.

Table 5.1: Extra axioms of μCRL^{tick}

$tick(n) = \delta$ **if** $n < 0$	DRT1
$tick(n) \cdot x + tick(n) \cdot y = tick(n) \cdot (x + y)$	DRT2

In Tables 5.1 and 5.2, the additional axioms and transition rules are listed for the special clock actions *tick*, *tock*, and *ring*, which make discrete relative timing possible in our setting, the latter being an action which indicates that at least one delay is finishing.[3] Actually, in μCRL^{tick}, only *tick* is available as an action when modelling, the other two only appear in resulting transitions. The *tick* action is used to model delays (rules 1, 2, 5 to 7, 23, 24, 26), *tock* indicates the delayability of actions and partial delays (rules 1, 3, 4, 6 to 8, 23 to 26), and *ring* represents the finishing of at least one delay (rules 5, 9 to 22). Besides that, we identify sets $\mathcal{A}_U$, $\mathcal{A}_D$, and $\mathcal{A}_C$ in relation to the action set $\mathcal{A}_\mathsf{T}$ of a μCRL^{tick} specification, with $\mathcal{A}_\mathsf{T} \cup \{\tau, \delta\} = \mathcal{A}_U \cup \mathcal{A}_D \cup \{tick\}$, $tick \notin \mathcal{A}_U \cup \mathcal{A}_D$, $\mathcal{A}_U \cap \mathcal{A}_D = \emptyset$, $\tau \in \mathcal{A}_U$, $\delta \in \mathcal{A}_D$, and $\mathcal{A}_C = \{tick, tock, ring\}$. Now, $\mathcal{A}_U$ constitutes the *urgent* or *undelayable* actions, while $\mathcal{A}_D$ contains all *delayable* actions. An enabled urgent action is an action, which will be disabled once a delay is fired (rule 4). An enabled delayable action, on the other hand, is an action, which, in principle, can also be fired in a later time unit, i.e. can be postponed (rule 3). Note that we consider τ to be urgent, and δ to be delayable (which is essential for the deadlock-freeness property). The set $\mathcal{A}_C$ constitutes the set of clock actions, which forms the

[3] We chose here not to incorporate the successful termination constant ϵ. Alternative approaches where timed process algebras include this constant are presented by e.g. Baeten and Reniers (2000) and Baeten (2003).

Table 5.2: Extra transition rules of μCRL^{tick}

$$1\ \frac{m, n > 0}{tick(m+n) \xrightarrow{tock(n)} tick(m)} \qquad 2\ \frac{m > 0}{tick(m) \xrightarrow{tick(m)} tick(0)} \qquad 3\ \frac{m > 0 \quad a \in \mathcal{A}_D}{a \xrightarrow{tock(m)} a}$$

$$4\ \frac{m > 0 \quad a \in \mathcal{A}_U}{a \xrightarrow{tock(m)} \delta} \qquad 5\ \frac{}{tick(0) \xrightarrow{ring} \surd} \qquad 6\ \frac{x \xrightarrow{tick(m)} x' \quad y \xrightarrow{tick(m) \vee tock(m)} y'}{x + y \xrightarrow{tick(m)} x' + y'}$$

$$7\ \frac{x \xrightarrow{tick(m) \vee tock(m)} x' \quad y \xrightarrow{tick(m)} y'}{x + y \xrightarrow{tick(m)} x' + y'} \qquad 8\ \frac{x \xrightarrow{tock(m)} x' \quad y \xrightarrow{tock(m)} y'}{x + y \xrightarrow{tock(m)} x' + y'}$$

$$9\ \frac{x \xrightarrow{ring} x' \quad y \xrightarrow{ring} y'}{x + y \xrightarrow{ring} x' + y'} \qquad 10\ \frac{x \xrightarrow{ring} \surd \quad y \xrightarrow{ring} y'}{x + y \xrightarrow{ring} y'} \qquad 11\ \frac{x \xrightarrow{ring} x' \quad y \xrightarrow{ring} \surd}{x + y \xrightarrow{ring} x'}$$

$$12\ \frac{x \xrightarrow{ring} \surd \quad y \xrightarrow{ring} \surd}{x + y \xrightarrow{ring} \surd} \qquad 13\ \frac{x \xrightarrow{ring} x' \quad y \overset{ring}{\not\longrightarrow}}{x + y \xrightarrow{ring} x'} \qquad 14\ \frac{x \xrightarrow{ring} \surd \quad y \overset{ring}{\not\longrightarrow}}{x + y \xrightarrow{ring} \surd}$$

$$15\ \frac{x \overset{ring}{\not\longrightarrow} \quad y \xrightarrow{ring} y'}{x + y \xrightarrow{ring} y'} \qquad 16\ \frac{x \overset{ring}{\not\longrightarrow} \quad y \xrightarrow{ring} \surd}{x + y \xrightarrow{ring} \surd}$$

$$17\ \frac{x \xrightarrow{ring} x'}{x \parallel y \xrightarrow{ring} x' \parallel y} \qquad 18\ \frac{x \xrightarrow{ring} \surd}{x \parallel y \xrightarrow{ring} y} \qquad 19\ \frac{y \xrightarrow{ring} y'}{x \parallel y \xrightarrow{ring} x \parallel y'} \qquad 20\ \frac{y \xrightarrow{ring} \surd}{x \parallel y \xrightarrow{ring} x}$$

$$21\ \frac{x \xrightarrow{ring} x'}{x \cdot y \xrightarrow{ring} x' \cdot y} \qquad 22\ \frac{x \xrightarrow{ring} \surd}{x \cdot y \xrightarrow{ring} y}$$

$$23\ \frac{x \xrightarrow{tick(m)} x' \quad y \xrightarrow{tick(m) \vee tock(m)} y'}{x \parallel y \xrightarrow{tick(m)} x' \parallel y'} \qquad 24\ \frac{x \xrightarrow{tick(m) \vee tock(m)} x' \quad y \xrightarrow{tick(m)} y'}{x \parallel y \xrightarrow{tick(m)} x' \parallel y'}$$

$$25\ \frac{x \xrightarrow{tock(m)} x' \quad y \xrightarrow{tock(m)} y'}{x \parallel y \xrightarrow{tock(m)} x' \parallel y'} \qquad 26\ \frac{a \in \{tick, tock\} \quad x \xrightarrow{a(m)} x'}{x \cdot y \xrightarrow{a(m)} x' \cdot y}$$

action basis for achieving discrete relative timing in μCRL. In fact, all axioms and transition rules in Tables 5.1 and 5.2 involve actions from $\mathcal{A}_C$, while the axioms and transition rules inherited from μCRL (see Chapter 2) should be seen as applicable to all actions not in $\mathcal{A}_C$. Note that the *tick* action is considered to be an element of $\mathcal{A}_\mathsf{T}$, in other words, it can be used in μCRL^{tick} specifications, while *tock* and *ring* are not.

We composed the tables by examining the rules of BPA^{drt}-ID and ACP^{drt} (Baeten and Middelburg, 2002), which are the Basic Process Algebra extended with discrete relative timing by means of a delay operator, and the Algebra of Communicating Processes extended with discrete relative timing, respectively, and comparing them with the existing, untimed axioms and transition rules of μCRL, as described by Groote and Ponse (1995) (for these, also see Chapter 2). In Table 5.1, DRT1 says that a negative delay constitutes a deadlock. DRT2 is related to the previously mentioned time determinism principle, in that it says that the passage of time is deterministic. In Table 5.2, rules 1 and 2 reflect the fact that a delay can be performed (partially). Rules 3 and 4 say that delayable actions can delay and undelayable actions are disabled after a delay, respectively. Rule 5 indicates that a delay of length 0 terminates with a *ring* action. Rules 6, 7 and 8 ensure the time determinism principle. The additional transition rules for the *ring* action are stated in rules 9 to 22. Rules 23, 24 and 25 express time progress for parallel composition. Finally, in rule 26, the delayability of a sequential composition is explained.

These axioms and transition rules imply weak time determinism, no time additivity, no persistency, no finite or bounded variability and bounded control. As said before, that these axioms and transition rules indeed yield these properties for μCRL^{tick} is proven in Appendix A.

Concerning maximal progress, Algorithm 6 shows a breadth-first search with maximal progress. We observe that $ring \in H$, where H is the set of actions to apply maximal progress on, allows for nice constructions, such as a time-out (see Section 5.3).

5.2 Transforming a μCRL^{tick} Model

5.2.1 Requirements and approach

The basic idea of achieving discrete relative timing in this chapter, is that a modeller can create a μCRL^{tick} specification $\mathcal{M}_\mathsf{T}$, which will then be transformed into a μCRL specification $\mathcal{M}$, in which the presented set of timing properties automatically holds. The definition of a μCRL^{tick} specification fully coincides with the definition of a μCRL specification, i.e. Definition 1, extended with some additional conditions. Though the goal is to require as little as possible of $\mathcal{M}_\mathsf{T}$, some conditions are inevitable.

The Input Specification An input specification $\mathcal{M}_\mathsf{T} = (\mathcal{D}_\mathsf{T}, \mathcal{F}_\mathsf{T}, \mathcal{A}_\mathsf{T}, \mathcal{C}_\mathsf{T}, \mathcal{P}_\mathsf{T}, \mathcal{I}_\mathsf{T})$ is of the form described by Definition 1, where in addition, the following holds:

- Besides the usual domains $\mathbb{N}$ and $\mathbb{B}$, we have a time domain $\mathbb{T}$. Since we need a discrete and totally ordered time domain, and for practical reasons, negative values are needed, the structure of the domain can be a copy of $\mathbb{Z}$.
- Similarly, there are functions using $\mathbb{T}$, based on the usual functions using $\mathbb{Z}$, in $\mathcal{F}$. Later on, we assume the presence of an *if-then-else* construct, written as $b \rightarrow\ x, y$, with $b : \mathbb{B}$ and $x, y : \mathbb{T}$, where $\mathrm{T} \rightarrow x, y \triangleq x$ and $\mathrm{F} \rightarrow x, y \triangleq y$.
- $tick \in \mathcal{A}_{\mathrm{T}}$ with $tick : \mathbb{T}$, which the modeller can use to model delays of t time units. Furthermore, sets $\mathcal{A}_U$ and $\mathcal{A}_D$ can be identified.
- *tick* is not involved in any rule of $\mathcal{C}$.
- In $\mathcal{P}_{\mathrm{T}}$, for all $X \in \mathcal{P}_{\mathrm{T}}$, there are no enumerations over $\mathbb{T}$. Why this would cause problems will be explained later. Furthermore, on top of the axioms and transitions rules of μCRL for actions in $\mathcal{A}_{\mathrm{T}} \setminus \{tick\}$, the axioms and transition rules presented in Figures 5.1 and 5.2 hold for all actions in $\mathcal{A}_{\mathrm{T}}$.

The Transformation In this section, we describe how $\mathcal{M}_{\mathrm{T}}$, which meets the given requirements, can be transformed into a specification $\mathcal{M}$ in which the desired timing properties are preserved. The majority of the work, of course, is performed on $\mathcal{P}_{\mathrm{T}}$.

We create a specification $\mathcal{M} = (\mathcal{D}, \mathcal{F}, \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{I})$, by first obtaining the following, given specification $\mathcal{M}_{\mathrm{T}}$:

- $\mathcal{D} = \mathcal{D}_{\mathrm{T}}$.
- $\mathcal{F} = \mathcal{F}_{\mathrm{T}} \cup \{\theta : (\mathbb{B} \times \mathbb{N})^{\mathrm{u}} \rightarrow \mathbb{T}\}$. The function θ is given a vector $\overrightarrow{b_i, n_i}$ and returns the smallest n_i for which b_i evaluates to T and $n_i > 0$. The upper-bound u on the size of the vector should be chosen sufficiently high, as it will imply that not more than u syntactical occurrences of *tick* are allowed in each process. For practical reasons, we fix a sufficiently large constant c (it should be greater than the largest single delay step in $\mathcal{M}_{\mathrm{T}}$)[4] and say, that $\theta(\overrightarrow{b_i, n_i}) = \mathrm{c}$ iff for all (b_i, n_i), $b_i = \mathrm{F} \vee n_i \leq 0$ evaluates to T or the vector size is 0. In the remainder of this chapter, we write θ when $u = 0$, and m (for any $m \in \mathbb{T}$) whenever it is clear that $\theta(\overrightarrow{b_i, n_i}) = m$.
- $\mathcal{A} = \mathcal{A}_{\mathrm{T}} \cup \{ring, tock, tock', tick'\}$, where the actions $tock'$ and $tick'$ are used for intermediate results of communication. For more on this, see the following description of $\mathcal{C}$ and $\mathcal{I}$.

[4] The constant c serves here as an upper-bound to the size of a time jump that a system can perform. Note that appropriate values for both u and c, given a specification $\mathcal{M}_{\mathrm{T}}$, can be derived statically from $\mathcal{M}_{\mathrm{T}}$. The value of u can be derived from the number of occurrences of the *tick* action in processes in $\mathcal{P}_{\mathrm{T}}$, and the value of c must be greater than or equal to t, where t is the biggest element of $\mathbb{T}$ for which $tick(t)$ occurs in a process in $\mathcal{P}_{\mathrm{T}}$.

- $\mathcal{C} = \mathcal{C}_{\top} \cup \{(tick, tock, tick'), (tock, tick, tick'), (tick, tick, tick'), (tock, tock, tock')\}$.

Next, we describe how to obtain $\mathcal{I}$. This is done by changing $\mathcal{I}_{\top}$ to the following:

$$\partial_{H \cup \{tock\}}(TX_0(d_0) \mathbin{|\!\top\!|} \ldots \mathbin{|\!\top\!|} TX_m(d_m))$$

Here, the $TX_i \in \mathcal{P}$ are translations of the corresponding $X_i \in \mathcal{P}_{\top}$ (we return to this later), and the special operator $\mathbin{|\!\top\!|}$ is defined as already mentioned in Section 3.3:

$$P \mathbin{|\!\top\!|} Q \triangleq \rho_{\{tick' \to tick, tock' \to tock\}}(\partial_{\{tick,tock\}}(P \parallel Q))$$

First, in $P \parallel Q$, all *tick* and *tock* actions are forced to communicate. As can be seen in the rules of $\mathcal{C}$, if at least one *tick* action is involved in communication, the result is $tick'$. After that, all resulting $tick'$ and $tock'$ are renamed to *tick* and *tock*. Commutativity and associativity of $\mathbin{|\!\top\!|}$ can be argued in a similar fashion as commutativity and associativity of $|\{tick\}|$ (see Section 3.3). It should be stressed, that the final encapsulation of *tock* actions in $\mathcal{I}$ ensures that the system as a whole will only perform a delay if at least one process performs a complete delay, i.e. a *tick*-step.

Finally, we explain how to obtain $\mathcal{P}$. Each $X \in \mathcal{P}_{\top}$ is of the LPE form of Definition 4.

We divide I as follows: $I = I_U \cup I_D \cup I_C$, where $i \in I_U$ iff $a_i \in \mathcal{A}_U$, $i \in I_D$ iff $a_i \in \mathcal{A}_D$ and $i \in I_C$ iff $a_i = tick$.

Now, for each $X \in \mathcal{P}_{\top}$ in $\mathcal{I}_{\top}$, we create $TX \in \mathcal{P}$ as presented in Figure 5.1. The vector $\overrightarrow{x_{i_c}}$ consists of all x_i's for which $i \in I_C$. Similar definitions apply for $\overrightarrow{f_{i_c}(d, e_{i_c})}$ and $\overrightarrow{h_{i_c}(d, e_{i_c})}$. For clarity reasons, we write $\boldsymbol{f_i, g_i}$ and $\boldsymbol{h_i}$ for $f_i(d, e_i), g_i(d, e_i)$ and $h_i(d, e_i)$, respectively.

$$\begin{aligned}
&TX(d : \mathbb{D}) = \\
&\quad \sum_{i \in I \setminus I_C} \sum_{e_i : \mathbb{D}_i} a_i(\boldsymbol{f_i}) \cdot TX_i(\boldsymbol{g_i}) \lhd \boldsymbol{h_i} \rhd \delta + \\
&\quad ring \cdot T\hat{X}(d) \lhd \mathrm{F} \vee \bigvee_{i \in I_C} (\boldsymbol{f_i} = 0 \wedge \boldsymbol{h_i}) \rhd \delta + \\
&\quad tick(\theta(\overrightarrow{\boldsymbol{h_{i_c}}, \boldsymbol{f_{i_c}}})) \cdot TX'(d, \overrightarrow{\boldsymbol{h_{i_c}} \to \boldsymbol{f_{i_c}} - \theta(\overrightarrow{\boldsymbol{h_{i_c}}, \boldsymbol{f_{i_c}}}), \boldsymbol{f_{i_c}}}) \lhd \theta(\overrightarrow{\boldsymbol{h_{i_c}}, \boldsymbol{f_{i_c}}}) \neq \mathsf{c} \rhd \delta + \\
&\quad \sum_{t \in \mathbb{T}} tock(t) \cdot TX'(d, \overrightarrow{\boldsymbol{h_{i_c}} \to \boldsymbol{f_{i_c}} - t, \boldsymbol{f_{i_c}}}) \lhd 0 < t < \theta(\overrightarrow{\boldsymbol{h_{i_c}}, \boldsymbol{f_{i_c}}}) \rhd \delta
\end{aligned}$$

Figure 5.1: The μCRL LPE TX, derived from the μCRL^{tick} process X

At this point, an explanation is in order. In the first line of $TX(d : \mathbb{D})$, essentially all the 'lines' of $X(d : \mathbb{D})$ dealing with actions other than *tick*, are adopted without change.

The second line contains a *ring* action, which is fired whenever at least one delay completely finishes. Note that the guard checks whether there are any *tick* actions in X enabled with a parameter equal to 0. This effectively transforms possible occurrences of delays of length 0 in X to *ring*. This is similar to χ_{t}, where finished delay actions are followed by τ, except that here, *ring* can represent the finishing of multiple delays simultaneously. Having fired a *ring* action, the process $T\hat{X}$ is called, which will be described next. In the third line, a single *tick* alternative is added to the process, which has $\theta(\overrightarrow{\boldsymbol{h_{i_c}}, \boldsymbol{f_{i_c}}})$ as its argument, in other words, the minimal non-zero argument of all $tick \in en_{\mathcal{A}}(X(d_0))$, for any $d_0 \in \mathbb{D}$. After that, TX' is called, the definition of which will be presented later. Suffice it to say at this point that, besides $d : \mathbb{D}$, it is equipped with arguments of type $\mathbb{T}$, each getting the initial value $\boldsymbol{f_{i_c}} - \theta(\overrightarrow{\boldsymbol{h_{i_c}}, \boldsymbol{f_{i_c}}})$, if $\boldsymbol{h_{i_c}}$ holds, and $\boldsymbol{f_{i_c}}$, if not. The intuition is that these arguments will be used to model timers, used for each *tick* action appearing in X. If the *tick* action in TX is fired, all 'enabled' timers (i.e. timers corresponding to enabled *tick* actions) must be decreased by the right amount. Finally, the fourth line contains a number of *tock* alternatives, ranging from *tock*(1) to $tock(\theta(\overrightarrow{\boldsymbol{h_{i_c}}, \boldsymbol{f_{i_c}}}) - 1)$. These alternatives allow partial delays to happen, and actually make delayable actions delayable. When one of these is fired, the enabled timers are updated accordingly.

Process $T\hat{X}$ is the transformation of $\hat{X}$, which is as shown in Figure 5.2, where e.g. a^i_j indicates a_j originating from X_i.

$$\hat{X}(d : \mathbb{D}) = \sum_{i \in I_C} \sum_{j \in I^{X_i}} \sum_{e^i_j \in \mathbb{D}^i_j} a^i_j(f^i_j(\boldsymbol{g_i}, e^i_j)) \cdot X^i_j(g^i_j(\boldsymbol{g_i}, e^i_j)) \lhd h^i_j(\boldsymbol{g_i}, e^i_j) \land \boldsymbol{f_i} = 0 \land \boldsymbol{h_i} \rhd \delta$$

Figure 5.2: The μCRL^{tick} LPE $\hat{X}$, derived from the μCRL^{tick} process X

In words, $\hat{X}$ consists of the alternative composition of all actions from processes X_i with $i \in I_C$. Furthermore, the conditions ensure, that only actions following finished delays can be enabled. This process is transformed into $T\hat{X}$, in order to ensure the desired timing properties.

The process TX' is also derived from X. It is as displayed in Figure 5.3.

In the first line of TX', all delayable actions of X appear unchanged. In the second line, as in TX, a *ring* action is placed, to indicate the finishing of delays. Note that here, it is checked, whether the corresponding timers ct_i have expired. In the third line, as in TX, a *tick* alternative is offered, but also here, instead of working with the f_i's, the current values of the timers are used. Finally, in the fourth line, the *tock* alternatives are displayed, also working with the timers instead of the f_i's.

Finally, we have process $T\hat{X}'$, which is the transformation of $\hat{X}'$ (and similar to $T\hat{X}$), as presented in Figure 5.4.

$$
\begin{aligned}
&TX'(d : \mathbb{D}, \overrightarrow{ct_{i_c} : \mathbb{T}}) = \\
&\quad \sum_{i \in I_D} \sum_{e_i : \mathbb{D}_i} a_i(\boldsymbol{f_i}) \cdot TX_i(\boldsymbol{g_i}) \lhd \boldsymbol{h_i} \rhd \delta + \\
&\quad ring \cdot T\hat{X}'(d, \overrightarrow{ct_{i_c}}) \lhd \mathrm{F} \vee \bigvee_{i \in I_C} (ct_i = 0 \wedge \boldsymbol{h_i}) \rhd \delta + \\
&\quad tick(\theta(\overrightarrow{\boldsymbol{h_{i_c}}, ct_{i_c}})) \cdot TX'(d, \overrightarrow{\boldsymbol{h_{i_c}} \to ct_{i_c} - \theta(\overrightarrow{\boldsymbol{h_{i_c}}, ct_{i_c}}), ct_{i_c}}) \lhd \theta(\overrightarrow{\boldsymbol{h_{i_c}}, ct_{i_c}}) \neq \mathsf{c} \rhd \delta + \\
&\quad \sum_{t \in \mathbb{T}} tock(t) \cdot TX'(d, \overrightarrow{\boldsymbol{h_{i_c}} \to ct_{i_c} - t, ct_{i_c}}) \lhd 0 < t < \theta(\overrightarrow{\boldsymbol{h_{i_c}}, ct_{i_c}}) \rhd \delta
\end{aligned}
$$

Figure 5.3: The μCRL LPE TX', derived from the μCRL^{tick} process X

$$
\begin{aligned}
&\hat{X}'(d : \mathbb{D}, \overrightarrow{ct_{i_c} : \mathbb{T}}) = \\
&\quad \sum_{i \in I_C} \sum_{j \in I^{X_i}} \sum_{e^i_j : \mathbb{D}^i_j} a^i_j(f^i_j(\boldsymbol{g_i}, e^i_j)) \cdot X^i_j(g^i_j(\boldsymbol{g_i}, e^i_j)) \lhd h^i_j(\boldsymbol{g_i}, e^i_j) \wedge ct_i = 0 \wedge \boldsymbol{h_i} \rhd \delta
\end{aligned}
$$

Figure 5.4: The μCRL^{tick} LPE $\hat{X}'$, derived from the μCRL^{tick} process X

In this manner, every encountered process X leads to $TX, T\hat{X}, TX', T\hat{X}'$. All the resulting processes together form $\mathcal{P}$.

At this point, we return to the earlier mentioned restriction, that enumerations over $\mathbb{T}$ are not allowed in $\mathcal{M}_\mathsf{T}$. One might question the necessity for a μCRL^{tick} process such as $X = \sum_{t \in \mathbb{T}} tick(t)$, since partial delays are already achieved in a different way. Note that the transformation cannot deal with it properly, since the θ-function cannot be applied.

5.3 Examples

5.3.1 A Watchdog Timer

Ioustinova (2004) presents a watchdog timer as an example of using the timing mechanism of Blom et al. (2003) and Ioustinova (2004). As the construction of a watchdog timer is very common in timed systems, this is a nice example to show the applicability of the mechanism. It should watch whether an assigned component works properly, using two channels to communicate. On the first channel, an *ok* message must be received from the component every m time units. The moment a time-out occurs, in other words, *ok* is not received within *m* time units, an *alarm* message is sent over channel 2. Ioustinova (2004) presents the μCRL specification for this as in Figure 5.5, where *Timer* is the data type used for timers, the initialisation line is started with *init*, and the meaning of the functions and actions used should be clear from the context.

$$
\begin{aligned}
&A(t : \mathit{Timer}, m : \mathbb{N}) = \\
&\qquad \mathit{expire}{\cdot}B(\mathit{reset}(t), m) \lhd \mathit{expired}(t) \rhd \delta\, + \\
&\qquad \mathit{tick}{\cdot}A(t-1, m) \lhd \neg\mathit{expired}(t) \rhd \delta\, + \\
&\qquad \mathit{recv}(ok){\cdot}A(\mathit{set}(t, m), m) \\
&B(t : \mathit{Timer}, m : \mathbb{N}) = \mathit{send}(\mathit{alarm}){\cdot}A(\mathit{set}(t, m), m) \\
&\mathit{init}\ A(\mathit{on}(5), 5)
\end{aligned}
$$

Figure 5.5: A watchdog timer in μCRL

The same watchdog timer can be specified with $\mu\text{CRL}^{\mathit{tick}}$ in a much more readable way. For $a \in \mathcal{A}_U$, we write $\underline{\underline{a}}$.

$$A = \mathit{tick}(5){\cdot}\underline{\underline{\mathit{send}(\mathit{alarm}){\cdot}}}A + \mathit{recv}(ok){\cdot}A$$

This leads to the μCRL processes presented in Figure 5.6. For readability purposes, we removed alternatives with conditions which do not hold, and filled in results of the θ-function directly, where possible.

$$
\begin{aligned}
&TA = \mathit{recv}(ok){\cdot}TA + \\
&\qquad \mathit{tick}(5){\cdot}TA'(0) + \sum_{t \in \mathbb{T}} \mathit{tock}(t){\cdot}TA'(5-t) \lhd 0 < t < 5 \rhd \delta \\
&TA'(ct_0 : \mathbb{T}) = \mathit{recv}(ok){\cdot}TA + \\
&\qquad \mathit{ring}{\cdot}\hat{TA}' \lhd ct_0 = 0 \rhd \delta + \mathit{tick}(ct_0){\cdot}TA'(0) \lhd ct_0 \neq 0 \rhd \delta\, + \\
&\qquad \sum_{t \in \mathbb{T}} \mathit{tock}(t){\cdot}TA'(ct_0 - t) \lhd 0 < t < \theta(\mathrm{T}, ct_0) \rhd \delta \\
&\hat{TA}' = \mathit{send}(\mathit{alarm}){\cdot}TA + \sum_{t \in \mathbb{T}} \mathit{tock}(t){\cdot}\hat{TA}'' \lhd 0 < t < \mathrm{c} \rhd \delta \\
&\hat{TA}'' = \sum_{t \in \mathbb{T}} \mathit{tock}(t){\cdot}\hat{TA}'' \lhd 0 < t < \mathrm{c} \rhd \delta
\end{aligned}
$$

Figure 5.6: The $\mu\text{CRL}^{\mathit{tick}}$ process A transformed

Note that in order to make this work as desired, maximal progress needs to be applied for *ring* and for *recv* and *send*, or, when the watchdog timer is part of a bigger system, for the eventual communication action (e.g. say, that $(recv, send, com) \in \mathcal{C}_\top$). Then, *tick* alternatives to the actions are pruned away during state space generation. If, however, the component cannot send a message yet, due to delaying, the watchdog timer can wait by firing a *tick* or *tock* action, and subsequently keep track of the number of time units, until the time-out is reached, at which point the *ring* action becomes enabled, which must be fired before any further progress of time, due to maximal progress.

5.3.2 A Dish Washing Cluster

An old χ_t example is a dish washing cluster, as illustrated in Figure 5.7.

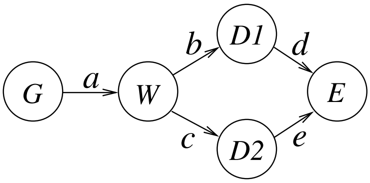

Figure 5.7: A dish washing cluster

A generator G supplies up to 14 plates, which are washed in sequence by W. Two driers $D1$ and $D2$ dry the plates concurrently, and finally, E removes the plates from the system. In Figure 5.8, we show how these processes are defined in μCRL^{tick}, where for each channel a, sa and ra represent sending and receiving over the channel, respectively, and $(sa, ra, ca) \in \mathcal{C}_\top$. It shows that the language is very suitable for specifying a system like this. The system leads to a state space of 940 states and 1,732 transitions without maximal progress, which can be generated in 0.9 seconds, and of 193 states and 193 transitions including maximal progress, which shows the practicality of our approach. The usage of the $\overset{\top}{\|}$ operator in $\mathcal{I}$ and the communication rules considering time actions in $\mathcal{C}$ ensure that the specified delays are always performed in as few steps as possible, i.e. time intervals in which no process can perform a normal action are jumped in a single transition.

5.3.3 The Turntable Revisited

We take another look at the turntable case of Chapter 4. We are very well able to model the turntable in μCRL^{tick} if we do not try to mimic the timing mechanism used in the original χ_t specification. The main difference in behaviour lies in the fact that χ_t

$$\begin{aligned}\mathcal{P}_{\mathsf{T}} = \{&G(n:\mathbb{N}) = sa\cdot G(n+1) \lhd n < 14 \rhd \delta,\\ &W = ra\cdot tick(15)\cdot(sb+sc)\cdot W,\\ &D1 = rb\cdot tick(25)\cdot sd\cdot D1,\\ &D2 = rc\cdot tick(25)\cdot se\cdot D2,\\ &E = (rd+re)\cdot E\}\end{aligned}$$

$$\mathcal{I}_{\mathsf{T}} = \partial_{\{sa,ra,sb,rb,sc,rc,sd,rd,se,re\}}(G(0) \parallel W \parallel D1 \parallel D2 \parallel E)$$

Figure 5.8: A μCRL^{tick} specification of a dish washing cluster

applies strong choice, while μCRL^{tick} applies weak choice. We display some μCRL^{tick} processes, modelling parts of the turntable model, in Figures 5.9, 5.10 and 5.11, merely to give an idea of modelling in μCRL^{tick}. We do not perform any model checking here. It should be emphasised that, when compared to the LPEs *TABLE*, *TESTER* and *DCONTROL* presented in Section 4.4, the μCRL^{tick} processes are more elegant.

$$\begin{aligned}&\mathit{TABLE}(\mathit{added}:\mathbb{B}, \mathit{removed}:\mathbb{B}) =\\ &\quad \mathit{rcEnvAdded}\cdot \mathit{TABLE}(\mathrm{T}, \mathit{removed}) +\\ &\quad \mathit{rcEnvRemoved}\cdot \mathit{TABLE}(\mathit{added}, \mathrm{T}) +\\ &\quad \mathit{scAdded}(\mathit{added})\cdot \mathit{TABLE}(\mathit{added}, \mathit{removed}) +\\ &\quad \mathit{scRemoved}(\mathit{removed})\cdot \mathit{TABLE}(\mathit{added}, \mathit{removed}) +\\ &\quad \mathit{rcTurn}\cdot \mathit{tick}(4)\cdot \mathit{scTurned}\cdot \mathit{TABLE}(\mathrm{F}, \mathrm{F})\end{aligned}$$

Figure 5.9: The μCRL^{tick} process *TABLE*

5.3.4 The PAR Protocol

Finally, let us specify and verify a well-known protocol. We choose to look at the so-called *Positive Acknowledgement with Retransmission* (PAR) protocol (see e.g. Tanenbaum (1989, Section 4.3.3)), since it is small but representative for timed systems, and it has been considered multiple times already in the literature, for instance by Baeten et al. (2002), Ioustinova (2004), and Reniers and Van Weerdenburg (2007), thereby facilitating comparisons.

$$
\begin{aligned}
&\mathit{TESTER}(\mathit{tester\_down} : \mathbb{B}) = \\
&\quad \mathit{rcTesterMove}\cdot \mathit{tick}(2)\cdot \mathit{scTesterMoved}(\mathrm{F})\cdot \mathit{TESTER}(\mathrm{F}) \lhd \mathit{tester\_down} \rhd \\
&\quad \mathit{tick}(2)\cdot(\mathit{scTesterMoved}(\mathrm{T})\cdot \mathit{TESTER}(\mathrm{T}) + \tau\cdot \mathit{TESTER}(\mathrm{T}))
\end{aligned}
$$

Figure 5.10: The μCRL^{tick} process *TESTER*

$$
\begin{aligned}
&\mathit{DCONTROL} = \\
&\quad \mathit{rcDrill}\cdot \mathit{scClampSwitch}\cdot \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcLocked}(\hat{b})\cdot(\mathit{scDrillSwitch} \lhd \hat{b} \rhd \delta)\cdot \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcDrillSwitched}(\hat{b})\cdot(\mathit{scDrillMove} \lhd \hat{b} \rhd \delta)\cdot \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcDrillMoved}(\hat{b})\cdot(\mathit{scDrillMove} \lhd \hat{b} \rhd \delta)\cdot \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcDrillMoved}(\hat{b})\cdot(\mathit{scDrillSwitch} \lhd \neg\hat{b} \rhd \delta)\cdot \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcDrillSwitched}(\hat{b})\cdot(\mathit{scClampSwitch} \lhd \neg\hat{b} \rhd \delta)\cdot \\
&\quad \sum_{\hat{b}:\mathbb{B}} \mathit{rcLocked}(\hat{b})\cdot(\mathit{scDrilled} \lhd \neg\hat{b} \rhd \delta)\cdot \mathit{DCONTROL}
\end{aligned}
$$

Figure 5.11: The μCRL^{tick} process *DCONTROL*

The diagram in Figure 5.12 shows a simple version of the PAR protocol, as studied by e.g. Baeten et al. (2002). A sender process S receives messages at input port 1, which it has to forward to process R through the unreliable channel K. Since K is unreliable, it has to be acknowledged by R that it indeed receives these messages, which it does by sending a signal over the (also unreliable) channel L. The main principle of the protocol is that S labels the messages with an alternating bit, in order to avoid duplicate output of a message at port 2 of process R. After sending a message, which takes t_S time units, S waits for another t'_S time units before it decides that the message was not received by R. If it receives an acknowledgement from R before it times out, it will wait for a new message from port 1 and change the value of the alternating bit, otherwise it will resend the message. Process R checks the bit labelling of incoming messages, which takes t_R time units; if this label is as expected, it outputs the message through port 2.

In any case, it produces and sends an acknowledgement signal via L, which takes t'_R time units. Finally, channels K and L take t_K and t_L time units to pass along messages, respectively.

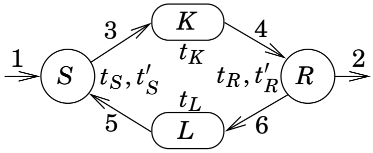

Figure 5.12: Diagram for the PAR protocol

We model the system as a $\mu\mathrm{CRL}^{tick}$ specification $\mathcal{M}_\top$, and consider messages to be natural numbers. In $\mathcal{M}_\top$, we define $\mathcal{D}_\top = \{\mathbb{N}, \mathbb{B}\}$, $\mathcal{F}_\top = \{=: \mathbb{N} \times \mathbb{N} \to \mathbb{B}, =: \mathbb{B} \times \mathbb{B} \to \mathbb{B}, \neg : \mathbb{B} \to \mathbb{B}\}$, $\mathcal{A}_\top = \{r1, s1, c1, r2, s2, c2, r3, s3, c3, r4, s4, c4, r5, s5, c5, r6, s6, c6, error, tick\}$ with r_i, s_i, c_i being actions for communication via port i, and *error* indicating the loss of a message on a channel. Furthermore, we have $(r_i, s_i, c_i) \in \mathcal{C}_\top$, for all $i \in \{1, 2, 3, 4, 5, 6\}$, and $\mathcal{P}_\top$ and $\mathcal{I}_\top$ as presented in Figure 5.13 with E being an environment process delivering messages to S. We do not enforce communication via port 2, thereby allowing the system to be open on that side.

$$
\begin{aligned}
\mathcal{P}_\top = \{&E = s1(1) \cdot E,\\
&S(b:\mathbb{B}) = \sum_{d:\mathbb{N}} r1(d) \cdot tick(t_S) \cdot SF(d,b),\\
&SF(d:\mathbb{N}, b:\mathbb{B}) = \underline{\underline{\underline{s3(d,b)}}} \cdot (tick(t'_S) \cdot SF(d,b) + r5 \cdot S(\neg b)),\\
&R(b:\mathbb{B}) = \sum_{d:\mathbb{N}} r4(d,b) \cdot tick(t_R) \cdot \underline{\underline{\underline{s2(d)}}} \cdot tick(t'_R) \cdot \underline{\underline{\underline{s6}}} \cdot R(\neg b)\\
&\qquad + \sum_{d:\mathbb{N}} r4(d,\neg b) \cdot tick(t'_R) \cdot \underline{\underline{\underline{s6}}} \cdot R(b),\\
&K = \sum_{d:\mathbb{N}} \sum_{b:\mathbb{B}} r3(d,b) \cdot tick(t_K) \cdot (\underline{\underline{\underline{s4(d,b)}}} + error) \cdot K,\\
&L = r6 \cdot tick(t_L) \cdot (\underline{\underline{\underline{s5}}} + error) \cdot L\}
\end{aligned}
$$

$$
\mathcal{I}_\top = \partial_{\{r1,s1,r3,s3,r4,s4,r5,s5,r6,s6\}}(E \parallel S(\mathrm{F}) \parallel K \parallel L \parallel R(\mathrm{F}))
$$

Figure 5.13: A $\mu\mathrm{CRL}^{tick}$ specification of the PAR protocol

Note the time-out mechanism in process SF; if an acknowledgement is received via port 5 within t'_S time units, the sender can start waiting for the next message to be sent. Otherwise, it needs to resend the current message. We refrain here from showing the μCRL specification resulting from the transformation of $\mathcal{M}_\mathrm{T}$. Suffice it to say, that we transformed $\mathcal{M}_\mathrm{T}$, and experimented with the delay constants t_S, t'_S, t_R, t'_R, t_K, and t_L. As remarked by Baeten et al. (2002), the protocol should only be correct if $t'_S > t_K + t_R + t'_R + t_L$ holds. If it does not hold, premature time-outs will occur in process SF. First we make sure that this condition holds; we set $t_S = t_R = t_K = t_L = 2$, $t'_R = 1$, and $t'_S = 8$. With this setting, the resulting state space before the enforcement of maximal progress has 158 states and 234 transitions. Having applied maximal progress, the final state space contains 64 states and 72 transitions. Using CADP, we observe that it does not contain any deadlocks, and that the protocol is correct here, i.e. two actions $c3(1,\mathrm{T})$ (or $c3(1,\mathrm{F})$) do not occur in a trace without an occurrence of either *error* or $c3(1,\mathrm{F})$ ($c3(1,\mathrm{T})$) between them. We specified this property in the following μ-calculus formula (likewise, it is possible to check for consecutive $c3(1,\mathrm{F})$ actions):

$$[\mathrm{T}^*.\mathtt{c3(1,T)}.(\neg(\mathtt{error} \mid \mathtt{c3(1,F)}))^*.\mathtt{c3(1,T)}]\,\mathrm{F}$$

Next, we set $t'_S = 4$. The resulting state space without maximal progress incorporates 3,132 states and 6,546 transitions, while the one with maximal progress has 686 states and 1,106 transitions. The state space with maximal progress contains no deadlock states, but the correctness property does not hold, in other words, at times SF prematurely concludes that it needs to resend a message.

5.4 Verification of Timing Properties

Now that we can model timed systems using $\mu\mathrm{CRL}^{tick}$, we would also like to express timing properties in order to check if they hold in these systems. For this, we use the approach as explained by Ioustinova (2004), but slightly modify it such that it works with parameterised *tick* actions. Ioustinova (2004) first extends the regular LTL temporal logic with time, after which a transformation is provided to LTL with *tick* actions. Here, we display the definition of regular LTL with time, after which we explain how the transformation is achieved and how ours differs from the one given by Ioustinova (2004).

The extension of regular LTL with time by Ioustinova (2004) is constructed by adding the possibility to express timing constraints. A timing constraint is of the form $\geq t$, $\leq t$, $= t$, where t is a non-negative integer constant. Such a timing constraint is referred to as tc.

Definition 9 (Syntax of regular LTL with time (Ioustinova, 2004)).

$$\phi ::= \mathrm{T} \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 U(r)_{tc}\phi_2$$

where tc is a timing constraint and r stands for a regular expression that does not mention the action tick*. We use* F*,* $\wedge$ *and* $\implies$ *as derived operators in the usual way, and define* $\langle r\rangle_{tc}\phi = \mathrm{T}U(r)_{tc}\phi$*,* $[r]_{tc}\phi = \neg\langle r\rangle_{tc}\neg\phi$*.*

Next, properties expressed in regular LTL with time should be transformed into regular LTL properties with *tick* actions. One approach for this uses the observation by Hopcroft et al. (2001) that regular expressions and deterministic finite automata have the same expressive power and can be translated into each other. Let A_r be the deterministic finite automaton obtained by the translation of a regular expression r. This automaton exactly recognises the set of strings for which r holds. Ioustinova (2004) provides a translation of timing constraints into deterministic finite automata. Having a deterministic finite automaton corresponding with a regular expression A_r and a deterministic finite automaton corresponding with a timing constraint, the product of these two recognises all interleavings of strings recognised by these two automata. We first provide the formal definition of deterministic finite automata and the set of strings they recognise. Finally, in Lemma 1, we redefine the translation of timing constraints such that parameterised *tick* actions are used.

Definition 10 (Deterministic finite automaton (DFA)). *A deterministic finite automaton (DFA) A is a quintuple* (S,Σ,T,s_0,F)*, where*

- S *is a set of states;*
- Σ *is a set of labels;*
- $T : S \times \Sigma \rightarrow S$ *is a transition function;*
- s_0 *is an initial state;*
- $F \subseteq S$ *is a set of final states.*

The set of strings recognised by A is given by $L(A) = \{\alpha_1 \ldots \alpha_n \mid \exists s_1,\ldots,s_n \in S \wedge s_n \in F \wedge \forall j = 0,\ldots,n-1.(s_j,\alpha_{j+1},s_{j+1}) \in T\}$*.*

Lemma 1. *For each timing constraint tc there is a DFA A_{tc} containing* tick *actions (with tick :* $\mathbb{T}$*) recognising it.*

Proof. The DFA recognising timing constraints can be constructed as follows:

$A_{\leq t} = (\{0,1,\ldots,t+1\},\{tick(1),\ldots,tick(\mathrm{c})\},\{(t+1,tick(t'),t+1),(i,tick(t'),min\{i+t',t+1\}) \mid i = 0,\ldots,t \wedge t' = 1,\ldots,\mathrm{c}\},\{0\},\{0,1,\ldots,t\})$.
$A_{=t} = (\{0,1,\ldots,t+1\},\{tick(1),\ldots,tick(\mathrm{c})\},\{(t+1,tick(t'),t+1),(i,tick(t'),min\{i+t',t+1\}) \mid i = 0,\ldots,t \wedge t' = 1,\ldots,\mathrm{c}\},\{0\},\{t\})$.
$A_{\geq t} = (\{0,1,\ldots,t\},\{tick(1),\ldots,tick(\mathrm{c})\},\{(t,tick(t'),t),(i,tick(t'),min\{i+t',t\}) \mid i = 0,\ldots,t-1 \wedge t' = 1,\ldots,\mathrm{c}\},\{0\},\{t\})$.

□

For Lemma 1, remember that c is our practical upperbound to the number of time units jumped in a single time transition. This means that no *tick*(t) can occur in a μCRL^{tick} state space with $t > \mathsf{c}$.

Besides the translation of timing constraints to DFA, the verification approach presented by Ioustinova (2004) can be reused for μCRL^{tick} state spaces. Therefore, we refer the reader to that work for more details.

5.5 Related Work

There are many papers on extending a specific process algebra with time. Here, we focus on papers on the basic timing concepts and comparisons of timing mechanisms. Nicollin and Sifakis (1991) discuss and compare a range of timing mechanisms according to a list of time properties (mentioned in this chapter in Section 5.1); additional axioms and transition rules are provided for a number of process algebras, and extra operators are introduced. Baeten and Middelburg (2002) extend both the Basic Process Algebra (BPA) (Bergstra and Klop, 1984a) and ACP (Bergstra and Klop, 1984b) with a range of different timing mechanisms. Some of the time properties are further elaborated on by Baeten (2003). A general framework for designing timed process languages is proposed by Ulidowski and Yuen (1997). The idea in these papers is to embed untimed into timed process algebra; in a sense one could say that we take an opposite approach, namely embedding timed into untimed process algebra, by means of a transformation procedure.

5.6 Conclusions and Extensions

We proposed an extension of μCRL, called μCRL^{tick}, which can practically be achieved, by transforming μCRL^{tick} models to μCRL models, such that the desired time properties still hold. We built on the work in Chapter 3 by emphasising ease of use and allowing time jumps of arbitrary length in a single transition. The toolset can directly be used for the verification of properties of μCRL^{tick} models, provided that it either includes an extra state space generation algorithm or a reduction tool providing maximal progress, as described in Section 3.3.1.

The setting allows for several extensions, of which we name a few here. For instance, Baeten (2003) and Baeten and Middelburg (2002) mention a current time unit *time-stop* $\underline{\underline{\delta}}$, which, in contrast to the standard deadlock, does not allow the passage of time. This could be added to μCRL^{tick} by adding an extra action in $\mathcal{A}_{\top}$ and in the transformation ensure that no delays can be performed whenever this action, by then translated to deadlock, is enabled. The communication mechanism can be left as is. Baeten and Middelburg (2002) also mention *relative time-out*, which places an upperlimit to the number of time units allowed to pass in the processes subjected to it. At least a system-wide version of it is achievable in our setting by including a process $R(c : \mathbb{T}) = \sum_{t \in \mathbb{T}} tock(t) \cdot R(c - t) \lhd c > 0 \wedge 0 < t \leq c \rhd \delta$ in $\mathcal{P}$ (after transformation), and placing R with a given upper-limit in parallel with the system.

Here, we chose a specific timing mechanism, most suitable for our transformation. There are, nevertheless, other mechanisms possible. Now we briefly go into how other decisions would affect the transformation. For instance, instead of weak choice, we could choose for strong choice. Then, it must be ensured in all processes that no delays are enabled whenever an urgent action is, which might result in very extensive conditions for the time actions. Another decision would be to interpret deadlock as a time-stop as opposed to a livelock, as mentioned by Baeten and Middelburg (2002) and Baeten (2003). Then the deadlock relates to the time-stop as described earlier.

Part II

Directed Quantitative Model Checking

In which an overview of searches is presented, a particular search is extended in several directions, a distributed setting is considered, and scheduling experiments are conducted

This part is based on the work of Aljazzar et al. (2007), Torabi Dashti and Wijs (2007), Wijs and Lisser (2007), and Wijs et al. (2005)

Chapter 6

State Space Searches for Directed Model Checking

Jedes Fragen ist ein Suchen. Jedes Suchen hat seine vorgängige Direktion aus dem Gesuchten her. Fragen ist erkennendes Suchen des Seienden in seinem Daß-und Sosein.

(*M. Heidegger*)

6.1 Introduction

STATE SPACE EXPLOSION IS STILL the main problem in the area of model checking. Model checking has proven to be very useful in lots of cases, but just as easily, one can find system models which yield enormous state spaces, that is, if they can in practice be generated at all. Abstracting away from unnecessary details can sometimes simplify the model and shrink the state space, but one cannot do this at leisure, as it should still be possible to verify the desired properties. Other possibilities are to use better tools, or faster computers with more memory, or to move to a distributed setting. But even then, there are very real limits as to what can be achieved.

Because of this, research is being done to efficiently explore state spaces to find deadlocks fast, particularly using Artificial Intelligence (AI) heuristic techniques, such as A^* (Edelkamp et al., 2004; Hart et al., 1968) and genetic algorithms (Godefroid and Khurshid, 2002). This approach is referred to as *directed* model checking (DMC) by Edelkamp et al. (2004). It has resulted in a whole scala of state space searches, where each search has its own unique way of searching a state space. Although most searches in this spectrum are used for so-called *qualitative* model checking, techniques like *beam search* (e.g. see Bisiani (1992)) can be applied for *quantitative* model checking as well, in particular to solve scheduling problems.

Model checking traditionally concerns modelling systems and checking properties, which either hold or not, in other words, the checks can be answered with either "yes" or "no". In more recent years, however, the awareness has grown that often other kinds of analyses, which cannot be answered in such a manner, are equally important. For these analyses, one is usually interested in some measurements, such as the throughput or efficiency of a particular system. Markov Chains, for instance, have shown to be useful when one needs to do performance analysis of a system (e.g. Bolch et al. (2006)). Although not common yet, sometimes scheduling problems are also addressed using model checking techniques, for instance by Behrmann et al. (2001a), Niebert et al. (2000), and Ruys (2003), since model checkers are usually equipped with highly expressive languages, making it possible to specify complex industrial scheduling questions. Comparing the two kinds of property checks, one could label traditional model checking as qualitative model checking, and the latter one as quantitative model checking (Huth and Kwiatkowska, 1997).

In this chapter, we give an overview of the spectrum of directed model checking search algorithms. In no way do we claim that our presentation covers the whole spectrum; we do, however, deal with the most frequently appearing techniques in this field, focussing in our presentation on the connections that can be drawn between the individual searches. As a help to the reader, the final section of this chapter contains a glossary of notions used in the upcoming sections.

In the following chapter, we limit ourselves to model checking techniques for scheduling. There, we take a very practical standpoint, describing, for a number of popular model checkers, how to model scheduling problems in general, and listing some of the available search techniques.

In Chapters 8 and 9, we limit our scope even further, by dealing with a specific search algorithm useful for scheduling, namely beam search. In Chapter 8, we extend two versions of this search to make them more effective for arbitrary state spaces. After that, in Chapter 9, we adapt the extensions to work in a distributed setting.

Finally, in Chapter 10, we discuss a number of experiments conducted, including an industrial case study, using a range of search algorithms.

6.2 Best-first Search

Let us again consider the breadth-first search algorithm presented in Chapter 2. We change it slightly to Algorithm 7, by explicitly checking for the reachability of goal states. Whenever such a state is encountered, we stop the search by invoking the *generation function GeneratePath*, which returns a trace leading from $\mathscr{I}$ to this goal state. If such a goal state does not exist, *false* is returned.[1]

[1] The decision to have Algorithm 7 and subsequent algorithms return *false* in case no goal state is encountered is made interpreting goal states as desirable states. Of course, when a goal state represents a bug

Algorithm 7 Breadth-first search for reachability analysis

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{I})$, goal states $\mathscr{G}$
Ensure: If exists, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathscr{L}_i \leftarrow \mathscr{I}$
  while $\mathscr{L}_i \neq \emptyset$ do
    $\mathscr{L}_{i+1} \leftarrow \emptyset$
    if $\mathscr{L}_i \cap \mathscr{G} \neq \emptyset$ then
      return $GeneratePath(\mathscr{L}_i \cap \mathscr{G})$
    end if
    for all $s \in \mathscr{L}_i$ do
      $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \cup \; nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathscr{L}_i \leftarrow \mathscr{L}_i \setminus \bigcup_{j=0}^{i-1} \mathscr{L}_j$
  end while
  return $false$
```

We can generalise this algorithm by introducing a *guiding function* f. The function f is used to determine which states to explore first. To keep things as general as possible, we leave the definition of f completely open in the current section. As Pearl (1984) notes: "[For best-first search, f,] in general, may depend on the description of [a state], the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain". In upcoming sections we look at searches using specific kinds of guiding functions.

A final selection of β states is done with the *select function* $select_\beta$: $2^{\mathscr{S}} \rightarrow 2^{\mathscr{S}}$, which employs a given *selection width* $\beta \in \mathbb{N} \cup \{\infty\}$; let S be a set of states, then $select_\beta(S) \subseteq S$ and $|select_\beta(S)| \leq \beta$. Algorithm 8 shows this generalisation, called *best-first search* (Russell and Norvig, 1995). In each round of the algorithm, f is used to select a set of states $\mathscr{L}_i \subseteq \mathscr{H}$ with minimal f-value, where we identify the set of states $\mathscr{H}$ as the *search horizon*. This set consists of all the states which have already been visited, but not yet explored. Concerning the usage of $select_\beta$, we distinguish two cases at this point: either $select_\beta$ selects exactly one state each time it is invoked, i.e. $\beta = 1$, or all the states, i.e. $\beta = \infty$. These two options lead to different versions of best-first search, i.e. an explicit-state version and a set-based version.

Since f is left unspecified for best-first search, many other searches, both complete and incomplete ones[2], can be seen as special instances of best-first search. In fact,

in a system, the proper answer when no goal state is found would be *true*.

[2] A search is called *complete* iff $\mathscr{G} \neq \emptyset$ and $\mathscr{G}$ is reachable implies that the search will find a trace to a goal state in $\mathscr{G}$.

Algorithm 8 Best-first search

```
Require: $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I})$, guiding function $f$, selection width $\beta$, goal states $\mathcal{G}$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathcal{H} \leftarrow \mathcal{I}$
  while $\mathcal{H} \neq \emptyset$ do
    $\mathcal{L}_i \leftarrow select_\beta(\{s \in \mathcal{H} \mid \forall s' \in \mathcal{H}.f(s) \leq f(s')\})$
    if $\mathcal{L}_i \cap \mathcal{G} \neq \emptyset$ then
      return $GeneratePath(\mathcal{L}_i \cap \mathcal{G})$
    end if
    for all $s \in \mathcal{L}_i$ do
      $\mathcal{H} \leftarrow \mathcal{H} \cup nxt_{\mathcal{M}}(s, en_{\mathcal{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathcal{H} \leftarrow \mathcal{H} \setminus \bigcup_{j=0}^{i-1} \mathcal{L}_j$
  end while
  return $false$
```

best-first search is not a concrete way of searching, but a name for a class of searches. So what are concrete possibilities for f then? To discuss this, we first move to a more general definition of a state space.

6.3 Weighted State Spaces and Cumulated Costs

Let us revisit the definition of a state space, as given in Chapter 2. We extend this to a *weighted state space* in Definition 11.

Definition 11 (Weighted state space). *A* weighted state space model *is a quintuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$, where $\mathcal{S}$ is a set of states, $\mathcal{I} \subseteq \mathcal{S}$ is a set of initial states, $\mathcal{A}$ is a finite set of action labels, $\mathcal{C} : \mathcal{A} \to \mathbb{K}$, with $\mathbb{K}$ a cost domain, is a total function assigning costs to action labels, and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation.*

Given a cost domain $\mathbb{K}$, in a weighted state space, every action in $\mathcal{A}$ is associated with a cost $c \in \mathbb{K}$. Such a, totally ordered, cost domain can for instance have the structure of the natural numbers or the integers, i.e. it can be discrete, or of the reals, in other words, it can be continuous. We do not consider negative cost values here. Note that a standard state space can be seen as a weighted state space where for all $\ell \in \mathcal{A}$, we have $\mathcal{C}(\ell) = 1$.

Weighted state spaces can typically be used to deal with priced problems, such as scheduling or planning problems (of which more in Chapter 7). These can be modelled as reachability problems, as shown by e.g. Behrmann et al. (2001a), where on top of the usual question whether a goal state $s \in \mathcal{G}$ can be reached or not, it is desired to find a trace to such a goal state with minimal cumulated cost. A first attempt at defining the *cumulated cost* of a state s, which matches the one found in most of the literature, is as follows:

Definition Attempt 1 (Cumulated cost). *Given a weighted state space $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$, we say that the* cumulated cost *of state $s \in \mathcal{S}$, denoted $g(s)$ (with $g : \mathcal{S} \to \mathbb{K}$), equals $g(s') + c$ iff $\exists \ell \in \mathcal{A}. s' \xrightarrow{\ell} s$ and $\mathcal{C}(\ell) = c$. For $s \in \mathcal{I}$, $g(s) = 0$.*

One problem with this definition becomes readily apparent, namely that for any $s \in \mathcal{S}$, $g(s)$ need not be unique, since many traces may lead from $\mathcal{I}$ to s. More on this later, but it raises the issue of *re-opening* of states in directed model checking. Typically, when generating a state space, and determining $g(s)$ for a state s on-the-fly, one may discover smaller $g(s)$ along the way. Because of this, there is a need to refer to the *minimal weighted distance* $\mathbb{d}$ from a set of states S to a set of states S'. This is defined in Definition 12.

Definition 12 ((Minimal) weighted distance between states). *Given a weighted state space $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$, a set of states $S \subseteq \mathcal{S}$ and a set of states $S' \subseteq \mathcal{S}$. Say that there exist $s \in S$, $s' \in S'$, and $s_0, \ldots, s_n \in \mathcal{S}$, with $n \geq 0$, such that $s \xrightarrow{\ell_0} s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_n} s_n \xrightarrow{\ell_{n+1}} s'$. Now we say that a* weighted distance *d from S to S' (along this trace), equals $c_0 + \ldots + c_{n+1}$, with $\mathcal{C}(\ell_i) = c_i$ for $0 \leq i \leq n+1$. Furthermore, this weighted distance d is called the* minimal weighted distance *between S and S', denoted $\mathbb{d}(S, S')$, iff there do not exist $\hat{s} \in S$, $\hat{s}' \in S'$, and $\hat{s}_0, \ldots, \hat{s}_m \in \mathcal{S}$, with $m \geq 0$, such that $\hat{s} \xrightarrow{\ell'_0} \hat{s}_0 \xrightarrow{\ell'_1} \hat{s}_1 \xrightarrow{\ell'_2} \ldots \xrightarrow{\ell'_m} \hat{s}_m \xrightarrow{\ell'_{m+1}} \hat{s}'$, with $\mathcal{C}(\ell'_i) = c'_i$ for $0 \leq i \leq m+1$ and $c'_0 + \ldots + c'_{m+1} < d$. In case S' is not reachable from S, meaning that there are no $s \in S$, $s' \in S'$ with $s \to^* s'$, then $\mathbb{d}(S, S') = \infty$.*

Clearly, if $S \cap S' \neq \emptyset$ then $\mathbb{d}(S, S') = 0$, since there is a state s for which $s \in S$ and $s \in S'$, which has a trace of length 0 to itself.

Now, we return to the notion of cumulated cost. As noted, $g(s)$, as defined in Definition Attempt 1, is in fact a relation, not per se a function, since $g(s)$ may have several values, in case there are multiple traces leading from $\mathcal{I}$ to s. We observe however that in on-the-fly searching, g is not merely a relation, but a function which is at times redefined, namely each time a state is (re-)opened; at any particular moment during a search, g is a (partial) function. Explicitly taking on-the-fly searching into account, we finally define the notion of cumulated cost in Definition 13. There, $s \to_T^* s'$ denotes that s' is reachable from s *through the set of transitions* T, i.e. there are $s_0, \ldots, s_n \in \mathcal{S}$ and $\ell_0, \ldots, \ell_{n+1} \in \mathcal{A}$, with $n \geq 0$, such that $s \xrightarrow{\ell_0} s_0 \in T$, $s_i \xrightarrow{\ell_{i+1}} s_{i+1} \in T$ for $0 \leq i \leq n-1$, and $s_n \xrightarrow{\ell_{n+1}} s' \in T$. Figure 6.1 shows an example of (re-)defining g on-the-fly. We return to this figure later on.

Definition 13 (Cumulated cost). *Given a weighted state space* $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, *we recursively define the* cumulated cost *of state* $s \in \mathscr{S}$ *relative to a given set of transitions* T *(called the* scope*), denoted* $g_T(s)$ *(with* $g_T : \mathscr{S} \rightarrow \mathbb{K}$*), as follows:*

- $g_{\emptyset}(s) = 0$ **if** $s \in \mathscr{I}$;
- *Given a (partial) function* g_T *and a transition* $s_0 \xrightarrow{\ell} s_1$ *such that* $s_0 \in \mathscr{I} \cup nxt_{\mathscr{M}}(T)$. *Then:*
  1. $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s) = g_T(s_0) + \mathscr{C}(\ell)$, **if** $s = s_1 \wedge \neg\exists s' \in \mathscr{I}.s' \rightarrow^*_T s$;
  2. $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s) = min\{g_T(s), g_T(s_0) + \mathscr{C}(\ell)\}$, **if** $s = s_1 \wedge \exists s' \in \mathscr{I}.s' \rightarrow^*_T s$;
  3. $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s) = g_T(s)$, **if** $\exists s' \in \mathscr{I}.s' \rightarrow^*_T s \wedge s \neq s_1 \wedge (s_1 \not\rightarrow^*_T s \vee (\exists s' \in \mathscr{I}.s' \rightarrow^*_T s_1 \wedge g_T(s_1) \leq g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s_1)) \vee s = s_0)$;
  4. $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s) = min(\{g_T(s)\} \cup \{g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(\hat{s}) + c \mid \exists \ell' \in \mathscr{A}.\hat{s} \xrightarrow{\ell'} s \in T \wedge \mathscr{C}(\ell') = c \wedge s_1 \rightarrow^*_T \hat{s}\})$, **if** $\exists s' \in \mathscr{I}.s' \rightarrow^*_T s \wedge s \neq s_1 \wedge s_1 \rightarrow^*_T s \wedge (\exists s' \in \mathscr{I}.s' \rightarrow^*_T s_1 \wedge g_T(s_1) > g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s_1)) \wedge s \neq s_0$.

In an on-the-fly search, initially, we only have the set of initial states $\mathscr{I}$ and no transitions, therefore we can only know the cumulated cost values of all $s \in \mathscr{I}$. The g-values of all other states, in $\mathscr{S} \setminus \mathscr{I}$, are undefined. We make this explicit by referring to the partial function $g_{\emptyset}$. As we search in the state space, new transitions are explored, leading to new states, i.e. our scope increases in size. Say that we are in round i of the search, and that our current scope T is increased to $T \cup \{s_0 \xrightarrow{\ell} s_1\}$. Then at the end of the round, we obtain a new cumulated cost function $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}$. Definition 13 distinguishes four cases:

1. A state s is the destination s_1 and it was not previously reachable through the scope T, i.e. it is a newly discovered state. Then $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s) = g_T(s_0) + \mathscr{C}(\ell)$.
2. A state s is the destination s_1 and it was previously reachable through the scope T, i.e. we revisit a state and have to consider updating its cumulated cost. Then $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s) = min\{g_T(s), g_T(s_0) + \mathscr{C}(\ell)\}$.
3. A state s is not the destination s_1. In this case, there are two possibilities:
   - s is not reachable from s_1 through the scope T. Clearly, then $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s) = g_T(s)$.
   - The destination s_1 is reachable through the scope T, i.e. $g_T(s_1)$ is defined, and its cumulated cost value has not been updated. Then, $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s) = g_T(s)$.

- s is the source s_0. Then $g_{T \cup \{s_0 \xrightarrow{\ell} s_1\}}(s) = g_T(s)$. Even if s_0 and s_1 are in a (shared) loop, i.e. $s_1 \rightarrow_T^* s_0$, since we consider non-negative costs, a cumulated cost update for s_1 due to $s_0 \xrightarrow{\ell} s_1$ cannot also lead to a cumulated cost update for s_0.

4. A state s is neither the source s_0, nor the destination s_1, but it is reachable from s_1 through the scope T. Furthermore, the cumulated cost value of s_1 has been updated and $g_T(s_1)$ is defined. Then the update of the cumulated cost of s_1 may affect the cumulated cost of s. In this case, we need to take the minimum of the original cumulated cost of s and the updated cumulated cost of each predecessor $\hat{s}$ reachable from s_1 plus the cost of the transition from $\hat{s}$ to s.

It follows that for all $T \subset \mathscr{T}$, $g_T : \mathscr{S} \rightarrow \mathscr{S}$ is a partial function. In fact, the only total function is $g_{\mathscr{T}}$. As we continue searching a state space, and our scope increases, we 'discover' the definition of function đ (note that $g_{\mathscr{T}} = \text{đ}$). In most of this thesis, we omit the scope of the cumulated cost function, as is custom in the literature. Definition 13, therefore, is mainly here to highlight the practice of discovering the final function g (i.e. the total function $g_{\mathscr{T}}$), and, with that, đ, on-the-fly.

Next, in Definition 14, we define *monotonicity of cumulated cost functions*.

Definition 14 (Monotonicity of cumulated cost functions). *A cumulated cost function $g_T : \mathscr{S} \rightarrow \mathbb{K}$ with scope $T \subseteq \mathscr{T}$ is called* monotonic *iff for all $s \in S \setminus \mathscr{I}$ there exist $s' \in \mathscr{S}$, $\ell \in \mathscr{A}$ with $s' \xrightarrow{\ell} s \in T$ and $g(s) \geq g(s')$.*

In words, for every state s in the scope, there exists a predecessor with a cumulated cost smaller than or equal to $g_T(s)$. We cannot claim that all predecessors of s have a smaller cumulated cost, because $g(s)$ might very well have been updated to a smaller value at least once during the search, while the g-values of some of its predecessors are not. An example of this is illustrated in Figure 6.1. At first, on the left of the figure, the cumulated cost of state s_3 is greater than the cumulated cost of state s_1. However, the transition from state s_2 to s_3 has not been explored yet. When increasing the scope to $T \cup \{s_2 \longrightarrow s_3\}$, we revisit s_3 by expanding s_2, and find that its cumulated cost should be updated. As the cumulated cost of s_1 is not updated, since it is neither the source s_0, nor the destination s_1, nor is it reachable from s_3, $g_{T \cup \{s_2 \longrightarrow s_3\}}(s_1) > g_{T \cup \{s_2 \longrightarrow s_3\}}(s_3)$.

When considering $\mathbb{K}$ without negative elements, monotonicity of g follows trivially from Definition 13.

Now we can define best-first search for weighted state spaces, where, in order to optimise the computational complexity, we store $g(s)$ with each state $s \in \mathscr{S}$. We refer to such a stored g-value as $s.g$. By doing so, we can compute $g(s)$ for any $s \in \mathscr{S}$ by accessing the stored g-value of (one of) its predecessor(s), and adding the cost of the fired action to it. For this purpose, we redefine $nxt_{\mathscr{M}}(s, T)$ as $nxt_{\mathscr{M}}(s, T) = \{\langle s', s.g + c \rangle \mid s' \in \mathscr{S} \wedge \exists \ell \in \mathscr{A}.\ (s \xrightarrow{\ell} s' \in T \wedge \mathscr{C}(\ell) = c)\}$. The result is presented in Algorithm 9.

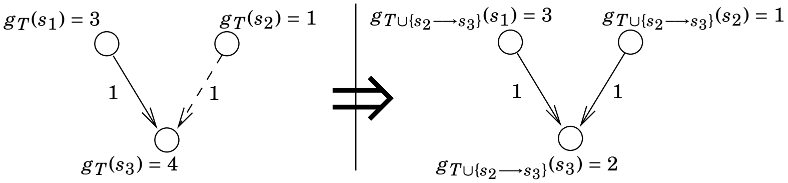
$g_T(s_1) = 3$
$g_T(s_2) = 1$
1
1
$g_T(s_3) = 4$
$g_{T\cup\{s_2 \longrightarrow s_3\}}(s_1) = 3$
$g_{T\cup\{s_2 \longrightarrow s_3\}}(s_2) = 1$
1
1
$g_{T\cup\{s_2 \longrightarrow s_3\}}(s_3) = 2$

Figure 6.1: Monotonicity example

Here, we leave open how the functions f and g relate to each other. Later on, we will consider possible relationships between f and g.

Algorithm 9 Best-first search for weighted state spaces

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, guiding function $f$, selection width $\beta$, goal states $\mathscr{G}$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathscr{H} \leftarrow \{\langle s, 0\rangle \mid s \in \mathscr{I}\}$
  while $\mathscr{H} \neq \emptyset$ do
    $\mathscr{L}_i \leftarrow select_\beta(\{\langle s, s.g\rangle \in \mathscr{H} \mid \forall \langle s', s'.g\rangle \in \mathscr{H}.f(s) \leq f(s')\})$
    if $\{s \mid \langle s, s.g\rangle \in \mathscr{L}_i\} \cap \mathscr{G} \neq \emptyset$ then
      return $GeneratePath(\{s \mid \langle s, s.g\rangle \in \mathscr{L}_i\} \cap \mathscr{G})$
    end if
    for all $\langle s, s.g\rangle \in \mathscr{L}_i$ do
      $\mathscr{H} \leftarrow \mathscr{H} \cup nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathscr{H} \leftarrow \{\langle s, s.g\rangle \in \mathscr{H} \mid \neg\exists g' \leq s.g \in \mathbb{K}.\langle s, g'\rangle \in \bigcup_{j=0}^{i-1} \mathscr{L}_j \wedge \neg\exists g' < s.g \in \mathbb{K}.\langle s, g'\rangle \in \mathscr{H}\}$
  end while
  return $false$
```

Note that duplicate detection, in the last line of the algorithm, is now refined to the removal of states from the search horizon $\mathscr{H}$ which have been encountered before with a lower or equally valued cumulated cost. If, however, in an earlier encounter the cumulated cost was greater, then the state should be re-opened. Of course, such a re-opening introduces redundancy in the search, but can often not be avoided if we wish to preserve cost-optimality. Besides that, duplicate detection considers multiple appearances of a state in $\mathscr{H}$ itself. When this occurs, only the smallest g-value should be maintained.

6.4 Uniform-cost Search

Now we come to another practical instance of best-first search. Given a monotonic cumulated cost function $g : \mathscr{S} \to \mathbb{K}$, if we say that $f = g$, then Algorithm 8 denotes what is referred to by e.g. Korf (1992), Pearl (1984), and Russell and Norvig (1995) as *uniform-cost search*, and as *lowest-cost-first search* by Poole et al. (1998); it is also known as *Dijkstra's search*, from Dijkstra (1959). With the same assumptions, Algorithm 9 represents a computationally optimised version of uniform-cost search. There, however, the duplicate detection is unnecessarily complicated. This follows from the following observation concerning uniform-cost search:

Lemma 2. *If we consider monotonic g for uniform-cost search, then for any states $s, s' \in \mathscr{S}$, if s is selected by the select$_\beta$ function in round i, and s' is selected by the select$_\beta$ function in round $j > i$, then necessarily $s.g \leq s'.g$. This is true independent of the value of β.*

Since Lemma 2 also holds when $s = s'$, it follows that in duplicate detection it suffices to check for earlier encounters of a state, independent of its g-value.

From Lemma 2 it follows that the first time a state $s \in \mathscr{G}$ is discovered, a trace from $\mathscr{I}$ to s is discovered with $g(s) = \mathbb{d}(\mathscr{I}, \{s\})$. Actually, if it is the very first goal state we encounter, then $g(s) = \mathbb{d}(\mathscr{I}, \mathscr{G})$.

We call the second property *cost-optimality*, which is defined in Definition 15.

Definition 15 (Cost-optimality of a best-first search). *Given a weighted state space $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$ and a set of goal states $\mathscr{G} \subseteq \mathscr{S}$, we call a best-first search* cost-optimal *iff it is ensured that it always returns a trace from $\mathscr{I}$ to a goal state $s \in \mathscr{G}$ with $f(s) = \mathbb{d}(\mathscr{I}, \mathscr{G})$ (unless $\mathscr{G} = \emptyset$ or unreachable).*

For instance, in general, breadth-first search applied on a weighted state space (which is represented in Algorithm 9 by choosing an appropriate f, such as $f(s) = 0$ **if** $s \in \mathscr{I}$, and $f(s) = f(s') + 1$, **if** $\exists \ell \in \mathscr{A}.s' \xrightarrow{\ell} s$,[3] and furthermore having $\beta = \infty$), is not cost-optimal, unless searching is continued once a trace to a goal state is found until there are no more states to explore. This is because there is no necessary correlation between the length of a trace and its total weight; so the first time a goal state is reached does not necessarily mean that a minimal trace to a goal state is found. On the other hand, in a weighted state space where for all $\ell \in \mathscr{A}$ $\mathscr{C}(\ell) = 1$, breadth-first search is cost-optimal.

Finally, depth-first search can be achieved in Algorithm 9 by e.g. determining f on-the-fly as $f(s) = 0$, **if** $s \in \mathfrak{I}$, and $f(s) = f(s') - 1$, **if** $\exists \ell \in \mathscr{A}.s' \xrightarrow{\ell} s$ and setting $\beta = 1$.

[3] Note that as with cumulated cost functions, f is technically not a function here, as a state s may receive different f-values during a search. It is, however, a (partial) function at any specific moment of the search.

6.5 Bounded Searches

In the remainder of this chapter, in order to keep things as general and clear as possible, we abstract away from the specifics to keep track of cumulated costs of states. For this reason, we adapt Algorithm 8, such that Algorithm 9 can be seen as an instance of it. In order to achieve this, we introduce a new duplicate detection function *DuplicateFree*, which performs duplicate detection on a given set of states (possibly accompanied by their stored g-values). This can either be the straightforward variant, i.e. checking for earlier encounters, or the more sophisticated one as presented in Algorithm 9. Furthermore, in the following algorithms we do not write $\langle s, s.g \rangle$, but only s, even when in practice one might need to keep track of $s.g$. We also take this more general stance when dealing with the detection of goal states. The result is presented in Algorithm 10.

Algorithm 10 General Best-first search

```
Require: $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$, guiding function $f$, selection width $\beta$, goal states $\mathcal{G}$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathcal{H} \leftarrow \mathcal{I}$
  while $\mathcal{H} \neq \emptyset$ do
    $\mathcal{L}_i \leftarrow select_\beta(\{s \in \mathcal{H} \mid \forall s' \in \mathcal{H}. f(s) \leq f(s')\})$
    $\mathcal{H} \leftarrow \mathcal{H} \setminus \mathcal{L}_i$
    if $\mathcal{L}_i \cap \mathcal{G} \neq \emptyset$ then
      return $GeneratePath(\mathcal{L}_i \cap \mathcal{G})$
    end if
    for all $s \in \mathcal{L}_i$ do
      $\mathcal{H} \leftarrow \mathcal{H} \cup nxt_{\mathcal{M}}(s, en_{\mathcal{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathcal{H} \leftarrow DuplicateFree(\mathcal{H}, \bigcup_{j=0}^{i-1} \mathcal{L}_j)$
  end while
  return false
```

Next, considering this more general algorithm, we can extend it further by introducing a *guiding upper-bound* U. It can be used to prune away parts of the state space during generation, while still preserving cost-optimality, given that U is indeed an upper-bound to $\mathbb{d}(\mathcal{I}, \mathcal{G})$. Such an upper-bound makes it practically possible to deal with reachability problems in infinite state spaces. Algorithm 11 shows best-first search using a cost upper-bound U. It is important to take U into account each time duplicate detection is performed, as now we need not only remove states from the search horizon which have been explored before, but we also should remove states with an f-value greater than U. In practice, an algorithm using U is referred to as a *cost-bounded search*.

Algorithm 11 Cost-bounded best-first search

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, guiding function $f$, selection width $\beta$, goal states $\mathscr{G}$,
  cost upper-bound $U$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathscr{H} \leftarrow \{s \in \mathscr{I} \mid f(s) < U\}$
  while $\mathscr{H} \neq \emptyset$ do
    $\mathscr{L}_i \leftarrow select_\beta(\{s \in \mathscr{H} \mid \forall s' \in \mathscr{H}.f(s) \leq f(s')\})$
    $\mathscr{H} \leftarrow \mathscr{H} \setminus \mathscr{L}_i$
    if $\mathscr{L}_i \cap \mathscr{G} \neq \emptyset$ then
      return $GeneratePath(\mathscr{L}_i \cap \mathscr{G})$
    end if
    for all $s \in \mathscr{L}_i$ do
      $\mathscr{H} \leftarrow \mathscr{H} \cup nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathscr{H} \leftarrow DuplicateFree(\mathscr{H} \setminus \{s \in \mathscr{H} \mid f(s) \geq U, \bigcup_{j=0}^{i-1} \mathscr{L}_j\})$
  end while
  return $false$
```

Common instances of cost-bounded best-first search are cost-bounded breadth-first search, cost-bounded depth-first search, and cost-bounded uniform-cost search, which can be obtained from Algorithm 11 in obvious ways. It should be pointed out that in some searches, in particular cost-bounded depth-first search, a bound on a monotonic g-function does not work well in the presence of infinite traces where U is never reached along the way, which is possible, since transitions may have a cost of 0 associated with them. One can avoid such a situation by also bounding the *depth* of the search. This is, for example, used in the model checker SPIN, the details of which are stated in Section 7.3.3.

As bounding the cost can in a way be seen as pruning in the *depth* of the state space, a related technique is to select in each round of the search a subset of states, thereby setting the *width* of a state space. In the presented algorithms this possibility is readily available by allowing β, besides 1 and ∞, to take any natural number as input. Note that there is an important difference between the two techniques, other than the dimension they affect; as they are defined currently, cost-bounding really *prunes* states, i.e. states are permanently removed from the search, whereas states which are not selected by $select_\beta$ remain in the search horizon and therefore can still be selected later on in the search. This is done since in the depth pruning can safely be performed without compromising cost-optimality, if $U \geq \mathrm{d}(\mathscr{I},\mathscr{G})$, while in the width this is not the case.

There are situations, however, where pruning in the width might be desirable. In practice, pruning in the width results in searches which require an amount of memory linear to the maximum search depth. Algorithm 12 includes the possibility to prune states in the width in each round of the search; by setting the flag *WidthPruning* we turn the option on. An example of algorithms using pruning in the width is *nearest neighbour search*, also known as *Gradient Descent* (see e.g. Zhang (1999)), which uses a cumulated cost function for guiding, i.e. $f = g$, and a selection width $\beta = 1$. It exactly traverses the trace which follows from always selecting the transition with the least cost among the outgoing transitions of the current state. It is identified by Zhang (1999) as a kind of *local search*, where local search is a type of search which looks for local optima, i.e. it never looks further ahead along a trace than the next transition(s) to take.

At this point, it is important to mention a phenomenon, referred to in the literature as *tie-breaking*; when, in a given round of a search algorithm, β states need to be selected, but more than β states have the best f-value, clearly selection needs to be done based on some other criteria. In practice, this may e.g. depend on the order of encountering the states. However, being forced to resort to tie-breaking is in general undesired, since it degrades the influence of the guiding function. Later on, in Chapter 8, we consider the avoidance of tie-breaking for a specific search algorithm.

6.6 Iterative and Bound-updating Searches

If we add the possibility, after finishing a search, to update the bounds, i.e. U and/or β, using e.g. the cumulated cost of a reached goal state or some prespecified interval, and build in the possibility to continue the search using the new bound values, then we achieve what we in general might call a *bound-updating search*. If we, after each update, reset the search, such that we start again with $\mathscr{I}$, then we refer to this search as an *iterative search*. These extensions are presented in Algorithm 13. For the bound-updating, the *bound setting function newdepthbound*: $\mathbb{K} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \rightarrow \mathbb{K}$ and the bound setting function *newwidthbound*: $\mathbb{N} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \rightarrow \mathbb{N}$ are used to calculate new upper-bounds. These functions typically take the current search horizon and the set of reached goal states into account. Besides that, we can turn on/off the iterative searching, i.e. reset the search completely, by setting the flag *IterativeSearch* to *true*/*false*. Note that we have replaced the **return** terms by **output** terms, in order to express that by executing these lines, the algorithm does not necessarily exit.

Iterative searches often calculate new bounds for the next iteration during the execution of the current iteration, taking states into account, which later on in the search are possibly pruned away. To express this behaviour, Algorithm 13 uses the *bound storing functions updatedepth and updatewidth*. The functions *updatedepth* :

Algorithm 12 Cost-bounded best-first search with width pruning

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, guiding function $f$, selection width $\beta$, goal states $\mathscr{G}$,
    cost upper-bound $U$, flag $WidthPruning$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathscr{H} \leftarrow \{s \in \mathscr{I} \mid f(s) < U\}$
  while $\mathscr{H} \neq \emptyset$ do
    $\mathscr{L}_i \leftarrow select_\beta(\{s \in \mathscr{H} \mid \forall s' \in \mathscr{H}. f(s) \leq f(s')\})$
    if $WidthPruning$ then
      $\mathscr{H} \leftarrow \emptyset$
    else
      $\mathscr{H} \leftarrow \mathscr{H} \setminus \mathscr{L}_i$
    end if
    if $\mathscr{L}_i \cap \mathscr{G} \neq \emptyset$ then
      return $GeneratePath(\mathscr{L}_i \cap \mathscr{G})$
    end if
    for all $s \in \mathscr{L}_i$ do
      $\mathscr{H} \leftarrow \mathscr{H} \cup nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathscr{H} \leftarrow DuplicateFree(\mathscr{H} \setminus \{s \in \mathscr{H} \mid f(s) \geq U\}, \bigcup_{j=0}^{i-1} \mathscr{L}_j)$
  end while
  return $false$
```

$\mathbb{K} \times \mathbb{K} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{K}$ and *updatewidth* : $\mathbb{N} \times \mathbb{N} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{N}$ store the bounds for the next iteration in U' and β', respectively. They can e.g. keep track of the minimum encountered cumulated cost still greater than the current upper-bound, or the maximum cumulated cost, making the search more greedy in a sense. Another option found in practice is to have the update functions increase the bounds by fixed intervals.

In particular, if we update U to $g(s)$ whenever we encounter a goal state s, and we do not reset the search, we achieve so-called *Branch-and-Bound* (BnB) algorithms, a technique e.g. described by Kumar (1992). In a similar manner, the possibility to restart the search and update U and β opens up the possibility to iteratively search larger and larger spaces. If we perform an iterative search while successively raising the selection width by some interval, and apply pruning in the width, then we perform *iterative broadening* (Ginsberg and Harvey, 1992). In case we successively raise the upper-bound, however, to the minimal total cost found among visited, but not expanded, states, we use *iterative deepening*; for instance, Korf (1985) analyses depth-first iterative deepening. In Algorithm 13, iterative deepening can be achieved by defining $updatedepth(U', U, \mathscr{H}, \mathscr{L}_i \cap \mathscr{G}) = min\{U', newdepthbound(U, \mathscr{H}, \mathscr{L}_i \cap \mathscr{G})\}$, and $newdepthbound(U, \mathscr{H}, \mathscr{L}_i \cap \mathscr{G})$ as follows:

$$\begin{aligned} &newdepthbound(U, \mathscr{H}, \mathscr{L}_i \cap \mathscr{G}) = \\ &\qquad 0, \textbf{if } \mathscr{L}_i \cap \mathscr{G} \neq \emptyset \\ &\qquad min\{f(s) > U \mid s \in \mathscr{H}\}, \textbf{otherwise} \end{aligned}$$

An important difference between depth-first BnB search and iterative deepening is that the first search continues from the current point once a goal state has been found, while in such a situation, the latter stops the current iteration and restarts the search. The definition of *newdepthbound* presented above achieves the desired behaviour for iterative deepening: by setting U to 0 once a goal state has been found (done by the first appearance of *newdepthbound* in Algorithm 13), the current iteration will terminate in the next round. As long as no goal state is found, the second appearance of *newdepthbound* in Algorithm 13 keeps track of the bound to use in the next iteration, which is the smallest encountered total cost greater than U, and stores it in U'. This value can be used once the current iteration is finished.

Finally, note in the last line of Algorithm 13 that the iterative searches can be stopped entirely by setting any of the bounds to 0.

6.7 Selecting Extra States

Up to now, we have only considered searches which in each round select the best states according to the guiding function f. In some cases, a bound is set on the number of states selected in a round, such that if there are too many best states, some of them

Algorithm 13 Cost-bounded iterative and bound-updating best-first search

```
Require: $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$, guiding function $f$, selection width $\beta$, goal states
    $\mathcal{G}$, cost upper-bound $U$, flag $WidthPruning$, bound-setting and updating functions
    $newdepthbound : \mathbb{K} \times 2^{\mathcal{S}} \times 2^{\mathcal{S}} \to \mathbb{K}$, $newwidthbound : \mathbb{N} \times 2^{\mathcal{S}} \times 2^{\mathcal{S}} \to \mathbb{N}$, $updatedepth$ :
    $\mathbb{K} \times \mathbb{K} \times 2^{\mathcal{S}} \times 2^{\mathcal{S}} \to \mathbb{K}$, and $updatewidth : \mathbb{N} \times \mathbb{N} \times 2^{\mathcal{S}} \times 2^{\mathcal{S}} \to \mathbb{N}$, flag $IterativeSearch$
Ensure: If found, a trace to a goal state is returned
  $Start \leftarrow true$
  $U', \beta' \leftarrow \infty$
  repeat
    if $IterativeSearch$ or $Start$ then
      $Start \leftarrow false$
      $i \leftarrow 0$
      $\mathcal{H} \leftarrow \{s \in \mathcal{I} \mid f(s) < U\}$
    end if
    while $\mathcal{H} \neq \emptyset$ do
      $\mathcal{L}_i \leftarrow select_{\beta}(\{s \in \mathcal{H} \mid \forall s' \in \mathcal{H}.f(s) \leq f(s')\})$
      if $WidthPruning$ then
        $\mathcal{H} \leftarrow \emptyset$
      else
        $\mathcal{H} \leftarrow \mathcal{H} \setminus \mathcal{L}_i$
      end if
      if $\mathcal{L}_i \cap \mathcal{G} \neq \emptyset$ then
        output $GeneratePath(\mathcal{L}_i \cap \mathcal{G})$
        $U \leftarrow newdepthbound(U, \mathcal{H}, \mathcal{L}_i \cap \mathcal{G})$
        $\beta \leftarrow newwidthbound(\beta, \mathcal{H}, \mathcal{L}_i \cap \mathcal{G})$
      end if
      for all $s \in \mathcal{L}_i$ do
        $\mathcal{H} \leftarrow \mathcal{H} \cup \ nxt_{\mathcal{M}}(s, en_{\mathcal{M}}(s))$
      end for
      $U' \leftarrow updatedepth(U', U, \mathcal{H}, \mathcal{L}_i \cap \mathcal{G})$
      $\beta' \leftarrow updatewidth(\beta', \beta, \mathcal{H}, \mathcal{L}_i \cap \mathcal{G})$
      $i \leftarrow i + 1$
      $\mathcal{H} \leftarrow DuplicateFree(\mathcal{H} \setminus \{s \in \mathcal{H} \mid f(s) \geq U\}, \bigcup_{j=0}^{i-1} \mathcal{L}_j)$
    end while
    output $false$
    $U \leftarrow U'$
    $\beta \leftarrow \beta'$
  until $\beta = 0 \vee U = 0$
```

will be either postponed or pruned (depending on the pruning setting).

In cases where there are fewer best states than the number allowed by the width bound, *selecting extra states* up to the width bound would be an option. These extra states would then typically be the next-best states. An example of an algorithm which illustrates this technique perfectly is called *k-best-first search* (Felner et al., 2003), therefore we adopt this name here. It practically compensates any inaccuracies in f by selecting more than only the best states, whenever the width bound allows it. Algorithm 14 incorporates this technique, which can be switched on by setting the flag *BestStatesOnly* to *false*.

6.8 Guiding with Heuristics

In the previous sections, given instances of the best-first searches presented always applied a cumulated cost function g as f. The result of choosing a cumulated cost function as the guiding function is that in most cases cost-optimality can be guaranteed (one exception to this being whenever pruning in the width is applied). Though this guarantee is a very strong point, in practice, state spaces can be so large that these searches require too much time and memory before they even provide one trace to a goal state. One of the main reasons for this is that the guiding function does not consider the future of traces, i.e. it does not try to determine what the structure is of the not yet discovered part of the state space.

One way to do this is to incorporate a so-called *heuristic function*. Such a function is typically used to predict the remaining cost needed to reach from a given state a goal state. In the literature, a heuristic function is often referred to as h, therefore here we consider a heuristic or estimation function $h : \mathscr{S} \to \mathbb{K}$. Most searches employing heuristics are not cost-optimal, though. Particularly in application areas where also near-optimal solutions to a problem are satisfactory, we may consider dropping cost-optimality of a search.

Figure 6.2 shows the typical relation between a cumulated cost function g and a heuristic function h to solve reachability problems within a weighted state space. A cumulated cost function reflects a real weighted distance along a trace, starting in $\mathscr{I}$, leading to a state s, while a heuristic function provides an estimated weighted distance between a state s and $\mathscr{G}$. In cases where the history of a state s, i.e. the trace leading from $\mathscr{I}$ to s, plays a role, typically $f(s) = g(s) + h(s)$, in other words, $f(s)$ takes (an estimation of) the whole weighted distance between $\mathscr{I}$ and $\mathscr{G}$ via s into account. One can imagine that with guiding functions like this, the further down we go into a state space, the more accurate f will be. Where the history is unimportant, we can use heuristics by e.g. setting $f(s) = h(s)$.

Considering the properties of heuristic functions, first of all, typically $h(s) = 0$ iff $s \in \mathscr{G}$, since naturally, the distance from a goal state to $\mathscr{G}$ is 0. In cases where $\mathscr{B} \neq \emptyset$,i.e.

Algorithm 14 Cost-bounded iterative and bound-updating k-best-first search

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, guiding function $f$, selection width $\beta$, goal states
  $\mathscr{G}$, cost upper-bound $U$, flag WidthPruning, bound-setting and updating functions
  $newdepthbound : \mathbb{K} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{K}$, $newwidthbound : \mathbb{N} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{N}$, $updatedepth :$
  $\mathbb{K} \times \mathbb{K} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{K}$, and $updatewidth : \mathbb{N} \times \mathbb{N} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{N}$, flag IterativeSearch,
  flag BestStatesOnly
Ensure: If found, a trace to a goal state is returned
  Start←true
  $U', \beta' \leftarrow \infty$
  repeat
    if IterativeSearch or Start then
      Start←false
      $i \leftarrow 0$
      $\mathscr{H} \leftarrow \{s \in \mathscr{I} \mid f(s) < U\}$
    end if
    while $\mathscr{H} \neq \emptyset$ do
      $\mathscr{L}_i = \emptyset$
      repeat
        $\mathscr{L}_i \leftarrow \mathscr{L}_i \cup select_{\beta-|\mathscr{L}_i|}(\{s \in \mathscr{H} \mid \forall s' \in \mathscr{H}. f(s) \leq f(s')\})$
        $\mathscr{H} \leftarrow \mathscr{H} \setminus \mathscr{L}_i$
      until BestStatesOnly or $(|\ \mathscr{L}_i\ | = \beta$ or $\mathscr{H} = \emptyset)$
      if WidthPruning then
        $\mathscr{H} \leftarrow \emptyset$
      end if
      if $\mathscr{L}_i \cap \mathscr{G} \neq \emptyset$ then
        output $GeneratePath(\mathscr{L}_i \cap \mathscr{G})$
        $U \leftarrow newdepthbound(U, \mathscr{H}, \mathscr{L}_i \cap \mathscr{G})$
        $\beta \leftarrow newwidthbound(\beta, \mathscr{H}, \mathscr{L}_i \cap \mathscr{G})$
      end if
      for all $s \in \mathscr{L}_i$ do
        $\mathscr{H} \leftarrow \mathscr{H} \cup\ nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
      end for
      $U' \leftarrow updatedepth(U', U, \mathscr{H}, \mathscr{L}_i \cap \mathscr{G})$
      $\beta' \leftarrow updatewidth(\beta', \beta', \mathscr{H}, \mathscr{L}_i \cap \mathscr{G})$
      $i \leftarrow i + 1$
      $\mathscr{H} \leftarrow DuplicateFree(\mathscr{H} \setminus \{s \in \mathscr{H} \mid f(s) \geq U\}, \bigcup_{j=0}^{i-1} \mathscr{L}_j)$
    end while
    output false
    $U \leftarrow U'$
    $\beta \leftarrow \beta'$
  until $\beta = 0 \vee U = 0$
```

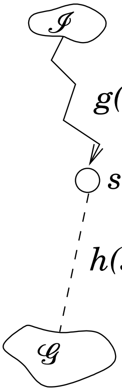

Figure 6.2: Cumulated cost function g and heuristic function h

there are undesired ('bad') states present, preferably, if $s \in \mathcal{B}$ then $h(s) = \infty$. If a heuristic function is, to some extent, successful in demotivating searches in the direction of bad states, then we say that the function incorporates some *deadlock avoidance*.

Of course, searches using heuristic functions heavily depend on the quality of those functions; a badly estimating heuristic may result in a bad search. In this thesis, we do not especially focus on the development of heuristic functions. In the literature, however, there are numerous examples of techniques to do this, e.g. by Edelkamp et al. (2001a), Edelkamp et al. (2004), Groce and Visser (2004), and Kupferschmid et al. (2006). One of these techniques can be called *relaxation of the conditions* (Pearl, 1984), which yields so-called *admissible* heuristic functions. Admissibility of a heuristic function, and, related to this, *monotonicity* of a heuristic function are defined in Definition 16.

Definition 16 (Admissible and consistent heuristics). *A heuristic function $h : \mathcal{S} \to \mathbb{K}$ is called*

- admissible*, iff for all $s \in \mathcal{S}$, we have $h(s) \leq \mathrm{d}(s, \mathcal{G})$*
- consistent*, iff for each $s, s' \in \mathcal{S}$ and $\ell \in \mathcal{A}$ such that $s \xrightarrow{\ell} s'$ and $\mathcal{C}(\ell) = c$, we have $h(s) \leq c + h(s')$*

Lemma 3. *If a heuristic function $h : \mathcal{S} \to \mathbb{K}$ is consistent, then it is admissible.*

Proof. Consider a trace $s \xrightarrow{\ell_0} s_0 \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_n} s_n$ with $s_n \in \mathcal{G}$ and $\mathcal{C}(\ell_i) = c_i$ for $0 \leq i \leq n$, such that $c_0 + \ldots + c_n = \mathrm{d}(s, \mathcal{G})$. Since $h(s)$ is consistent, we have $h(s) \leq c_0 + h(s_0)$. Furthermore, since $h(s_0) \leq c_1 + h(s_1)$, we have $h(s) \leq c_0 + c_1 + h(s_1)$. In this manner, it follows that $h(s) \leq c_0 + \ldots + c_n + h(s_n)$, which means $h(s) \leq c_0 + \ldots + c_n$, since by $s_n \in \mathscr{G}$, $h(s_n) = 0$. Finally, because $c_0 + \ldots + c_n = \text{d}(s, \mathscr{G})$, we have $h(s) \leq \text{d}(s, \mathscr{G})$. □

Relaxation of the conditions works like this: Typically, one can identify at least one factor of a problem, which makes it hard to solve. By removing this factor, often a variant of the problem is obtained, for which a much easier estimation function can be constructed. Surely, a correct estimation function for such a relaxed version of the problem constitutes an admissible estimation function for the real problem, i.e. the estimation function will provide optimistic estimations for the real problem.

Let us consider some examples of heuristic searches. If we consider $f(s) = h(s)$, do not prune in the width, and set $\beta = 1$, then we obtain *greedy search*, at least as defined by Russell and Norvig (1995). Curiously, another definition of greedy search is present in the literature, for instance provided by Zhou and Hansen (2005), which corresponds with using a guiding function $f(s) = g(s) + h(s)$, pruning in the width, and a selection width $\beta = 1$. To avoid confusion, we dub the second definition *heuristic nearest neighbour search*, as it can be seen as a nearest neighbour search with heuristics added to it.

If we choose $f(s) = g(s) + h(s)$, and apply pruning in the width and the selection of extra states, then we obtain *beam search for weighted state spaces*, based on classic detailed beam search, as presented by e.g. Valente and Alves (2005a).[4] In this context, β, which may be set to any value, is referred to as the *beam width*. If we have $f(s) = g(s) + h(s)$, use an admissible estimation function, do not prune in the width, and only select the best states, then we either achieve A^* (if $\beta = 1$) or *set-based* A^* (if $\beta = \infty$) (Hart et al., 1968).

A^* is cost-optimal, and an important property of A^*, reported by Dechter and Pearl (1985), is that, when using a consistent estimation function, it expands the minimum number of states before a goal state is found, compared to all other algorithms which are cost-optimal, up to tie-breaking. It should be noted that, besides A^*, it is hard to state anything general about whether the admissibility (or monotonicity) of a heuristic yields a good performance of a search using this heuristic.

Finally, we can compare admissible estimation functions for a problem as follows:

Definition 17 (Informedness of heuristics (Pearl, 1984)). *A heuristic function h_2 is said to be* more informed than h_1 *if both are admissible and*

$$\forall s \in \mathscr{S} \setminus \{\mathscr{G} \cup \mathscr{B}\}. h_2(s) > h_1(s)$$

When considering admissible functions, Definition 17 expresses that given a problem to solve, preferably a heuristic function to help in solving the problem is as accurate as possible in predicting the actual distance between a given state and a set of goal states.

[4] The original definition of beam search is much less restrictive concerning the guiding function, requiring only pruning in the width and the selection of extra states.

In the upcoming chapters, we specifically focus on applying beam search on state spaces, since typically, in the literature, it is only applied on highly structured trees. In Chapter 8, we will see that we can achieve A^* by extending beam search. Observe the connection between beam search and the heuristic nearest neighbour search, as given in the previous paragraph.

6.9 Some Other Search Examples

At this point in the chapter, we briefly present some more examples of (classes of) searches present in the literature, before we continue with a final extension of our setting, and the consideration of action-based guiding. The amount of existing searches (and identified classes of searches) is vast, therefore it should be stressed that this is by no means a complete list.

General Branch-and-Bound Earlier in this chapter, we described the Branch-and-Bound technique; however, this technique is often considered to be one example of a much larger class called Branch-and-Bound (BnB) search. Nau et al. (1984) identify the class as *general* BnB and the instance as *ordinary* BnB. General BnB consists of all searches for which in each round there is the guarantee of still having at least one optimal solution in the search horizon, i.e. which are cost-optimal. Nau et al. (1984) show that A^* can be seen as an instance of general BnB.

Z Pearl (1984) identifies a type of search comprising a subclass of best-first search, which he calls *Z search*. A search is an instance of Z iff its guiding function incorporates a so-called *recursive-weight-computation*. An example of such a computation is calculating the cumulated cost value of a state. Therefore, uniform-cost search is an instance of Z, as is A^* (which is in fact an instance of Z^*). Heuristic depth-first search and greedy search, however, are not. Curiously, note that if we follow the original definition of beam search, we must conclude that some beam searches are instances of Z, while others are not.

ϵ-allowance A BnB technique, which is, like beam search, near-optimal. As described by Zhang (1999), it works as regular BnB, but it also takes a lower bound into account. Whenever a solution is found with a total cost no more than this lower bound plus a pre-given distance ϵ, the search stops. In our algorithms such a mechanism would be included in the *newdepthbound* function.

***T*-cut** This is, like ϵ-allowance and beam search, also a near-optimal algorithm (Zhang, 1999). It operates like BnB, but it terminates whenever some additional (resource-related) bound T is met, which e.g. expresses the maximal number of states to explore.

Tabu search The already mentioned nearest neighbour search is an example of a local search, meaning that during the search, only the next transition to take is considered, not looking at the global picture (i.e. taking the history and the further future into account). A drawback of local searches is that they tend to get trapped into local minima; a way to get around this problem is to include a backtracking facility, such that whenever a solution is found, the search backtracks to an earlier point, and continues searching in another direction. Tabu search does this, by maintaining a tabu list, in which parts of the searched space are kept (Zhang, 1999). States present in the tabu list will never be re-opened. In our algorithms, tabu search can be seen as iterative nearest neighbour, except that an iteration does not necessarily start from $\mathscr{I}$; it can start from any (sub)set of previously visited states. The tabu list can be taken into account in the *DuplicateFree* function, restricting duplicate detection to states present in the tabu list.

Randomised searches Randomised searches take certain exploration decisions at random. E.g. *simulated annealing* (Zhang, 1999) works in a way comparable with tabu search, except that both searching and backtracking is done in a random fashion. Searching is not always done best-first, but at random, and backtracking can be done at any point, instead of in particular situations. We can incorporate this kind of searches in our spectrum by allowing random decisions in our algorithms, for instance in the select_β function.

IDA * **and** ***MIDA*** * Iterative Deepening A^* (Korf, 1985) works as A^*, but uses a total cost upper-bound value to perform an iterative deepening version of it, updating the upper-bound for each iteration by a fixed interval. As it is a depth-first search, its space requirements are, unlike those of A^*, linear to the maximal search depth. Wah (1991) has analysed how updating the bounds impacts the search overhead; he describes *MIDA** as an *IDA** search, where the bounds are updated dynamically, using information collected at run-time.

MA * On the one hand, A^* may require a lot of memory, since all the encountered states need to be kept in memory, and on the other, *IDA** requires a minimal amount of memory, but maintains no trace information at all when moving to the next iteration. Chakrabarti et al. (1989) remark that there are searches situated between these two extremes. They keep track of the number of states expanded so far, and prune states out of the memory whenever this total number exceeds a pre-given maximum. Whenever this maximum is set to 0, *MA** behaves as *IDA**; whenever it is very large, *MA** behaves as A^*.

6.10 Multi-phase Searches

In this section, we propose a final generalisation of our best-first search, that we call *multi-phase* best-first search, which gives rise to the idea of compositionality of best-first searches. This extension is by no means common; in the literature we find some searches which may be identified as instances of multi-phase best-first search, but the general form is not described. The three possible instances we found are *filtered beam search* (reported, e.g., by Pinedo (1995); Si Ow and Morton (1988); Valente and Alves (2005c)), A_ϵ^*, described by Pearl (1984), and *heuristic depth-first search* (see e.g. Poole et al. (1998)). Besides the existence of these searches, our proposed extension is furthermore motivated by the work in Chapter 8, where we consider more instances of this best-first search and their applicability in practice.

In the extensions of best-first search thus far, it is possible to enable a range of options, thereby manipulating the way in which the $\mathscr{L}_i$ are constructed; we have e.g. a guiding function f, a cost upper-bound U, and a selection width β. A generalisation would be to allow a list of n guiding functions $f_1, \ldots, f_n$, a list of cost upper-bounds $U_1, \ldots, U_n$, and a list of selection widths $\beta_1, \ldots, \beta_n$. The lists could be used to construct an $\mathscr{L}_i$ through a number of intermediate phases, such that in practice, we basically stack search algorithms on top of each other. We will further illustrate this concept next.

Say we have an n-phase best-first search, with guiding functions $f_{1,\ldots,n}$, cost upper-bounds $U_{1,\ldots,n}$, selection widths $\beta_{1,\ldots,n}$, flags $\mathit{WidthPruning}_{1,\ldots,n}$, $\mathit{BestStatesOnly}_{1,\ldots,n}$, and update functions $\mathit{newdepthbound}_{1,\ldots,n}$ and $\mathit{newwidthbound}_{1,\ldots,n}$. In each round i of the algorithm, we construct $\mathscr{L}_i$ from $\mathscr{H}$ as follows:

1. In the first phase, we use function f_1 to select β_1 states from $\mathscr{H}$, either pruning or not (depending on $\mathit{WidthPruning}_1$), and either selecting only the best states or not (depending on $\mathit{BestStatesOnly}_1$). The selected states together form the intermediate search horizon $\mathscr{H}_1$. If a goal state is found, a trace from $\mathscr{I}$ to this goal state will be produced.

2. In the second phase, we take $\mathscr{H}_1$ and produce the intermediate search horizon $\mathscr{H}_2$ by applying function f_2 to select β_2 states from $\mathscr{H}_1$, either pruning or not (depending on $\mathit{WidthPruning}_2$), and either selecting only the best states or not (depending on $\mathit{BestStatesOnly}_2$).

3. ...

n. In the n^{th} phase, we take the intermediate search horizon $\mathscr{H}_{n-1}$, and produce $\mathscr{L}_i$ by applying function f_n to select β_n states from $\mathscr{H}_{n-1}$, either pruning or not (depending on $\mathit{WidthPruning}_n$), and either selecting only the best states or not (depending on $\mathit{BestStatesOnly}_n$).

By adopting such a multi-phased mechanism, we can compose search algorithms useful for several reasons. For instance, the aforementioned filtered beam search can be seen as a two-phase best-first search, where in the first phase states are selected based on some computationally cheap guiding function, which does not incorporate the history of the states, while the remaining states are pruned away. In the second phase, a second selection is performed, and pruning is again applied, using a more precise, but computationally much more expensive, guiding function. By this approach, we can avoid evaluating all states in the original search horizon with the computationally more expensive guiding function.

One can also imagine combining cost-optimal searches. For instance, an n-phase uniform-cost search could be useful to deal with so-called *multi-cost problems*, which are e.g. mentioned by Behrmann et al. (2005) and focussed on by Larsen and Rasmussen (2005). As an example, consider the problem of constructing a new building, where money, time and manpower are the three types of resources to consider. The goal is to find a way to construct the building, such that the amount of money needed should absolutely be minimised. Given this condition, the quickest possible solution should be chosen, and finally, given those two conditions, we should try to minimise the amount of manpower needed. Such a solution, in the state space of the specification of this problem represented by a trace, could be found by using a three-phase uniform-cost search, where $f_1(s) = g_1(s)$ keeps track of the amount of money spent, $f_2(s) = g_2(s)$ reports the time needed thus far, and $f_3(s) = g_3(s)$ reflects the total amount of manpower. One can imagine that changing the priorities of these three types of resources leads to different kinds of solutions, and that changing the order of the phases in the multi-phase uniform-cost search allows us to deal with these different priorities.

On a side note, it should be pointed out that multi-phase searches raise the interesting question how duplicate detection should be performed, or more specifically, when to re-open states and when not. For instance, if we open a state s, after computing both $f_1(s)$ and $f_2(s)$, and later, we re-encounter s, this time with a lower $f_1(s)$ value, but with a greater $f_2(s)$ value, what to do next? Of course, the re-opening policy should be decided based on the importance of the individual guiding functions.

Algorithm 15 shows our multi-phase best-first search, incorporating all the extensions previously introduced. The concept of this search will reappear in subsequent chapters when dealing with so-called *G-synchronised beam search* for general state spaces. In particular the way in which pruning in the width is performed here should be taken into account. The chosen policy is that whenever pruning in the width is set for a phase j, all states in horizon $\mathcal{H}_{j-1}$, which are not selected for the next phase, are permanently pruned from the overall search horizon (i.e. $\mathcal{H}_0$). States selected in phase j are not removed from $\mathcal{H}_0$ at this stage, since these may very well neither be selected nor pruned away in subsequent phases, in which case we should reconsider them in the next level of the search. The final selection of the level, though, once phase n has finished, is removed from the search horizon, since these states are going to be fully explored.

Returning to the previously mentioned instances of multi-phase best-first search, as explained before, filtered beam search applies two different types of beam search in two phases, subsequently, where in the first phase a computationally cheap guiding function is used, and the second phase deals with a more thorough selection among the remaining states. This search is described in more detail in Section 8.3.5, since we move our attention fully towards beam search in Chapter 8. The A^*_ϵ search can be seen as a multi-phase search, where in the first phase, standard (set-based) A^* with selection of extra states is used to make an intermediate selection of states. In this phase, all states need to be selected with an f-value not greater than the minimal f-value plus a pre-given value ϵ. In the second phase, $f_2(s) = h_2(s)$, where h_2 expresses a *search effort* estimate. The search effort estimate must not be confused with the estimated remaining cost along a trace; it concerns the remaining computational effort needed by the search algorithm to find a goal state. This second estimate is used to make a final selection of states for exploration from the intermediate set of states. This approach gives rise to the notion of *ϵ-admissibility*; an algorithm is ϵ-admissible iff it is guaranteed to find a solution not worse than the optimal solution plus ϵ. This notion is derived from the notion of admissibility of algorithms, which states that an algorithm is called admissible iff it is guaranteed to find an optimal solution, i.e. it is cost-optimal.

Finally, heuristic depth-first search is a depth-first search that uses an estimation function to decide for each s the order in which the successors need to be explored. Note that the estimation function is used locally per state here. A global way, considering the whole search horizon, would constitute a search which could be called *heuristic breadth-first search*. Heuristic depth-first search can be seen as a two-phase best-first search, where the first phase selects the states as a depth-first search would. For this, we can determine f_1 on-the-fly as $f_1(s) = 0$, **if** $s \in \mathscr{I}$, and $f_1(s) = f_1(s') - 1$, **if** $\exists \ell \in \mathscr{A}. s' \xrightarrow{\ell} s$. In the second phase, an estimation function $h : \mathscr{S} \to \mathbb{K}$ is applied as f_2 to select one state from the set of states delivered by the first phase.

6.11 Action-based Guiding

When referring to guiding functions in this chapter, we always adopted the approach to apply such a function on a state, and using the outcome for the guiding. As state spaces comprise of states and labelled transitions, an obvious alternative approach is to base the guiding on transitions. Here, we refer to this alternative as *action-based guiding*.

A first possibility in this field that comes to mind, is to associate priorities to action labels. More precisely, we can use a *priority* function *prio* $: \mathscr{A} \to \mathbb{Z}$ to guide a search. With such a function, at each state in the search, we can select up to a certain number of transitions, which have the *highest* priority (as opposed to the *lowest* f-value in other

Algorithm 15 Multi-phase best-first search

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, guiding functions $f_1, \dots, f_n$, selection widths
  $\beta_1, \dots, \beta_n$, goal states $\mathscr{G}$, cost upper-bounds $U_1, \dots, U_n$, flags $\mathit{WidthPruning}_{1,\dots,n}$,
  bound-setting and updating functions $\mathit{newdepthbound}_{1,\dots,n} : \mathbb{K} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{K}$,
  $\mathit{newwidthbound}_{1,\dots,n} : \mathbb{N} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{N}$, $\mathit{updatedepth} : \mathbb{K} \times \mathbb{K} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{K}$, and
  $\mathit{updatewidth} : \mathbb{N} \times \mathbb{N} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \to \mathbb{N}$, flag $\mathit{IterativeSearch}$, flags $\mathit{BestStatesOnly}_{1,\dots,n}$
Ensure: If found, a trace to a goal state is returned
  $\mathit{Start} \leftarrow \mathit{true}$; $U'_1, \beta'_1 \leftarrow \infty$
  repeat
    if $\mathit{IterativeSearch}$ or $\mathit{Start}$ then
      $\mathit{Start} \leftarrow \mathit{false}$; $i \leftarrow 0$; $\mathscr{H}_0 \leftarrow \{s \in \mathscr{I} \mid f(s) < U_1\}$
    end if
    while $\mathscr{H}_0 \neq \emptyset$ do
      for $j = 1$ to $n$ do
        $\mathscr{H}_j = \emptyset$
        repeat
          $\mathscr{H}_j \leftarrow \mathscr{H}_j \cup \mathit{select}_{\beta_j - |\mathscr{H}_j|}(\{s \in \mathscr{H}_{j-1} \mid \forall s' \in \mathscr{H}_{j-1}.f_j(s) \le f_j(s')\})$
        until $\mathit{BestStatesOnly}_j$ or $(\mid \mathscr{H}_j \mid = \beta_j$ or $\mid \mathscr{H}_j \mid = \mid \mathscr{H}_{j-1} \mid)$
        if $\mathit{WidthPruning}_j$ then
          $\mathscr{H}_0 \leftarrow \mathscr{H}_0 \setminus (\mathscr{H}_{j-1} \setminus \mathscr{H}_j)$
        end if
        if $\mathscr{H}_j \cap \mathscr{G} \neq \emptyset$ then
          output $\mathit{GeneratePath}(\mathscr{H}_j \cap \mathscr{G})$
          $U_j \leftarrow \mathit{newdepthbound}_j(U_j, \mathscr{H}_{j-1}, \mathscr{H}_j \cap \mathscr{G})$
          $\beta_j \leftarrow \mathit{newwidthbound}_j(\beta_j, \mathscr{H}_{j-1}, \mathscr{H}_j \cap \mathscr{G})$
        end if
        $\mathscr{H}_j \leftarrow \mathscr{H}_j \setminus \{s \in \mathscr{H}_j \mid f_j(s) \ge U_j\}$
      end for
      $\mathscr{L}_i \leftarrow \mathscr{H}_j$; $\mathscr{H}_0 \leftarrow \mathscr{H}_0 \setminus \mathscr{L}_i$
      for all $s \in \mathscr{L}_i$ do
        $\mathscr{H}_0 \leftarrow \mathscr{H}_0 \cup \ \mathit{nxt}_{\mathscr{M}}(s, \mathit{en}_{\mathscr{M}}(s))$
      end for
      $U'_1 \leftarrow \mathit{updatedepth}(U'_1, U_1, \mathscr{H}_0, \mathscr{H}_1 \cap \mathscr{G})$
      $\beta'_1 \leftarrow \mathit{updatewidth}(\beta'_1, \beta_1, \mathscr{H}_0, \mathscr{H}_1 \cap \mathscr{G})$
      $i \leftarrow i + 1$
      $\mathscr{H}_0 \leftarrow \mathit{DuplicateFree}(\mathscr{H}_0 \setminus \{s \in \mathscr{H}_0 \mid f(s) \ge U_1\}, \bigcup_{j=0}^{i-1} \mathscr{L}_j)$
    end while
    output $\mathit{false}$
    $U_1 \leftarrow U'_1$; $\beta_1 \leftarrow \beta'_1$
  until $\beta = 0 \vee U = 0$
```

guiding functions). Obviously, such a function does not explicitly take the history of a state into account, but only considers the next transition(s) to take. It is, on the other hand, implicitly important, since the outgoing transitions of different states cannot be compared fairly. Consider a state space in which each trace leads to a goal state $s \in \mathscr{G}$, and contains each $\ell \in \mathscr{A}$ exactly once. This is, for instance, a common situation when a scheduling problem has been modelled (as will be explained in more detail in the next chapter). Within this state space, we wish to guide the search by associating priorities with actions, thereby stimulating the early execution of highly important actions. Now, let us look at states s and s' in Figure 6.3. Say that we have $\ell, \ell' \in \mathscr{A}$, and a guiding function $prio : \mathscr{A} \to \mathbb{Z}$ with $prio(\ell) > prio(\ell')$, based on the fact that the execution of action ℓ has a higher priority than the execution of action ℓ'. Clearly, in the situation of state s, where both ℓ and ℓ' are possibilities for exploration, if we need to choose one transition based on *prio*, we will choose ℓ. If we would take all the outgoing transitions of states s and s' together into account, and simply make the same decision based on *prio*, than we would make a mistake. Note that in the trace leading from $\mathscr{I}$ to s', action ℓ has already been fired once. In other words, in s' we are in a completely different situation, due to the fact that s' has a history different from s; in fact, it might be that s' represents a situation in which we have made more progress with the problem than in state s, thereby we only have actions left to execute with lower priorities. As we do not want to delay or even prune away this more promising trace, histories of states are important when selecting transitions, and therefore we may only compare transitions exiting the same state.

The inability of priority functions to globally compare states may be a limitation of this kind of functions, compared to state-based (cost) functions, but priority functions also have advantages; first of all, the computational complexity is, in general, far less. Priority functions usually only encompass looking up a priority for a given action label (in e.g. a table), while cost functions may incorporate complex calculations, particularly when heuristics are involved. Second of all, no cost values are involved with a priority function, therefore such a function can be applied on a, more traditional, unweighted state space, which implies that the cheaper version of duplicate detection can be performed, in which g-values are not checked.

Of course, a way to explicitly include the history of states is to define $prio : \mathscr{S} \times \mathscr{A} \to \mathbb{Z}$. Functions like this can also be used to make the assignment of priorities to actions more dynamic; note that a function $prio : \mathscr{A} \to \mathbb{Z}$ assigns fixed priorities to action labels, but a function $prio : \mathscr{S} \times \mathscr{A} \to \mathbb{Z}$ may assign different priorities to the same action label at different times, depending on the situation, i.e. state. Fixed priorities are, nevertheless, practically interesting; when used for scheduling problems, they seem to resemble the *dispatch* scheduling strategy in Artificial Intelligence terminology, as described e.g. by Si Ow and Morton (1988). Finally, both fixed and dynamic priority assignment functions, although mostly applied on states, can be found in the literature on *priority beam search* for well-structured trees, as explained in upcoming chapters.

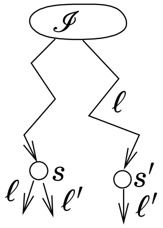

Figure 6.3: The selection of transitions

E.g. Valente and Alves (2004, 2005a,b,c) report on using priority beam search for well-structured trees.

6.12 DMC Glossary

A

A*** A best-first search with $f = g + h$, g being a cumulated cost function $g : \mathscr{S} \to \mathbb{K}$ and h being an (admissible) estimation function $h : \mathscr{S} \to \mathbb{K}$, no pruning in the width, no selection of extra states, and $\beta = 1$. If $\beta = \infty$, then we call the search *set-based A^**. *(See page 137)*

$\boldsymbol{A^*_\epsilon}$ A two-phase best-first search, where in the first phase, standard (set-based) A^* with selection of extra states is used to make an intermediate selection of states. In this phase, all states need to be selected with an f-value not greater than the minimal f-value plus a pre-given value ϵ. In the second phase, $f_2(s) = h_2(s)$, where h_2 expresses a search effort estimate, concerning the remaining computational effort needed by the search algorithm to find a goal state. *(See page 142)*

action-based guiding Guiding a search using a function $prio : \mathscr{A} \to \mathbb{Z}$, which assigns priorities to action labels, instead of a function f on states. Typically, in each round of action-based guided searches, those transitions are selected for exploration, which have an action label with the (locally) highest priority. *(See page 142)*

B

beam search A best-first search with $f = g + h$, g being a cumulated cost function $g : \mathscr{S} \to \mathbb{K}$ and h being an estimation function $h : \mathscr{S} \to \mathbb{K}$, pruning in the width, and the selection of extra states. In this context, β, which may be set to any value, is referred to as the beam width. *(See page 137)*

best-first search A class of searches, where a guiding function f is employed to guide the search through a state space. *(See page 121)*

bound setting function *newdepthbound* The function *newdepthbound* : $\mathbb{K} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \rightarrow \mathbb{K}$ sets a given depth upper-bound to a new value. The function typically takes the current search horizon and the set of reached goal states into account. Often, *newdepthbound* either sets the upper-bound to the cumulated cost associated with an already found goal state, or raises the value of the upper-bound by some fixed interval. It is used in cost-updating searches. *(See page 130)*

bound setting function *newwidthbound* The function *newwidthbound* : $\mathbb{N} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \rightarrow \mathbb{N}$ sets a given width upper-bound to a new value. The function typically takes the current search horizon and the set of reached goal states into account. Often, *newwidthbound* raises the value of the upper-bound by some fixed interval. It is used in cost-updating searches. *(See page 130)*

bound storing functions *updatedepth* and *updatewidth* *updatedepth* : $\mathbb{K} \times \mathbb{K} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \rightarrow \mathbb{K}$ and *updatewidth* : $\mathbb{N} \times \mathbb{N} \times 2^{\mathscr{S}} \times 2^{\mathscr{S}} \rightarrow \mathbb{N}$ are used during a round in an iterative search to calculate the new upper-bounds on-the-fly. They can be used to, for instance, keep track of the minimum encountered cumulated cost still higher than the current upper-bound, or the maximum cumulated cost, making the search more greedy in a sense. Another option found in practice is to have the functions increase the bounds by fixed intervals. *(See page 130)*

bound-updating search A cost-bounded search, which updates its bounds U and/or β upon finishing, e.g. after having encountered a goal state, and then continues searching, using the new bounds. *(See page 130)*

Branch-and-Bound (BnB) search A class of searches which are guaranteed to have at least one optimal solution in each round in the search horizon. Ordinary BnB search is an instance of this class, which updates its weighted depth upper-bound whenever a new goal state is found, and continues the search with this new upper-bound. *(See page 138)*

C

completeness A search is complete if it is ensured that, given a non-empty and reachable set of goal states $\mathscr{G}$, it always returns a trace from a state in $\mathscr{I}$

to a state in $\mathscr{G}$. *(See page 121)*

cost-bounded search A search employing a guiding upper-bound U. *(See page 129)*

cost-optimality A search is cost-optimal if it is ensured that, given a non-empty and reachable set of goal states $\mathscr{G}$, it always returns a trace from a state in $\mathscr{I}$ to a state s in $\mathscr{G}$, with $f(s) = \text{đ}(\mathscr{I}, \mathscr{G})$. *(See page 127)*

D

deadlock avoidance The ability of a heuristic function to predict the locations of (undesired) deadlock states in a state space. *(See page 136)*

duplicate detection function *DuplicateFree* Applies duplicate detection on a set of states S, i.e. it checks for each element in S whether it has been explored earlier in the search. When considering cumulated costs, it is checked whether each element has been explored before with a cumulated cost smaller than or equal to the newly found cumulated cost. *(See page 128)*

F

filtered beam search A two-phase best-first search which applies two different types of beam search in two phases, subsequently, where in the first phase a computationally cheap guiding function is used, and the second phase deals with a more thorough selection among the remaining states. *(See page 142)*

G

generation function *GeneratePath* Given a set of (goal) states, it returns a path, or trace, from $\mathscr{I}$ to one of the elements in the set. The function is typically used for reachability purposes. *(See page 120)*

greedy search In the literature, at least two searches are called greedy search, the first one being a best-first search with $f = h$, h being an estimation function $h : \mathscr{S} \rightarrow \mathbb{K}$, no pruning in the width, and $\beta = 1$, and the second one being a best-first search with $f = g + h$, g being a cumulated cost function $g : \mathscr{S} \rightarrow \mathbb{K}$, pruning in the width, and $\beta = 1$. To avoid confusion, we dub the second one *heuristic nearest neighbour search*. *(See page 137)*

guiding function *f* A function on states, meant to be used to guide a search through a state space. In general, f may depend on the description of a state, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain. The lower $f(s)$ for a given state s, the more promising it is to find a goal state by exploring s. *(See page 121)*

guiding upper-bound $\boldsymbol{U}$ An upper-bound to the guiding function f. Whenever it is applied in a search, the search will never explore states with an f-value greater than the upper-bound. *(See page 128)*

H

heuristic depth-first search A two-phase best-first search with $f_2 = h$, h being an estimation function $h : \mathscr{S} \to \mathbb{K}$, and $\beta = 1$. The first phase selects states in a depth-first search manner. In the second phase, h is applied to explore one of the states selected by the first phase. *(See page 142)*

heuristic nearest neighbour search Referred to by e.g. Zhou and Hansen (2005) as *greedy search*, heuristic nearest neighbour search is a best-first search with $f = g + h$, g being a monotonic cumulated cost function $g : \mathscr{S} \to \mathbb{K}$ and h being an estimation function $h : \mathscr{S} \to \mathbb{K}$, $\beta = 1$, and pruning in the width is applied. *(See page 137)*

heuristic or estimation function $\boldsymbol{h}$ A function $h : \mathscr{S} \to \mathbb{K}$ estimating the weighted distance between a given state and the set of goal states. *(See page 134)*

I

iterative search Whenever a given, non-iterative, search would terminate, its iterative version will start a new iteration by throwing away all generated state space levels, possibly updating the used bounds, and considering $\mathscr{I}$ for exploration. *(See page 130)*

L

local search A search which does not look further ahead in the state space than the next transition(s) to take. *(See page 130)*

M

minimal weighted distance $\boldsymbol{\text{đ}}$ The minimal weighted distance between two sets of states S and S' equals the minimum of the weights of all traces leading from a state in S to a state in S'. *(See page 123)*

monotonicity of cumulated cost functions A cumulated cost function $g : \mathscr{S} \to \mathbb{K}$ is called monotonic iff for all $s, s' \in \mathscr{S}$ with $s \xrightarrow{\ell} s'$ we have $g(s') \geq g(s)$. *(See page 125)*

multi-phase search A search in which each round consists of a number of phases, where each phase is comparable to a round in a usual search. Round i takes the output of round $i-1$ (or the current search horizon if $i = 1$) and outputs an intermediate search horizon $\mathscr{H}_i$. The last phase in a round delivers the final output of the round. *(See page 140)*

N

nearest neighbour search Also known as *gradient descent*, the nearest neighbour search is a local search with $f = g$, g being a monotonic cumulated cost function $g : \mathscr{S} \to \mathbb{K}$, $\beta = 1$, and pruning in the width is applied. *(See page 130)*

P

pruning in the width A search prunes in the width whenever, in each round, it removes the states not selected for exploration from the search horizon. By setting the flag *WidthPruning* to *true*, this is activated. *(See page 130)*

S

select function $\boldsymbol{select_\beta}$ The function $select_\beta : 2^{\mathscr{S}} \to 2^{\mathscr{S}}$ selects a subset of states from a given set of states S, and the selected subset has no more than β elements, i.e. $select_\beta(S) \subseteq S$ and $\mid select_\beta \mid \leq \beta$. How $select_\beta$ makes selections may differ from one search to another. *(See page 121)*

selecting extra states A search using this functionality may also select states which are not the most promising, i.e. states with an f-value greater than the minimal one found in the search horizon. It is typically used to compensate for inaccuracies in the guiding function. In the literature, a best-first search employing the selection of extra states is often called a k-best-first search. By setting the flag *BestStatesOnly* to *false* it is activated. *(See page 134)*

selection width $\boldsymbol{\beta}$ An upper-limit to the number of states which may be selected for exploration in each round of a search algorithm. *(See page 121)*

T

tie-breaking Selecting a subset of states from a set of states comprising of equally promising states, i.e. states with the same f-value. *(See page 130)*

U

uniform-cost search Also known as *lowest-cost-first search* and *Dijkstra's search*, uniform-cost search is a best-first search with $f = g$, g being a monotonic cumulated cost function $g : \mathscr{S} \rightarrow \mathbb{K}$, and no pruning or bounding is performed. *(See page 127)*

W

weighted state space A state space extended with a given function $\mathscr{C} : \mathscr{A} \rightarrow \mathbb{K}$, $\mathbb{K}$ being a cost domain. *(See page 122)*

Z

***Z* search** A class of searches incorporating so-called recursive-weight-computation, e.g. a cumulated cost function. *(See page 138)*

Chapter 7

Model Checking Methods for Scheduling

> *Aan de oevers van de tijd keek ik om me heen. Ik wachtte aan de kant.*
>
> (*Spinvis*)

7.1 Problem Description

IN RECENT YEARS, MODEL CHECKERS have been applied to solving combinatorial optimisation problems, i.e. problems where one of the best combinations of possible values for a given set of variables needs to be found. In particular, scheduling (or planning) problems have been considered, often using a range of available model checkers. In a paper by Niebert et al. (2000), the problem of minimum-time reachability for timed automata is considered. It is shown that this problem can be solved by examining acyclic paths in a forward reachability graph generated on-the-fly from a timed automaton. Based on this, Behrmann et al. (2001a) consider the model checker UPPAAL, describing how to deal with instances of job shop scheduling. The jobshop problem is the most classic scheduling problem in the literature. In its most basic form, we have a finite set M of resources, and a number of jobs $J_1, \ldots, J_n$, which compete in using the resources in a specific order and for a finite number of time units. The problem is to allocate the resources such that the jobs are finished in minimal time. Behrmann et al. (2001a) introduce linearly priced timed automata as an extension of timed automata with prices on both transitions and locations. They consider the minimum-cost reachability problem. An algorithmic solution is offered, based on a combination of branch-and-bound techniques, which can be used for limiting the search space and for quickly finding near-optimal solutions, and a new notion of priced regions. It is shown that using these techniques reduced the explored state space by 90% when compared to a straight-forward breadth-first search. For UPPAAL, e.g. Behrmann et al. (2005) suggest to model each job and resource as a timed automaton. Another technique is to model the problem with a single process, as

Ruys (2003) does with PROMELA. More on these two techniques later. The common approach here is to model the system at hand, such that the resulting state space contains all possibilities to deal with the problem. In such a state space, the problem is interpreted as a reachability problem, where the question is, in a system where costs are associated with transitions, what the minimal necessary cost is to reach a state $s \in \mathscr{G}$, where $\mathscr{G} \subseteq \mathscr{S}$ is a set of successful termination states (i.e. 'good' states where a complete schedule for the given problem has been achieved). A trace providing this minimal cost then represents a schedule for the problem at hand.

A scheduling problem, within this setting, is typically about processing a certain number of entities (for instance, products or jobs, in the case of jobshop scheduling). The processing is usually done by a resource, or combination of resources, which can perform tasks[1] $t_1,\ldots,t_m \in Ta$, provided that the accompanying sets of constraints $C_1,\ldots,$ C_m are met.[2] Furthermore, each task t_i has an execution time $d(t_i)$ associated with it, given by the function $d : Ta \to \mathbb{T}$. In these problems, a certain goal should be reached, usually having completely processed a finite batch of entities. The question asked in scheduling is not mainly *if* this goal can be reached, but *how efficiently* this can be done.

Over the years, many techniques have been developed to deal with this kind of scheduling problem, for instance by Brucker et al. (1994). Certainly it has been shown that model checking can also be applied in this area. One could argue, however, whether model checking can compete here with other methods, the majority of which have been used much longer in this area and often specifically optimised to deal with this kind of problems. For instance, there are countless attempts to deal with jobshop scheduling, and when we apply model checking for this, the feared state space explosion problem arises very quickly.

However, a major strength of most model checkers is the expressibility of their modelling languages. For instance, the language μCRL, for which we describe techniques to deal with scheduling problems in this part of the thesis, is a very expressive language and allows the use of abstract data types, by which most useful data structures can be defined. Model checkers are primarily designed to allow the modelling of complex industrial systems, which can then be functionally verified. This expressibility can justify the use of model checkers for scheduling. In the existing scheduling literature, the majority is either aimed at very specific types of scheduling problems, as for instance job shop scheduling, or an individual case to be scheduled, which usually means that an implemented algorithm to solve the case is directly built into the implementation of the problem. In other words, a general modelling technique is often lacking.

As the main goal of the work in this thesis is to develop techniques that allow the integration of qualitative and quantitative analysis, we want to achieve the possibility

[1] We denote task labels here as coming from a set Ta.

[2] To keep things general, we do not fix these constraints to a specific notation here. Suffice it so say that they can deal with time and data.

to model a system and use that one specification to do both functional analysis and scheduling, if so desired. We observe that in order to achieve this, we need to keep in mind that the techniques for scheduling should be applicable on arbitrary state spaces. In scheduling literature, the search space of a scheduling problem often resembles a highly structured tree, where the leaves represent the termination of a possible solution, and every node in level i of a tree with n levels has exactly $n - i$ outgoing edges. An example, where $n = 3$, is displayed in Figure 7.1. In the figure, goal nodes are depicted as grey nodes. In an arbitrary state space, however, there are cycles present, states can have multiple incoming transitions, and paths may end unsuccessfully (i.e. the system deadlocks). Later on, in Chapter 10, we give some examples of systems which deliver a state space containing all of these elements. Already in this chapter, we deal with these more general search spaces.

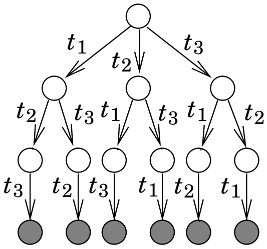

Figure 7.1: Search tree for a single-resource scheduling problem with tasks t_1, t_2, t_3

As we perform scheduling using model checking tools, we are able to deal with complex industrial systems, the specifications of which tend to lead to very big, arbitrary state spaces. We model tasks as transitions, meaning that performing task t_i in an execution appears as $s_i \xrightarrow{t_i} s_{i+1}$ in a state space $\mathscr{M}$, where s_i and s_{i+1} are two states in the trace corresponding with the execution. In state spaces where the traces represent schedules, we can observe the following.

A function *progress*: $\mathscr{S} \rightarrow \mathbb{K}$ can be constructed, which can access the state variables of a state s, using the underlying specification of $\mathscr{M}$ and quantifies the progress made to reaching some predetermined goal, for instance having completely processed a given batch of entities. In general, say we have $c_0, c_{end} \in \mathbb{K}$, $\forall s \in \mathscr{S}.c_0 \leq \mathit{progress}(s) \leq c_{end}$ and $\forall s \in \mathscr{I}.\mathit{progress}(s) = c_0$, in other words, c_0 is the initial (no) progress and c_{end} represents having reached the goal. We do not claim any monotonicity of this function, as in general one can imagine tasks which provide negative progress, leading a schedule further from the goal.

Because of the presence of the *progress* function, we need to refine the description of deadlock states in Chapter 2, where we mentioned that a state s is a deadlock state whenever $en_{\mathscr{M}}(s) = \emptyset$. Now, we need to distinguish deadlock states and successful termination states. We can do this as described in Definition 18.

Definition 18 ((un)successful termination). *A state s is a* successful termination *iff $en_{\mathscr{M}}(s) = \emptyset$ and $progress(s) = c_{end}$. A state s is an* unsuccessful termination *iff $en_{\mathscr{M}}(s) = \emptyset$ and $progress(s) \neq c_{end}$.*

Often, a scheduling problem is modelled such that each goal state is a successful termination state, although, of course, one can imagine goal states which are not termination states. In most cases, therefore, $\mathscr{G}$ and the set of successful termination states coincide. In this context, we associate $\mathscr{B}$ with the set of unsuccessful termination states, i.e. $\mathscr{B} = \{s \in \mathscr{S} \mid en_{\mathscr{M}}(s) = \emptyset\} \setminus \mathscr{G}$.

7.2 Scheduling with Model Checkers

In this section, we discuss some techniques for solving scheduling problems using modelling languages and model checkers. Firstly, we give an impression of modelling scheduling problems with PROMELA (for the model checker SPIN (Holzmann, 2004)) and priced timed automata (for the model checker UPPAAL CORA (Behrmann et al., 2005)). After that, we describe how one can model a large class of scheduling problems using μCRL, also considering problems which require a parallel solution.

The tools mentioned in the chapter were specifically chosen, because firstly, all have been used for solving scheduling problems (e.g. Behrmann et al. (2001a), Behrmann et al. (2005), and Fehnker (1999) report on using UPPAAL CORA, Brinksma and Mader (2000), Ruys (2003), and Ruys and Brinksma (1998) describe how to deal with scheduling problems using SPIN, and Wijs et al. (2005) presented techniques for scheduling with μCRL), and secondly, the techniques for SPIN are based on depth-first search, the techniques for the μCRL toolset and CADP are based on breadth-first search, and UPPAAL CORA incorporates both kinds of searches. These two search techniques are the two most basic search algorithms in model checking. Thirdly, the first two approaches work with different temporal logics. SPIN uses LTL (Pnueli, 1981), which is state-based, meaning that you refer to state variables in its formulas; CADP, however, uses regular alternation-free μ-calculus (Mateescu and Sighireanu, 2003), which is action-based, meaning that you refer to transition labels in its formulas. For both types of logics we show how to deal with schedules. Finally, both SPIN and μCRL deal with discrete time scheduling problems, where costs are natural numbers, while UPPAAL CORA can deal with real-time scheduling problems. We will, however, not deeply go into the syntax and semantics of the modelling languages associated with these tools. Instead, we provide an idea of the approach to modelling scheduling problems, the reason for this being that the focus of this part of the thesis lies on the search algorithms used to solve the problems, and not so much on the input languages to model them. For more information on the input languages, the reader is referred to the papers mentioned earlier on the specific languages and tools.

First of all, in order to model a scheduling problem, we need to model some notion of cost. One can create a specific variable for this and make sure that every time an action associated with a task t_i is fired, the value of this variable is raised by $d(t_i)$. This approach has been carried out using SPIN, μCRL and UPPAAL CORA, and will be used later in Sections 7.3.1, 7.3.2, 7.3.3 and 7.3.5. Another approach in μCRL is described by Wijs et al. (2005), based on the work by Blom et al. (2003), Ioustinova (2004), and Wijs and Fokkink (2005).[3] Here, a special *tick* action is used, which models time progression. This is comparable with relative discrete time (Baeten and Middelburg, 2002): A *tick* action indicates that the system moves to the next time slice. The duration of an execution now equals the number of *tick* actions occurring in this trace. Of course, instead of time, one can also view *tick* more generally as the progression of cost.[4] Note that this closely relates to delay transitions of timed automata, used in both UPPAAL and UPPAAL CORA, as described by e.g. Behrmann et al. (2004). These two approaches allow us to define a *minimal-cost trace* in two ways, as presented by Definitions 19 and 20. Both approaches can be seen as practical ways to identify the (minimal) weighted distance between $\mathscr{I}$ and a state s, as described by Definition 12 earlier on.

Definition 19 (minimal-cost trace with (parameterised) action-based costs). *Given a state space $\mathscr{M}$ and a set of successful termination states $\mathscr{G} \subseteq \mathscr{S}$, we say that there is a trace with total cost c ($c \in \mathbb{K}$) to $\mathscr{G}$ iff there is a trace in $\mathscr{M}$ starting from a starting state $s_0 \in \mathscr{I}$ and reaching a state $s \in \mathscr{G}$, such that the number of (or, in case of parameterised cost actions, the sum of the parameters of)* tick *(or delay) transitions occurring in this trace equals c. We define a trace from $\mathscr{I}$ to $\mathscr{G}$ to be* minimal-cost *if there is no other trace in $\mathscr{M}$ from $\mathscr{I}$ to $\mathscr{G}$ with fewer* tick *(or delay) transitions (or a lower sum of the parameters).*

Definition 20 (minimal-cost trace with state-based costs). *Given a state space $\mathscr{M}$, where each state $s \in \mathscr{S}$ contains a variable cost (of a type representing $\mathbb{K}$) denoted $s.cost$, where $s.cost = 0$ if $s \in \mathscr{I}$ and $s.cost = s'.cost + c'$ if $s \notin \mathscr{I} \wedge \exists \ell \in \mathscr{A}.s' \xrightarrow{\ell} s \wedge \mathscr{C}(\ell) = c'$, and a set of successful termination states $\mathscr{G} \subseteq \mathscr{S}$, we say that there is a trace with total cost c ($c \in \mathbb{K}$) to $\mathscr{G}$ iff there is a trace in $\mathscr{M}$ starting from a starting state $s_0 \in \mathscr{I}$ and reaching a state $s \in \mathscr{G}$, such that $s.cost = c$. We define a trace from $\mathscr{I}$ to $\mathscr{G}$ to be* minimal-cost *if there is no other trace in $\mathscr{M}$ from $\mathscr{I}$ to any $s' \in \mathscr{G}$ with $s'.cost < c$.*

[3] The relevant work by Blom et al. (2003), Ioustinova (2004), and Wijs and Fokkink (2005) is also presented in Chapter 3 of this thesis.

[4] By using *tick* actions to model costs, we actually connect the use of $\mathbb{T}$ from previous chapters with $\mathbb{K}$, as used in this part. Following the guidelines for $\mathbb{T}$ as given in Section 3.3, where its structure is defined as equal to $\mathbb{Z}$, we would obtain discrete timing as a cost measure. Though in the practical examples used in this part we have used these guidelines, in general, one can imagine defining $\mathbb{T}$ in other ways, making the *tick* approach flexible enough to deal with the techniques described in this and the following chapter using any cost domain.

In other words, note that Definitions 19 and 20 practically determine $\mathbb{d}(\mathscr{I},\mathscr{G})$, where costs are modelled using the special action *tick* or delay transitions, either parameterised or not[5], and using a variable *cost*, respectively. Using these definitions, we can formulate a scheduling problem as a reachability problem: finding an optimal schedule to perform a batch of tasks successfully can also be seen as finding a minimal-cost trace to a state in $\mathscr{G}$, in other words a state representing success, in a state space containing all possible schedules as traces.

The general structure of a specification of a scheduling problem in PROMELA can be described as consisting of a process, which is an alternative composition of all tasks t_i, each followed sequentially by an update of the *cost* variable, in order to indicate the execution time (or cost) of each task. On top of that, the tasks t_i can only be executed if the accompanying conditions C_i are met, written in the specification as conditions for the actions representing the tasks, and, once executed, the task has an effect on the current state of the process (comparable with the function *progress*). Therefore, this model can execute all available tasks as long as the constraints are satisfied. The choices which tasks to execute and when are non-deterministic; there are no built-in priorities.

Ruys (2003) describes how to model scheduling problems using PROMELA. There, however, the more general situation, in which unsuccessful termination states, i.e. bad states $\mathscr{B}$, are present in the state space, is not considered. In this section, we present the search algorithms extended with bad state detection and avoidance. For this purpose, on the modelling side, we raise a flag *finished* whenever successful termination has been reached.

In UPPAAL CORA, priced timed automata are used to specify a scheduling problem. Here, in general, multiple processes, which synchronise with each other using channels, together express the problem. Recall that a scheduling problem often consists of a set of passive objects, called resources, and a set of active objects, called jobs (Behrmann et al., 2005). A resource process is usually a two-location cyclic process with one local clock. The locations indicate that the resource is either waiting or operating. The resource starts operating whenever a job synchronises over a **start** channel, resetting the clock. The moment a certain use time is reached, the resource moves back to the waiting location and initiates synchronisation over a channel **done**.

A job process is an acyclic sequence of locations, where the initial state represents the start of the job, and the final location, which we call **Finished** here for comparison reasons, indicates that the job is complete. The locations in between represent the acquisition and release of resources. A resource is acquired by achieving synchronisation over the correct **start** channel and setting the use time. It remains in the same location until synchronisation is performed over the **done** channel. The reachability problem can now be formulated in UPPAAL CORA as the question whether a state can be reached in which all the jobs are in the location **Finished**.

[5] Chapter 5 describes a method to use parameterised *tick* actions in μCRL specifications.

Moving our attention to μCRL, we can create a specification of a scheduling problem as described in this section, in ways very similar to both the PROMELA and the timed automata approach. Like the approach described by Ruys (2003), we can often model a scheduling problem in just one process, which contains all possible task alternatives t_i as actions, each guarded by a condition C_i, and each having a cost $d(t_i)$ associated with it; in μCRL, we model $d(t_i)$ in an action-based manner, using, as mentioned earlier, the special action label *tick*. The structure of a scheduling process actually resembles the one of an LPE (see Chapter 2). Therefore, based on the LPE form, we present the general form of a μCRL scheduling process in Definition 21.

Definition 21 (scheduling linear process equation). *A* scheduling linear process equation *is a guarded recursive equation of the following form:*

$$
\begin{aligned}
&X(d:\mathbb{D}) = \\
&\quad \sum_{i \in I} \sum_{e_i \in \mathbb{D}_i} a_i(f_i(d,e_i)) \cdot tick(w_i(d,e_i)) \cdot X_i(g_i(d,e_i)) \lhd\ h_i(d,e_i) \rhd \delta\ + \\
&\quad finished \cdot \checkmark\ \lhd progress(d) = c_{end} \rhd \delta
\end{aligned}
$$

where I is a finite index set, $\mathbb{D}, \mathbb{D}_i, \mathbb{D}_{a_i}, \mathbb{K} \in \mathcal{D}$, $a_i \in \mathcal{A}$, $a_i : \mathbb{D}_{a_i}$, $tick : \mathbb{K}$, $f_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{D}_{a_i}$, $w_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{K}$, $g_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{D}$ and $h_i : \mathbb{D} \times \mathbb{D}_i \to \mathbb{B}$.

Of course, in this equation, actions $a_i(f_i(d,e_i))$ correspond with tasks t_i, conditions $h_i(d,e_i)$ relate to the scheduling conditions C_i, and function w_i assigns the costs to the tasks. In relation to weighted state spaces, $w_i(d,e_i) = c$ iff $\mathscr{C}(a_i) = c$. Note that the use of *tick* here is the parameterised form as explained in Chapter 5. An alternative is to place in each line of the specification $t_i(d,e_i)$ non-parameterised *tick* actions in sequence. It should be stressed that in a state space resulting from such a specification, transitions with a label different from *tick* essentially have no cost associated with them. Furthermore, we use a special action called *finished* to indicate successful termination (i.e. in $\mathscr{M}$, $\forall s \in \mathscr{S}.(\exists s' \in \mathscr{S}.s' \xrightarrow{finished} s \iff s \in \mathscr{G})$). This is mainly necessary to express reachability using the μ-calculus later on. The condition for the successful termination alternative is a direct translation of the *progress* check as explained earlier.

With μCRL, it is moreover possible to specify a scheduling problem in a way very similar to the technique described by e.g. Behrmann et al. (2005) for timed automata. When, for instance, applied on jobshop problem instances, as described earlier in this section, the technique involves mapping each resource and job to an individual process. The feasibility of this technique first of all hinges on synchronisation over the channels **start** and **done**, which can be specified with μCRL using appropriate communication rules and the encapsulation operator ∂_H with $start, done \in H$. Second of all, synchronisation of timing is essential, i.e. all processes in the specification must agree on the progression of time. Since this is achievable with μCRL by using e.g. the special operator $|\{tick\}|$ (see Section 3.3) or with μCRL^{tick} using $|\!\top\!|$ (see Chapter 5) to put processes in parallel, we can directly adopt the same recipe to construct the resource and job processes.

In planning literature, sequential plans (schedules) for a problem are distinguished from parallel schedules. A sequential schedule directly corresponds with a sequence of actions leading to a goal state, while in a parallel schedule independent actions may be fired simultaneously and dependent actions should be fired sequentially. To point out the structural difference between the two types of schedules, we define the notions of sequential and parallel schedule in Definitions 22 and 23, based on the ones provided by Edelkamp (2003), using the terminology of Section 7.1.

Definition 22 (Sequential schedule). *Given the state space of a scheduling problem $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$ and a set of goal states $\mathscr{G} \subseteq \mathscr{S}$, a* sequential schedule $\omega_s = (t_1, \ldots, t_n)$ *is an ordered sequence of tasks $t_i \in Ta$ such that the corresponding actions in $\mathscr{M}$ lead from a state $s \in \mathscr{I}$ to a state $s' \in \mathscr{G}$, i.e. there exist states $s_0, \ldots, s_n \in \mathscr{S}$ such that $s_0 \in \mathscr{I}$, $s_n \in \mathscr{G}$, and $s_i \xrightarrow{t_{i+1}} s_{i+1}$, for $0 \leq i < n$.*

Definition 23 (Parallel schedule). *Given the state space of a scheduling problem $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$ and a set of goal states $\mathscr{G} \subseteq \mathscr{S}$, a* parallel schedule $\omega_c = ((t_1, u_1), \ldots, (t_n, u_n))$ *is a schedule of tasks $t_i \in Ta$ such that the corresponding actions in $\mathscr{M}$ lead from a state $s \in \mathscr{I}$ to a state $s' \in \mathscr{G}$, where task t_i is executed at absolute time u_i.*

Note that Definition 23 does not state how a parallel schedule should be represented in a state space, unlike Definition 22 for a sequential schedule, which directly corresponds with a sequence of transitions. Next, we will reason about parallel schedules, and illustrate how such schedules can be represented in a state space using the μCRL toolset, even if they cannot be represented by a sequential schedule, given that for each task $t_i \in Ta$ we have $d(t_i) > 0$.

When trying to construct a parallel schedule, one approach often considered is to detect a successful trace in the state space, and converting this trace into a partially ordered schedule (Pednault, 1986; Regnier and Fade, 1991; Veloso et al., 1991). The partial order then gives rise to the possibilities to execute tasks in parallel. The conversion involves analysing the pre- and post-conditions of the tasks, and determining for each task t_i which other tasks need to be completed before it can be executed, and which other tasks it needs to precede in execution. Necessarily, this also leads to insight into which tasks are independent of each other. Note that when using this approach, it is assumed that every possible parallel schedule for a problem can be obtained by relaxing the order in a sequential one. Later, we will deal with a situation where this is not the case.

Next, we illustrate how a scheduling problem which calls for a parallel schedule, assuming that it can be represented by a sequential schedule, can be modelled using μCRL in such a way that a trace in the resulting state space directly relates to a partially ordered schedule, which moreover can directly be interpreted as a parallel schedule without any conflicts arising. The key for this is the presence of timing information in the traces. First, we assume that the μCRL specification of such a scheduling problem is structured conform the description given earlier to model a scheduling problem by means of a number of processes representing jobs and resources in parallel composition, i.e. the approach when using UPPAAL CORA. More specifically, we can assume that the individual process descriptions are in a form very similar to the one of Definition 21. Each job needs to perform a finite number of actions.

We illustrate two possible kinds of dependencies between actions. On the one hand, the execution of a task t_1 may enable the execution of another task t_2. Here, we call this a *supporting* dependency. Either t_1 and t_2 are part of the same job, in which case they are ordered inside one process description, or they stem from different jobs. In the latter case, the dependency is in fact enforced by some resource process, as dependencies between jobs are established via some resource.

The other kind of dependency could be called a *competing* dependency, where the execution of a task t_1 (temporarily) disables the execution of another task t_2. Again, the tasks may either be part of the same job, in which case this dependency is established by the conditions present in the process description, or they stem from different jobs, which means that the dependency is (again) established via the resource processes. If we have two job processes J_1 and J_2, and one resource process R, which J_1 and J_2 both wish to use, then either J_1 may synchronise with R, thereby disabling the possibility for J_2 to synchronise with R, or vice versa.

Given a trace from a state space, we show how this trace can be interpreted as a parallel schedule. If t_2 is dependent on t_1, then the first possibility is that t_1 and t_2 are part of the same job. In this case, clearly, by the form of the job description, t_1 and t_2 appear in different time units in the trace, i.e. the task executions are sequentially separated from each other by at least one *tick* action. Also, if the tasks stem from different jobs, the (shared) resource, which upon being used by a job delays until the execution is finished, ensures that t_1 and t_2 appear in different time units in the trace. Conversely, if two tasks t_1 and t_2 appear in the same time unit, i.e. they are not sequentially separated by a *tick* action, then they are not part of the same job, nor are they dependent on each other. In other words, they can safely by executed in parallel. This relates to the notion of independent actions for partial order reduction (Peled et al., 1996).

Now, let us move to a more general setting, in which it is possible that a parallel schedule cannot be represented by a sequential one. Such a situation may arise if we, for instance, loosen the mutual exclusion behaviour of the resources. Consider the case where we have a shuttle-bus which accepts passengers to be brought to an airport. The bus is fully automatic; it opens its doors exactly once, and after it has detected that a passenger has entered it, it closes the doors and leaves. If we have two people who wish to be transported, and it is our goal that these two people both reach the airport, then no sequential schedule is successful; as soon as one person has entered the bus, it heads for the airport, leaving the other person behind.

The key to the solution is the fact that both persons may enter the bus simultaneously. If we wish to model this system, we must therefore allow jobs to obtain a resource in true concurrency. One way to do this is in fact used for the Clinical Chemical Analyser (see Section 10.5). There, multiple cranks may perform operations on a table, adding, removing, and mixing fluids. The scheduling problem is specified by means of a single process description, which not only lists all the individual operations in alternative composition, but also all the possible combinations represented by additional action labels, which express the operations being performed concurrently.

If the specification consists of multiple processes in parallel composition, another way to model true concurrency is by employing the communication mechanism. It is possible, in a way similar to the synchronisation of *tick* and *tock* actions (see Section 3.3) and the modelling of shared variables in μCRL (see Section 3.4.13), to allow the actions of multiple jobs and resources to synchronise with each other, resulting in a single transition in the state space. If we, moreover, do *not* encapsulate the individual actions, then we also still allow sequential executions. In the shuttle-bus example, an action of a person process would, first of all, need to synchronise with an action of the shuttle-bus process, which is the usual approach, but besides that, in addition, we would allow actions of other person processes to synchronise simultaneously with that same shuttle-bus action. As *tick* actions synchronise as well, the delays associated with the scheduling actions will be performed concurrently. Related to this, using a single process to describe all possible behaviour, also concurrent behaviour, in fact allows the possibility that the cost of performing some actions concurrently is not the same as the sum of the costs of the individual actions. These situations actually occur in the case of the Clinical Chemical Analyser.

One benefit of incorporating true concurrency in traces is that we can search for parallel schedules. Another benefit is that if we employ some kind of heuristics to search for promising areas of the state space, we need not be concerned about this presence of concurrency at all. If concurrency is represented by some partial order of actions, then it means that a state space search encounters intermediate states. For example, if tasks t_1 and t_2 are to be executed simultaneously, and we have a trace containing $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2$, then we need to be aware of the fact that s_1 does not represent a 'real' situation, since in fact the situation represented by s_2 follows immediately from the one represented by s_0. This observation should be anticipated in the heuristics. However, if we have instead a transition $s_0 \xrightarrow{t_{1,2}} s_2$, where $t_{1,2}$ is the action representing the simultaneous execution of t_1 and t_2, then this anticipation can be absent. Moreover, in an action-based heuristic, we can differentiate between executing actions simultaneously or not.[6]

[6] In the planning setting, e.g. Haslum and Geffner (2000) explain how to automatically construct admissible heuristic functions to help in obtaining sequential or parallel plans.

If we enforce concurrency as much as possible, i.e. we encapsulate the individual actions, then a third benefit is that with true concurrency, we avoid interleavings of actions in the state space, which may lead to a drastic reduction of the average branching factor of the state space if there is a lot of concurrency present in the problem.[7]

Having created a specification, it is possible, using the appropriate toolset, to generate a state space from it. This state space incorporates all possible behaviour of the system described by the specification. Given that there exist successful traces in the state space, i.e. at least one successful termination state is reachable, somewhere in this state space there is at least one minimal-cost trace to a successful finish. Given Definition 19, we use the *finished* action to detect states $s \in \mathscr{G}$, in order to be able to capture in the μ-calculus a minimal-time trace to a successful termination. In UPPAAL CORA, as previously mentioned, a state $s \in \mathscr{G}$ is identified as a state where all the job processes are in the **Finished** location. When using (state-based) LTL formulas in practice, however, it appears we are not able to incorporate the detection of successful termination in the formulas themselves. When using SPIN following the approach of Ruys (2003), where the formula is used to bound the search through each trace, incorporating this detection will result in less efficient bounding behaviour, or even the removal of it. The detection can sometimes, however, be performed by other means, while in other cases it can be avoided altogether, at the cost of an increase of the state space size. This will be explained in the following sections. It should be pointed out that, although in this section we presented some techniques to specify scheduling problems, the algorithms to search the resulting state spaces, presented in the next section and the following chapter, are mostly designed with arbitrary state spaces in mind, i.e. practically any specification can be analysed with these searches; they are by no means limited to the analysis of a certain class of constructed specifications.

7.3 Finding Optimal Schedules

In this and the subsequent section, we describe the search algorithms used for scheduling in SPIN and UPPAAL CORA in an abstract manner, and introduce some techniques that allow the analysis of scheduling problems with μCRL, either in the state space generating μCRL toolset or the model checker CADP. Here, we consider μCRL as the input language of CADP, although of course LOTOS can also be used.

[7] Hoffmann and Geffner (2003) point out that in parallel planning, the usually high average branching factor of state spaces is a major problem. They discuss how to deal with this by proposing a range of branching schemes. Enforcing the concurrency as much as possible is important here, as otherwise we may introduce a lot of additional transitions. E.g. Haslum and Geffner (2000) mention that such an approach, i.e. where maximal concurrency is not enforced, may lead to an exponential increase of the average branching factor.

7.3.1 Iterative Searching

The most straightforward technique to search for solutions to a scheduling problem is to iteratively search the state space using a set of formulas, written in a temporal logic, such as LTL or μ-calculus.

First of all, using the specification of a scheduling problem and the matching toolset, a full state space needs to be generated. Next, one needs to formulate, using a temporal logic, the property ϕ that every trace in $\mathscr{M}$ has a cost greater than or equal to $U \in \mathbb{K}$ before reaching successful termination. Here, U is chosen as an upper-bound to the actual minimal cost of reaching successful termination. Given that U is an upper-bound, the model checker will be able to find a counter-example to the property and provide a new, smaller, possibly minimal cost $U' \in \mathbb{K} < U$. Again, now with U', the property is checked, possibly leading to another counter-example and a new value $U'' \in \mathbb{K} < U'$. This process is repeated until the model checker finds that the property holds, at which point the currently minimal cost is the minimal cost we are looking for and the counter-example given in the previous iteration is one of the minimal-cost traces.

The practical application of this technique differs from toolset to toolset. Here we briefly describe how it works in both SPIN, as presented by Ruys (2003), and CADP.

In SPIN, when writing LTL formulas, one can refer to state variables in a PROMELA specification. In this case, it allows us to use a variable in the specification that is used to keep track of the total cost up to each state.[8] In the formula we can now express the property as $\phi = \Diamond(cost \geq U)$, where $\Diamond$ expresses eventuality. As thus, ϕ expresses that eventually, the value of *cost* will be greater than or equal to U. Since SPIN searches through state spaces in a depth-first manner, it traverses a trace until it either finds that the property holds (i.e. it finds a state where $(cost \geq U)$), after which it can continue the search in another trace by employing back-tracking, or it concludes that the property does not hold in a trace and a counter-example is produced, after which, in the case of scheduling, the search also continues. This potentially goes on until the complete state space has been searched. In cases, however, where unsuccessful termination states are present, i.e. $\mathscr{B} \neq \emptyset$, we do not want a counter-example each time the property is violated, since we also have the requirement that a trace ends in successful termination. As mentioned before, including this requirement in the property leads to less efficient bounding, so we need to incorporate that in another way. By extending unsuccessful traces infinitely, which can be done in the specification by always allowing cost increasing steps once a state $s \in \mathscr{B}$ has been detected, we make sure that ϕ holds for these traces and thereby avoid unwanted counter-examples. As mentioned previously, this, however, leads to a larger state space.

In CADP, one can use regular, alternation-free μ-calculus to express properties. This temporal logic, however, does not allow referring to state variables, it is purely action-

[8] Note that encoding the total cost as a value in each state tends to lead to infinite state spaces if the original specification (without a cost variable) contains cycles.

based. For that reason, in our setting, we need to count the *tick* labels in each trace, in order to determine its cost. In the μ-calculus formula, we are able to differentiate between successful and unsuccessful termination by referring to the *finished* action[9]:

$$\phi = [\neg\texttt{tick}^*.((\mathrm{T} \mid \epsilon).(\neg\texttt{tick}^*))^{U-1}.\texttt{finished}]\,\mathrm{F}$$

Since CADP searches state spaces in a breadth-first manner, on average it has to explore a lot more states, compared to SPIN, before it is potentially able to find a counter-example, since it will consider all possible traces at the same time, therefore only reaching $\mathscr{G}$ at a later stage.[10]

This technique works, but is highly inefficient, and therefore quickly becomes unusable for bigger problem instances. The main reason for this is that the entire state space needs to be generated and searched multiple times, both when property checking can be performed on-the-fly and when it needs to be done after generation. The searching takes up a number of iterations, each time worst-case going over all the states in the state space. On a practical note one can say that a depth-first search works in general more efficiently here than a breadth-first search. The next technique improves on the number of iterations at the searching stage, and the fact that its efficiency depends less on the search order.

7.3.2 Displaying List of Labels

This technique, described by Wijs et al. (2005), is mainly focussed on CADP, since it is an approach that happens to be practically possible in that toolset, in contrast to the other tools. First of all, in the specification, we use the variable *cost* to represent the progression of cost, and extend the *finished* action with a parameter, providing the current total cost to the outside world when the action is fired.

Again, the complete state space of the specification needs to be generated. Then, in the toolset, there has to be a possibility to display all the action labels occurring in the state space. Besides labels representing the execution of tasks t_i, the list will contain a number of *finished* labels, each accompanied with a fixed parameter value c. Now, the minimal c value, let us call it c' here, needs to be obtained from this list, which directly provides the minimal cost value in the state space. Next, one has to search the state space only once using a μ-calculus formula indicating that the action *finished*(c') can never be reached, and the model checker will provide an optimal schedule as a counter-example.

[9] Here it is checked that all traces leading to *finished* do not contain $U-1$ or fewer *tick* transitions. The $(\mathrm{T} \mid \epsilon)$ expression accepts at most one action (including *tick*). Finally, the $\mathtt{A}^n$ notation is not a valid μ-calculus expression, but a shorthand for A written n times in sequence.

[10] In practical cases, SPIN seems to search much more states before finding a schedule, compared to μCRL and CADP. An example of this can be seen in Section 10.3. These numbers can, however, not be compared, since the tools apparently count states in completely different ways.

Similar to the former technique, this approach also becomes impractical when the problem instances get bigger. Instead of a varying number of iterations, however, we can now claim that an optimal solution is found after two iterations; the first one going over the whole state space, constructing the list of action labels, the second one using the formula to obtain a schedule. The search order is less important here than for the iterative search technique, since in the first iteration the complete state space needs to be searched anyway, after which there is only one iteration left.

7.3.3 Depth-first Branch-and-Bound

In the next technique, we use the most common form of the Branch-and-Bound approach (see Kumar (1992) and Section 6.6). The idea of this is that while searching, we can stop exploring a trace once it is clear that all traces in the subtree below the current state lead to total-costs greater than the one of the best solution found so far. It needs to be stressed that Branch-and-Bound is not a heuristic method, the pruning of parts of the state space only happens when it is known that it can safely be done. This in contrast with techniques like beam search, already described in Chapter 6, and more extensively studied later in Chapters 8 and 9, which can therefore only claim finding near-optimal solutions.

The idea of this technique is based on the iterative search, and is also described by Ruys (2003). Instead of iteratively checking and updating an LTL formula, the formula is adapted on-the-fly while searching the state space. Once a smaller cost c is found, the new value is placed in the formula and the search continues where the model checker stopped, now using the updated formula. The benefit of using this technique is that the state space only needs to be searched once, and on average not completely, since parts of the state space can be pruned away. Still, though, it can take a lot of time and memory to find a solution. In SPIN 4.0, this technique can practically be used, since it allows the use of C primitives. Hidden variables can be used to contain temporary data while searching. An update section in the specification, written in C, is fired each time a counter-example is found, which writes the counter-example to a file and updates the (hidden) minimal-cost variable, thereby changing the property to check. We observe that this technique also allows the detection of unsuccessful termination, since we can extend the guard of the update section, such that it is only fired when the property does not hold *and* the trace ends successfully. For a PROMELA specification example suitable for depth first BnB in SPIN, which incorporates this additional check, see Appendix B.

As with iterative searching, the depth-first search performed by SPIN is well-suited for this technique. See Section 10.3 for an example where this iterative search is used. Algorithms 16 and 17 show a presentation of this approach using the DMC notation of Chapter 6, which expresses the practical technique of checking for (cost-wise) better traces using an LTL property and detecting successful termination in two consecutive steps. We use the notation $\phi(U)$ to express the property that all traces have a weight greater than or equal to the upper bound U. This is written as a safety property $\phi(U) = \Diamond(cost \geq U)$ (where $\Diamond$ is the eventuality operator) which depends on the given value $U \in \mathbb{K}$. Related to this property, we can identify two types of sets of states $S_{\downarrow,<U}, S_{\geq U} \in \mathscr{S}$. First of all, we say that a state $s \in S_{\downarrow,<U}$ iff s is a termination state and $s.cost < U$. For a state $s \in S_{\downarrow,<U}$, we can locally determine, i.e. we do not need to look ahead to states reachable from s (since there are no states reachable from s), that $\phi(U)$ does not hold for s. In fact, all traces leading from an $s \in \mathscr{I}$ to an $s' \in S_{\downarrow,<U}$ constitute all the counter-examples to $\phi(U)$. In Algorithm 17, the sets $S_{\downarrow,<U}$ are used to check whether a counter-example has been found which is better than the last one found. In addition to this, we say that a state $s \in S_{\geq U}$ iff $s.cost \geq U$. For a state $s \in S_{\geq U}$, we can locally determine that $\phi(U)$ holds for s. With the sets $S_{\geq U}$, bounding on U can be performed, since for every state s where $s.cost \geq U$, we know that $\phi(U)$ holds for all states in traces from an $s' \in \mathscr{I}$ through s, hence we do not need to check the property for any $s'' \in \mathscr{S}$ such that $s \rightarrow^* s''$.

Algorithm 16 Depth-first BnB search for scheduling

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{I})$, set of goal states $\mathscr{G}$, depth upper-bound $D$, cost upper-
   bound $U$
Ensure: if exists, a minimal-cost trace to a goal state is returned
  $Closed \leftarrow \emptyset$
  $\mathscr{L}_0 \leftarrow \mathscr{I} \setminus S_{\geq U}$
  while $\mathscr{L}_0 \neq \emptyset$ do
    $\hat{\mathscr{L}}_0 \leftarrow select_\beta(\mathscr{L}_0)$ (by increasing $g(s)$)
    $\mathscr{L}_0 \leftarrow \mathscr{L}_0 \setminus \hat{\mathscr{L}}_0$
    $Closed \leftarrow Closed \cup \hat{\mathscr{L}}_0$
    $\langle U, Closed \rangle \leftarrow sched\_dfs(\hat{\mathscr{L}}_0, Closed, \mathscr{G}, D, U)$
  end while
  return true
```

If a state is in $S_{\downarrow,<U}$, we need to check whether the goal has been reached, e.g. for a problem as described in Section 7.1, that $progress(s) = c_{end}$, or, in an action-based setting, that *finished* can be performed. This can be done by determining whether the state is also in $\mathscr{G}$. In fact, we can consider $S_{\downarrow,<U} \setminus \mathscr{G}$ to be a set of bad states $\mathscr{B}$, i.e. unsuccessful termination states.

Notice that if $select_\beta$ always selects up to one state, i.e. $\beta = 1$, Algorithms 16 and 17 express an explicit depth-first search. Finally, the successor states are explored in order of weight. This is called *node ordering*, and helps in finding an optimal solution quickly, as reported by Zhang (1999).

One of the pitfalls of this technique is the presence in $\mathscr{M}$ of cycles in which the cumulated cost is not increasing. It is possible to deal with these by providing a cut-off depth, indicating the maximum depth at which the algorithm is allowed to search. In

Algorithm 17 Procedure sched_dfs($\mathscr{L}_i, Closed, \mathscr{G}, D, U$)

```
if $\mathscr{L}_i \neq \emptyset$ then
  $\mathscr{L}_{i+1} \leftarrow \emptyset$
  if $\mathscr{L}_i \cap S_{\downarrow,<U} \cap \mathscr{G} \neq \emptyset$ then
    $U \leftarrow min\{g(s) \mid s \in \mathscr{L}_i \cap S_{\downarrow,<U} \cap \mathscr{G}\}$
    output $GeneratePath(\mathscr{L}_i \cap S_{\downarrow,<U} \cap \mathscr{G})$
  end if
  if $i+1 < D$ then
    for all $s \in \mathscr{L}_i$ do
      $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \cup\ nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
    end for
    $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \setminus (Closed \cup S_{\geq U})$
    while $\mathscr{L}_{i+1} \neq \emptyset$ do
      $\hat{\mathscr{L}}_{i+1} \leftarrow select_{\beta}(\mathscr{L}_{i+1})$ (by increasing $g(s)$)
      $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \setminus \hat{\mathscr{L}}_{i+1}$
      $Closed \leftarrow Closed \cup \hat{\mathscr{L}}_{i+1}$
      $\langle U, Closed \rangle \leftarrow sched\_dfs(\hat{\mathscr{L}}_{i+1}, Closed, \mathscr{G}, D, U)$
    end while
  end if
end if
return $\langle U, Closed \rangle$
```

practice, SPIN always performs bounded depth-first search. It is used in Algorithm 17 as D, while the current depth is represented by i. Note the important difference from U, which is an upper-bound on the cost. Also, note that here, in contrast with the use of U in cost-bounded directed model checking algorithms (see Section 6.5 and ongoing), the bounding is done with U using the sets $S_{\geq U}$, which more closely relates to the practical technique used in SPIN. In fact, the main reason for showing Algorithms 16 and 17, even though bounded depth-first search fits into the searches presented in Chapter 6, is to focus on the practical approach of SPIN concerning this search.

In Algorithm 17, the duplicate detection is straightforward, without checking if a state has been encountered before with a greater cumulated cost or not. This extra check can be avoided here, since the variable *cost*, by which way we practically keep track of cumulated costs g, is part of the state itself, thereby each state has a unique cumulated cost throughout the whole search. Compared to associating cumulated costs dynamically with states during the search, i.e. the costs are calculated while exploring and not part of the states themselves, the latter technique has in general the advantage, however, of producing smaller state spaces; this is due to the fact that when costs are part of the states, for any states $s, s' \in \mathscr{S}$, if $g(s) \neq g(s')$, then necessarily $s \neq s'$, while this is not the case when g-values are calculated on-the-fly.

7.3.4 g-Synchronised or Minimal-cost Search

The main disadvantage of the iterative searching method is the fact that the state space needs to be generated completely, and needs to be searched multiple times, each time focussing on a new possible minimal cost, because in a standard search, the costs of the different traces cannot be compared in one iteration. As already seen, one way of improving this is the use of Branch-and-Bound as described in the previous section. This, however, is not always applicable, since it requires the possibility of updating the temporal logic formula while searching. Another approach is to manipulate the search order in such a way, that the intermediate cumulated costs of all traces can be compared on-the-fly. Approaches like this, however, require that the model checker is extended with new techniques.

The μCRL toolset has been extended with new generation algorithms. One of these is called minimal-cost search, also referred to as g-synchronised search.[11] Here, *tick* transitions are used to represent the progress of cost, and other transitions are in fact without cost. Besides the levels $\mathscr{L}_i$, there is a set $\mathscr{W}$. For all s in the current $\mathscr{L}_i$ to be expanded, a successor s' ends up in $\mathscr{W}$ if $s \xrightarrow{tick} s'$, and in $\mathscr{L}_{i+1}$ otherwise. The $\mathscr{L}_i$ set is continuously used to select new states, until $\mathscr{L}_i = \emptyset$, at which point the search moves to $\mathscr{L}_{i+1}$. If this level is empty at the start, all states in $\mathscr{W}$ are moved to $\mathscr{L}_{i+1}$ and the searching continues, in other words, the algorithm starts considering states with a greater cumulated cost. Basically, g-synchronised search equals uniform-cost search from Section 6.4, where the cost is modelled using additional actions.

Algorithm 18 presents this technique. Whether a state s is in $\mathscr{G}$ is deduced here by determining whether it is reached via a *finished* transition or not.

Searching with this ordering principle means we know that we find a minimal-cost solution to the problem the first time we find a solution, and can therefore stop immediately. Technically, for all levels $\mathscr{L}_i \subseteq \mathscr{S}$, we have for all states $s \in \mathscr{L}_i$ that $g(s) = \text{d}(\mathscr{I}, \mathscr{L}_{i-1})\}$, where $\text{d}(\mathscr{I}, \mathscr{L}_{i-1})$ is calculated in an action-based way (see Definition 19). Furthermore, as in the case of BnB for SPIN in Section 7.3.3, duplicate detection does not have to incorporate the checking of cumulated costs, since by its very nature, uniform-cost search encounters a state the moment it has found a minimal-cost trace to the state. In practice it shows that this technique pays off; the bigger the problem instance, the higher the percentage of the state space that can be skipped entirely (for examples, see Chapter 10).

Algorithm 18 is designed for unparameterised *tick* actions, as presented in Section 3.3. We can, however, extend this algorithm to parameterised *tick*(t) actions, as used in Chapter 5. For that, we do not just need one set $\mathscr{W}$, but sets $\mathscr{W}_1, \ldots, \mathscr{W}_\mathrm{c}$, where c is the largest time jump the system can make in one step. Then, when checking tran-

[11] g-Synchronised search is an instance of the more general G-synchronised search. Besides synchronising on the cumulated cost, one could also, for instance, imagine synchronising on a heuristic function. In Chapter 8, we will return to G-synchronised searching.

Algorithm 18 Minimal-cost search for $\mathscr{M}$ with *tick*-encoded costs

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{I})$
Ensure: If exists, a minimal-cost trace to a goal state is returned
  $\mathscr{W} \leftarrow \emptyset$
  $i \leftarrow 0$
  $\mathscr{L}_i \leftarrow \mathscr{I}$
  $\mathscr{L}_{i+1} \leftarrow \emptyset$
  while $\mathscr{W} \neq \emptyset \vee \mathscr{L}_i \neq \emptyset$ do
    if $\mathscr{L}_i = \emptyset$ then
      $\mathscr{L}_i \leftarrow \mathscr{W}$
      $\mathscr{W} \leftarrow \emptyset$
    end if
    for all $s \in \mathscr{L}_i$ do
      for all $s \xrightarrow{\ell} s' \in en_{\mathscr{M}}(s)$ do
        if $\ell = \mathit{finished}$ then
          return $\mathit{GeneratePath}(\{s'\})$
        else if $\ell = \mathit{tick}$ then
          $\mathscr{W} \leftarrow \mathscr{W} \cup \{s'\}$
        else
          $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \cup \{s'\}$
        end if
      end for
    end for
    $i \leftarrow i + 1$
    $\mathscr{L}_i \leftarrow \mathscr{L}_i \setminus \bigcup_{j=0}^{i-1} \mathscr{L}_j$
    $\mathscr{W} \leftarrow \mathscr{W} \setminus \bigcup_{j=0}^{i-1} \mathscr{L}_j$
  end while
  return $\mathit{false}$
```

sitions, each transition with action label $tick(t)$ is placed in set $\mathscr{W}_t$. Whenever $\mathscr{L}_i = \emptyset$, we set $\mathscr{L}_i$ to $\mathscr{W}_j$ with j the smallest index such that $\mathscr{W}_j \neq \emptyset$. With $\mathscr{W}_j$ made empty, the remaining sets $\mathscr{W}_k$ with $j < k \leq \mathrm{c}$ are shifted to the sets $\mathscr{W}_{k-j}$. This implements that the search performs a time jump of j time (or cost) units. Currently, though, this extension is not yet available in the toolset.

7.3.5 Real-time Branch-and-Bound Scheduling

UPPAAL CORA has a number of searches built-in, which can help in solving scheduling problems. Uniform-cost search, identified in UPPAAL CORA as best-first search, is available to find cost-optimal schedules. Most other available searches are not cost-optimal; we return to these later on. Fehnker (2002) describes an algorithm to perform (ordinary) BnB on priced timed automata, comparable with depth-first BnB in SPIN, setting a time upper-bound and using the global clock for comparison, or keeping track of the minimal total cost found thus far in a variable named *cost*.

7.4 Finding Near-optimal Schedules

Up to now we described techniques which guarantee finding an optimal solution. To be able to guarantee this, the complete state space $\mathscr{M}$ needs to be searched, or bounding needs to be limited to situations where a cost upper-bound has been reached. In practice however, $\mathscr{M}$ can be very large. One could consider not keeping the expanded states in memory and writing them directly to disk, in cases where the state space of a scheduling problem resembles a tree.[12] But even then, although memory is not an issue anymore, searching the entire state space can take a very long time. In cases where a near-optimal solution practically suffices, one can prevent exhaustive searching. In this section we explain some of the techniques available for such an approach.

7.4.1 Breadth-first and Depth-first Search

As remarked by Fehnker (2002), regular breadth-first and depth-first search can be used to return solutions to a weighted problem, but they rarely return an optimal solution. As he has experienced with UPPAAL, breadth-first search quickly runs out of memory, and depth-first search actually returned the worst possible solution when analysing the Sidmar Steel Plant case study. The problem here lies in the fact that both breadth-first and depth-first search do not take cumulated costs into account.

[12] It should be noted that there are techniques known which allow writing states directly to disk even when the state space does not resemble a tree, e.g. Hammer and Weber (2006) describe a technique where duplicate detection is performed using a so-called Bloom filter. This filter is inquired whenever it needs to be determined whether a state has already been written to disk earlier in the search, or not.

7.4.2 Nearest Neighbour Heuristic or Gradient Descent

For some problems, e.g. the Traveling Salesman Problem (TSP), (see e.g. Lawler et al. (1985)), the so-called nearest neighbour heuristic or Gradient Descent (see Section 6.5), can provide acceptable solutions. This search selects for every state, which in the case of TSP represents a city, the nearest successor state for further exploration. Since the other successors are discarded, it can only promise to find near-optimal solutions. This process is repeated until a complete schedule is found. In SPIN, this technique has been used by Ruys (2003) by carefully specifying the processes in a specific order and placing the different statements in **if**-clauses. This suffices to ensure that always the nearest successor is selected. The search, however, only appears useful for problems where a local view on states, i.e. for each state only considering the next transition to take, suffices. The search seems to be particularly ineffective if the state space contains unsuccessful traces, which initially appear promising.

7.4.3 Beam Search

We briefly mention beam search here, since it should be placed in this category of searches for scheduling problems. It is, however, the main focus of Chapter 8, where this search, which is originally designed to be applied on highly structured trees (see Figure 7.1), is adapted and further extended to be effective for searching state spaces in general. The main concept of beam search, and its place within the whole spectrum of searches, is provided in Section 6.8. It is a technique that we have implemented in the μCRL toolset, and has been applied on a number of case studies, as presented in Chapter 10.

7.4.4 Near-optimal Real-time Scheduling

Fehnker (2002) describes a number of approaches to find near-optimal solutions to scheduling problems, specific for UPPAAL. First of all, UPPAAL allows random depth-first search. Whenever the state space does not contain cycles (and all traces are finite), a random search is guaranteed to finish at some point. Running it several times and calculating the average result is then an option; of course, the average result will often be much greater than the optimal solution.

Another option is to declare certain actions urgent, i.e. they should be fired whenever enabled before time passes. This technique, however, may also postpone useful delays, thereby missing the optimal solutions.

Besides regular breadth-first and (random) depth-first search, UPPAAL CORA provides *smallest heuristic first* search, which is in fact greedy search (Russell and Norvig, 1995), where the heuristics used can, for instance, be an estimation of the remaining cost. Then there is *best depth-first* search, which operates like nearest-neighbour. It explores states in increasing order of the cumulated cost, but limits its search horizon in a depth-first manner, therefore it looks for local minima, making it not cost-optimal, unless one would exhaustively search the whole state space in this manner (possibly using BnB). If an estimation function is provided, UPPAAL CORA automatically incorporates it into its uniform-cost and nearest-neighbour searches, making them comparable with A^* and heuristic nearest neighbour search, respectively.

7.5 Possible Extensions

In this chapter, an overview was provided of different techniques, which can be used to solve the most common scheduling problems. These techniques are available in the model checking tools SPIN, CADP together with the μCRL toolset, and UPPAAL CORA. At this point, one can consider a number of possible extensions to these techniques, thereby dealing with other, related, scheduling issues. Most of what is discussed here, however, remains to be investigated.

First of all, one can consider the construction of an online scheduler for a specific scheduling problem. In other words, instead of creating a schedule, given a batch of entities to process, one would create a scheduler able to deal with any possible input sequence of entities. Behrmann et al. (2005) refer to this as finding an optimal *infinite* trace in $\mathscr{M}$. In cases where there is only one kind of input entity, or several kinds arriving in a fixed sequence, a cycle of actions (or edges in priced timed automata) would suffice. However, when there are several kinds of entities to process, arriving in an arbitrary order (which could be imagined, for instance, in the case of the Clinical Chemical Analyser of Chapter 10), one would possibly have to look for a flower-shaped structure, with, on average, the most entities processed per cost unit. Here, a flower-shaped structure is made up of a set of states $S \subseteq \mathscr{S}$, which is the set of initial states for this structure, and a number of cycles moving from, and returning to S, depending on the number of possible combinations of types of entities arriving as input. The problem seems to be related to finding an optimal playing strategy for a game, which is the subject of e.g. the search algorithm SSS^* (Pearl, 1984). This search tries to find a minimal-cost subtree of a search tree. The subtree must contain all possible reactions to all possible actions of the opponent, and the cost of the subtree is defined as the highest cost among the minimal-costs of goal states in the subtree. As the SSS^* algorithm does not consider cycles though, it will need to be extended to be applicable on arbitrary state spaces.

Another possible extension is the possibility to deal with *multiple* cost variables, in situations where, for instance, not only time, but also money, energy, etc. play a role. For priced timed automata, one already has moved in this direction, as reported by Behrmann et al. (2005) and Larsen and Rasmussen (2005). The interesting phenomenon here, is that a problem with multiple costs might have different solutions, depending on the priority given to the different costs. For instance, if we put the highest emphasis on the minimisation of time, we might get a very different answer compared to when we minimise on money. As mentioned in Chapter 6, particular instances of multi-phase best-first search, like multi-phase uniform-cost search, and synchronised beam search, which is presented in Chapter 8, might be able to deal with multiple costs. In the first case, we could subsequently order the states based on the different types of cost, and in the second case we may consider synchronising on one type of cost, but using another in the evaluation function. In both searches, the first type would then be our main minimisation concern, while the second type would be kept as low as possible, given the circumstances.

Chapter 8

Directed Model Checking With Beam Search

> *Your thinking [...] involves, therefore, something over and beyond the mere inspection of a four-dimensional associational structure. It involves interpretation of that structure.*
>
> (*J.W. Dunne*)

8.1 Introduction

OVER THE YEARS, A NUMBER OF techniques have emerged to prune, while generating, parts of the state space that are not (or do not seem) promising given the task at hand. Some of these techniques, such as partial order reduction (POR) algorithms (for instance, see Clarke et al. (1999)), guarantee that no essential information is lost after pruning. On the other hand, this chapter focuses mainly on heuristic pruning methods which heavily reduce the generation time and memory consumption, but may prune away essential parts of the state space. The idea is that a user-supplied heuristic function guides the generation algorithm such that ideally only relevant parts of the state space are actually explored. This is, in fact, at odds with the core idea of model checking when studying qualitative properties of systems, i.e. to exhaustively search the complete state space to find any corner case bug. However, heuristic pruning techniques can very well target performance analysis problems, as approximate answers are usually sufficient when using model checking in quantitative analyses of systems (for more on this type of analysis, see, e.g., Brinksma et al. (2001)). A comparison between the two types of pruning can thus not fairly be made, since different problems allow different techniques. However, such heuristics are not restricted to quantitative properties, as, for instance, Torabi Dashti and Wijs (2007) show, how a known POR algorithm for security properties can be cast to *priority beam search*, which is a version of beam search explained in the following sections.

In this chapter, we investigate how *beam search* can be integrated into the state space generation setting. Beam search is a heuristic method for combinatorial optimisation problems, which has extensively been studied in artificial intelligence and operations research, among others by Lowerre (1976), Rubin (1978), Fox (1983), Si Ow and Smith (1988), Sabuncuoglu and Bayiz (1999), and Della Croce and T'kindt (2002). There, beam search is similar to breadth-first search, as it progresses level by level through a highly structured search tree containing all possible solutions to a problem, but it does not explore all the encountered nodes. At each level, all the nodes are evaluated using a heuristic cost (or priority) function, but only a fixed number of them is selected for further examination. This aggressive pruning heavily decreases the generation time, but may in general miss essential parts of the tree for the problem at hand, since wrong decisions can be made while pruning. Therefore, beam search has so far been mainly used in searching trees with a high density of goal nodes. Scheduling problems, for instance, have been perfect targets for using beam search, as their goal is to optimally schedule a certain number of jobs and resources, while near-optimal schedules, which densely populate the tree, are in practice good enough.

The idea of using beam search in state space generation is an attempt towards integrating functional analysis, to which state spaces are usually subjected, and quantitative analysis. Since model checkers, like SPIN (Holzmann, 2004), UPPAAL (Behrmann et al., 2004), and the μCRL toolset (Blom et al., 2001), which generate these state spaces, usually have highly expressive input languages, beam search for state spaces can be applied in a more general framework than its more traditional version for search trees. Applying beam search to search state spaces tightly relates to directed model checking (DMC) (Edelkamp et al., 2004), and guided model checking (for (priced) timed automata) (Behrmann et al., 2001a), where heuristics are used to guide the state space exploration when checking properties. Here, we mainly aim for checking quantitative properties, where approximate results are often sufficient. This singles out our work from the existing DMC techniques.

In the current chapter, we motivate and thoroughly discuss adapting the beam search techniques to deal with arbitrary structures of state spaces. By this, we stretch the idea of DMC to the field of quantitative analysis.

Next, we extend the classic beam search in two directions. First, we propose *flexible* beam search, which, broadly speaking, does not stick to a fixed number of states to be selected at each search level. This partially mitigates the problem of determining this exact fixed number in advance. Second, we introduce the notion of *synchronised* beam search, practically an instance of multi-phase best-first search (from Section 6.10), which aims at separating the heuristic pruning phase from the underlying exploration strategy. Possible combinations of these variants create a spectrum of search algorithms that, as will be described, encompasses some known search techniques such as A^* search and partial order reduction algorithms.

Remarkably, POR can be seen as a sort of careful pruning in which certain functional properties of the model are preserved. In related work, the framework also allowed us to extend this algorithm to deal with branching security protocols (Fokkink et al., 2007). This confirms that the use of beam search, applied with suitable heuristic functions, goes beyond checking quantitative properties.

Most of the upcoming beam searches have been implemented in the μCRL state space generation toolset. Experimental results on comparing this toolset with SPIN in scheduling are eventually presented in Chapter 10.

The classic beam search for highly structured trees is described in Section 8.2; here, we temporarily deviate from the model checking setting, and present beam search in its 'traditional' setting. Section 8.3 deals with the adaptation of two existing variants of beam search to the state space generation setting. There we also propose our extensions to the beam search algorithms. After that we focus on the implementation of some of these adapted and extended beam search algorithms in the μCRL toolset. Related issues such as memory management and selecting heuristic functions are also discussed. In Section 8.7, we describe how our framework encompasses A^* search and a partial order reduction algorithm for security protocols. Section 8.8 presents our related work and Section 8.9 concludes the chapter.

8.2 Beam Search

Beam search (see, e.g., Si Ow and Morton (1988), Bisiani (1992), and Pinedo (1995)) is a heuristic search algorithm for combinatorial optimisation problems, which was originally used in the artificial intelligence community by Lowerre (1976) for speech recognition, and by Rubin (1978) for image understanding. Later on, this technique has been applied to scheduling problems, for example by Fox (1983), Si Ow and Smith (1988), and Sabuncuoglu and Bayiz (1999) in systems designed for jobshop environments (for an explanation of this kind of problem, see Section 7.1). Since then, new variants of beam search, such as filtered beam search (Pinedo, 1995; Si Ow and Morton, 1988, 1989) and recovery beam search (Della Croce and T'kindt, 2002; Valente and Alves, 2005c), have been introduced.

Beam search is similar to breadth-first search as it progresses level by level. At each level of the search tree, it uses a heuristic evaluation function to estimate the promise of encountered nodes[1], while the goal is to find a path from the initial state (initial node of the tree) to a leaf node that possesses the minimal evaluation value among all the leaves. At each level, only the β most promising nodes are selected for further examination and the other nodes are permanently discarded. The *beam width* parameter β is fixed to a value before searching starts. Because of this aggressive pruning, the

[1] In this section, we use the most common terminology when referring to beam search, i.e. we reason about nodes and edges, as opposed to states and transitions. This emphasises that we adapt the beam search techniques to a different setting.

generation time is linear to the maximum search depth, and is thus heavily decreased. However, since wrong decisions can be made while pruning, beam search is neither complete (or cost-optimal), i.e. is not guaranteed to find a solution when there is one, nor optimal, i.e. does not guarantee finding an optimal solution. To limit the possibility of wrong decisions, one can increase the beam width, at the cost of increasing the required computational effort and memory use.

The original definition of beam search allows any kind of guiding function to be used. Using the terminology of Chapter 6, it only demands that pruning in the width is performed, and extra states are selected; as Fox (1983) put it: “[Beam search] builds a highly pruned search tree of labelling alternatives which resembles a beam”. In this chapter, however, we focus on two types of evaluation functions, which have traditionally been used often for beam search, as, for instance, reported by Si Ow and Morton (1988) and Valente and Alves (2005b): *priority* evaluation functions and *total-cost* evaluation functions, which lead to the *priority* and *detailed* beam search variants, respectively. In priority beam search, at each node a priority evaluation function calculates a priority for each successor node, and the algorithm selects based on those priorities. At the root of the search tree, up to β most promising successors (i.e. those with the highest priorities) are selected, while in each subsequent level only one successor with the highest priority is selected per examined node. Figure 8.1 describes the basic idea of traditional priority beam search. There, s_0 is the root of the search tree, and all leaves are assumed to be located at the same level.

1. Set B = $\emptyset$, C = $\emptyset$
   - Branch s_0 to generate its children
   - Perform priority evaluation of each child node
   - Select min{β,number of children} best child nodes, add them to B
2. For each node in B:
   - Branch node to generate its children
   - Perform priority evaluation of each child node
   - Select best child node, add it to C
3. Set B = C; Set C = $\emptyset$
4. Stopping Condition: if all nodes in B are leaf, select node with lowest total-cost and stop, otherwise go to step 2.

Figure 8.1: Priority beam search for search trees (Valente and Alves, 2005a)

In detailed beam search, at each node n, the (state-based) evaluation function $f(n) = g(n) + h(n)$ calculates an estimate of the total-cost of the best schedule that can be found, continuing from the partial schedule represented by node n. At each level, up to β most promising nodes (i.e. those with the lowest total-cost values) are selected, regardless of who their parent nodes are. If there are more than β nodes that receive the best evaluation value, a selection is made based on other criteria, e.g. the order of encountering the nodes (see Section 8.3.4 for other possibilities). Clearly, when $\beta = \infty$, detailed and priority beam search behave as exhaustive breadth-first search. Figure 8.2 represents traditional detailed beam search.

```
1. Set C = $\emptyset$, B = $\{s_0\}$
2. For each node in B:
     • Branch node to generate its children
     • Perform detailed evaluation of each child node $n$ (i.e. calculate $f(n) = g(n) + h(n)$)
     • Select min{$\beta$,number of children} best child nodes, add them to C
3. Set B = $\emptyset$; Select min{$\beta$, |C|} best nodes in C, add them to B; Set C = $\emptyset$
4. Stopping Condition: if all nodes in B are leaf, select node with lowest total-cost and stop, otherwise
   go to step 2.
```

Figure 8.2: Detailed beam search for search trees (Valente and Alves, 2005a)

In comparison, priority evaluation functions have a local view of the problem, since they only consider the next job to be scheduled, while total-cost evaluation functions have a more global view, taking the complete schedule into account and comparing different branches of the tree. The intuition behind priority evaluation functions is that one cannot simply compare priorities of jobs, i.e. nodes, which are connected to different executions, because the suitability of a job depends on what came before in the execution (for the more general state space setting, this is explained in more detail in Section 6.11). The children of the root node, however, can fairly be compared as they share the same (empty) execution history. A total cost evaluation function, on the other hand, allows comparison of nodes from different executions, as it shows the progress each execution is making.

In general, total-cost evaluation functions are computationally more expensive than priority evaluation functions, but often provide more accurate heuristics because of their global view (Si Ow and Morton, 1988).

Figure 8.3 shows the application of a detailed beam search on a search tree. The grey nodes are selected using the evaluation function, while the obscured ones are nodes that would have been encountered, had their parents been selected. Typically, this is a detailed beam search as opposed to a priority beam search. First of all, in a detailed beam search, states with the lowest f-values are selected, while in a priority beam search, priorities must be high in order to qualify for selection. Second, in a priority beam search, up to β transitions from the root of the tree are followed, after which in each subsequent level of the tree one outgoing transition with the highest priority is selected per examined node. In a detailed beam search, however, at each level up to β nodes are selected to continue, regardless of what their parent nodes are, therefore it could be the case, as in level 4 of Figure 8.3, that some nodes have multiple

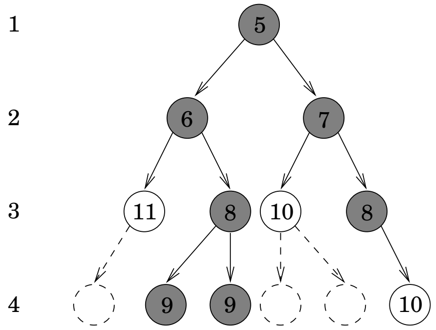

Figure 8.3: Example of detailed beam search with $\beta = 2$ in a search tree

selected children, while others have none. The reason for this is that one cannot simply compare priorities of actions which are connected to different executions, due to the fact that selection of an action depends on what came before in the execution. A total-cost evaluation function does allow comparison of nodes from different executions though, since using such a function allows us to see the progress each execution is making.

8.3 Adapting Beam Search for State Space Generation

8.3.1 Motivation

Beam search is typically applied on highly structured search trees, like the one shown in Figure 7.1, which contain all possible orderings of a given number of jobs, e.g. see Oechsner and Rose (2005), and Valente and Alves (2005c). Such a search tree starts with n jobs to be scheduled, which means that the root of the tree has n outgoing transitions. Every node has exactly $n - k$ outgoing transitions, where k is the level in the tree where the node appears. State spaces, however, supposedly contain information on all possible behaviours of a system. Therefore, they may contain cycles or confluence of traces (i.e. states have multiple incoming transitions), and have more complex structures than the well-structured search trees usually subjected to beam search. This necessitates modifying the beam search techniques to deal with arbitrary state spaces. Moreover, the beam search algorithms search for a particular state in the search space, while in (and after) generating state spaces one might desire to study a property beyond simple reachability (see Section 8.7 on partial order reduction as an instance of extended beam search). We therefore extend beam search to a state space *generation* setting, as opposed to its traditional setting that focuses only on *searching*.

See Section 8.5 for possible optimisations when restricting beam search to verify reachability properties. This, along with the necessary machinery for handling cycles, raises memory management issues in beam search, as we will see in Section 8.5.

First, we revisit priority and detailed beam search for state space generation. Next, we propose two variants of beam search which have, in our case studies, proved essential for handling large state spaces. Flexible beam search mitigates the problem of determining a sufficiently large beam width, while synchronised beam search separates the pruning phase from the exploration strategy.

Figure 8.4 shows the spectrum of the variants that are described in the following sections. There, DBS and PBS correspond, respectively, to the detailed and priority beam searches, adapted to deal with arbitrary state spaces (Sections 8.3.2 and 8.3.3). The F and S prefixes refer to the flexible and synchronised beam search variants (Sections 8.3.4 and 8.3.5).

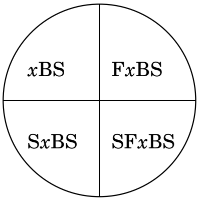

Figure 8.4: Beam search spectrum, $x \in \{\mathrm{D}, \mathrm{P}\}$

8.3.2 Priority Beam Search for State Space Generation

Next, we motivate and describe the changes that we have made to the traditional priority beam search to deal with state space generation.

Priority beam search is shown in Algorithm 19. The user-supplied function $prio : \mathscr{A} \to \mathbb{Z}$ provides the priority of actions, as opposed to states (see Section 6.11).[2]

We motivate this deviation from the traditional notion of priority beam search by noting that *jobs* in the scheduling terminology correspond more naturally with *actions* in a state space when specified for a model checker. Moreover, since priority beam search focuses on generating an approximate state space rather than looking for a particular goal, no total-cost function is in general needed (and provided). The set *Buffer* temporarily keeps seemingly promising transitions. The function $prio_{min} : 2^{\mathscr{T}} \to \mathbb{Z}$ returns the lowest priority of the actions of a given set of transitions. We define $prio_{min}(\emptyset) = -\infty$. The function $getprio_{min} : 2^{\mathscr{T}} \setminus \emptyset \to \mathscr{T}$, given a set T of transitions,

[2] See Section 6.11 for a discussion about this kind of function. Note that as the domain of priorities is here identified as $\mathbb{Z}$, priorities are totally ordered.

Algorithm 19 Priority beam search for state spaces

```
Require: $\mathcal{M}$=($\mathcal{S}$,$\mathcal{A}$,$\mathcal{T}$,$\mathcal{I}$), widening factor $\alpha$, stabilisation level $l$, priority function
    $prio : \mathcal{A} \rightarrow \mathbb{Z}$, set of goal states $\mathcal{G}$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathcal{L}_i \leftarrow \mathcal{I}$
  $Buffer \leftarrow \emptyset$
  $limit := \alpha$
  while $\mathcal{L}_i \neq \emptyset$ do
    $\mathcal{L}_{i+1} \leftarrow \emptyset$
    if $\mathcal{L}_i \cap \mathcal{G} \neq \emptyset$ then
      return $GeneratePath(\mathcal{L}_i \cap \mathcal{G})$
    end if
    for all $s \in \mathcal{L}_i$ do
      for all $s \xrightarrow{\ell} s' \in en_{\mathcal{M}}(s)$ do
        if $prio(\ell) > prio_{min}(Buffer)$ then
          if $|Buffer| = limit$ then
            $Buffer \leftarrow Buffer \setminus \{getprio_{min}(Buffer)\}$
          end if
          $Buffer \leftarrow Buffer \cup \{s \xrightarrow{\ell} s'\}$
        end if
      end for
      $\mathcal{L}_{i+1} \leftarrow \mathcal{L}_{i+1} \cup nxt_{\mathcal{M}}(s, Buffer)$
      $Buffer \leftarrow \emptyset$
    end for
    $i \leftarrow i + 1$
    $\mathcal{L}_i \leftarrow \mathcal{L}_i \setminus \bigcup_{j=0}^{i-1} \mathcal{L}_j$
    if $i = l$ then
      $limit := 1$
    end if
  end while
  return $true$
```

returns one of the transitions in T labelled by an action with priority $prio_{min}(T)$. Note that the stopping condition of the traditional priority beam search algorithm of Section 8.2 is represented here by the condition $\mathscr{L}_i \neq \emptyset$ of the **while** loop, which does not assume that all leaves occur in the same level (this, moreover, avoids cycles). The algorithm terminates when it has explored all the states in its beam.

In priority beam search, originally, up to β children of the root are selected. The resulting beam of width β is then maintained by sprouting only one child per node in subsequent levels. This works for trees such as the one shown in Figure 7.1, where the root has more outgoing transitions than any other node in the tree. In state spaces, however, the root has typically considerably fewer outgoing transitions than the average branching factor of the state space. Fixing the beam width at such an early stage is therefore not reasonable.

Selecting all transitions at each level until β or more transitions are found in a single level would be an option. However, if this number drastically exceeds β, it would not be clear which transitions should be pruned away. To mitigate this problem, instead of β, Algorithm 19 is provided with the pair (α, l), where $\alpha, l \in \mathbb{N}$ and $\alpha^l = \beta$. We call α the *widening factor* and l the *stabilisation level*. The idea is that the algorithm uses the *prio* function to prune non-promising states from the very first level, but in two phases: before reaching β states in a single level it considers per state the most promising α transitions for further expansion, but after that (i.e. once it has reached level l), it sticks to the one child per node rule. Here, the assumption is made that in the first l levels of the state space, each state has at least α outgoing transitions. In practice, this is not such a strong assumption, considering that the kind of problems for which beam search is suitable typically produces state spaces which resemble trees that expand quickly.

8.3.3 Detailed Beam Search for State Space Generation

The original idea of detailed beam search does not need to change much to fit into the state space generation setting, except for handling cycles. When exploring a cyclic state space, to guarantee the termination of the algorithm, it is necessary to store the set of expanded states (in the sets $\mathscr{L}_i$ in, for example, Algorithm 8), to allow detecting duplicates to avoid exploring a state more than once. However, if a state is reached via a path with a lower cost, the state has to be re-examined. This is because the total-cost of each state depends on the cost to reach that state from $\mathscr{I}$. Thus a state (and its successors) may more competently qualify for further explorations if it is reached via a lower cost path. This is a known issue when dealing with weighted state spaces (Section 6.3), and reappears in the upcoming text.

As a side note, the average running time of the detailed beam search algorithm of Section 8.2 can be reduced if the order of exploration and evaluation is reversed. Intuitively, instead of first expanding the nodes of the current level and then evaluating the children and selecting the β most promising of them to constitute the next level (c.f. the algorithm of Section 8.2), we first evaluate the states of the current level, select the β most promising states among them and then expand them, to constitute the next level. When performed successively, these two orders are identical. However, since the number of nodes to be evaluated is a priori known in each level, evaluation of the states of a level containing no more than β states can altogether be avoided.[3]

Besides that, to reduce the space complexity of the traditional detailed beam search algorithm of Section 8.2, while evaluating, only the β most promising states can be kept in a set and the rest can be discarded (note that β^2 states are stored in the algorithm of Section 8.2). This optimisation, of course, does not depend on the order of evaluation and exploration. In the broader context of DMC, beam search applies pruning in the width and selection of extra states (Chapter 6), and the beam width β is comparable with the selection width β in cost-bounded best-first search (Algorithm 11),

Algorithm 20 shows detailed beam search after the mentioned optimisations. It can be applied on a weighted state space $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$. The total-cost evaluation function is called $f : \mathscr{S} \rightarrow \mathbb{K}$. This function is decomposed into $f(s) = g(s) + h(s)$, in other words, as explained in Chapter 6, it incorporates a cumulated cost part and a heuristic part. The $g(s)$ function represents the cumulated cost taken to reach s from $\mathscr{I}$, which is defined as $g(s) = g(s') + c$ if $s' \xrightarrow{\ell} s$ and $\mathscr{C}(\ell) = c$. The set of (action label,weight) pairs $\mathscr{C}$ is user-supplied. The weights can, e.g., denote the time needed to perform different jobs in a scheduling problem. These weights are fixed before searching starts. In practice, often, the weights range over non-negative numbers. In these cases, the values of g never decrease along a path, i.e. $s \rightarrow^* s' \implies g(s') \geq g(s)$; in other words, g is monotonic (Definition 14). Since the generation of successor states incorporates calculating the new cumulated costs for them, we redefine $nxt_{\mathscr{M}}(s, T)$, as is done for Algorithm 9 in Section 6.3, as $nxt_{\mathscr{M}}(s, T) = \{\langle s', s.g + c\rangle \mid s' \in \mathscr{S} \wedge \exists \ell \in \mathscr{A}.\, (s \xrightarrow{\ell} s' \in T \wedge \mathscr{C}(\ell) = c)\}$.

The user-supplied heuristic function $h(s)$ estimates the cost it would take to efficiently complete the schedule continuing from s. Similar to g, the total-cost function f is called *monotonic* iff $s \rightarrow^* s' \implies f(s) \leq f(s')$.

The function $getf_{max} : 2^{\mathscr{S}} \setminus \emptyset \rightarrow \mathscr{S}$, given a set of states, returns one of the states that has the highest f-value. It thus computes $f(s) = g(s) + h(s)$ for each member of the set (it can of course be optimised for consecutive calls to the same set). The level sets $\mathscr{L}_i$ contain pairs of states and corresponding g-values, i.e. $\langle s, s.g\rangle$.

Note that a state will be revisited only if it is reached via a path with a lower cost than the cumulated cost assigned to it.

[3] Actually, the evaluation of the heuristic estimation part, which is computationally the most expensive phase, is the part that is avoided. See Algorithm 20 for details.

Algorithm 20 Detailed beam search for state spaces

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, heuristic function $h : \mathscr{S} \to \mathbb{K}$, beam width $\beta$, set of goal states $\mathscr{G}$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathscr{L}_i \leftarrow \{\langle s, 0 \rangle \mid s \in \mathscr{I}\}$
  while $\mathscr{L}_i \neq \emptyset$ do
    $\mathscr{L}_{i+1} \leftarrow \emptyset$
    while $|\mathscr{L}_i| > \beta$ do
      $\mathscr{L}_i \leftarrow \mathscr{L}_i \setminus \{\langle s, s.g \rangle \in \mathscr{L}_i \mid s = \mathit{getf}_{max}(\mathscr{L}_i)\}$
    end while
    if $\{s \mid \langle s, s.g \rangle \in \mathscr{L}_i\} \cap \mathscr{G} \neq \emptyset$ then
      return $\mathit{GeneratePath}(\{s \mid \langle s, s.g \rangle \in \mathscr{L}_i\} \cap \mathscr{G})$
    end if
    for all $\langle s, s.g \rangle \in \mathscr{L}_i$ do
      $\mathscr{L}_{i+1} \leftarrow \mathscr{L}_{i+1} \cup \mathit{nxt}_{\mathscr{M}}(s, \mathit{en}_{\mathscr{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathscr{L}_i \leftarrow \{\langle s, s.g \rangle \in \mathscr{L}_i \mid \neg\exists g' \leq s.g \in \mathbb{K}.\langle s, g' \rangle \in \bigcup_{j=0}^{i-1} \mathscr{L}_j \wedge \neg\exists g' < s.g \in \mathbb{K}.\langle s, g' \rangle \in \mathscr{L}_i\}$
  end while
  return $\mathit{true}$
```

8.3.4 Flexible Beam Search

A major issue that still remains unaddressed in the beam search adaptations of Sections 8.3.2 and 8.3.3 is how among equally competent candidates, e.g. having the same f values, pruning should be carried out.

Actions in state spaces can have several parameters. The same action can thus appear multiple times as an outgoing transition of a given state, each time having different parameter values, possibly leading to equally competent states. This potentially leads to situations where, during selection, a large number of transitions or states have equal evaluations (for some examples, see Chapter 10). In such cases, a selection has to be made among equally competent candidates if they happen to be (one of) the most promising transitions or among the β-best states. These selections are beyond the influence of the evaluation (or priority) function and can undesirably make the algorithm non-deterministic. Hence, we propose two variants of beam search that we call *flexible detailed* and *flexible priority* beam search, in which the beam width can change during state space generation.

In flexible detailed beam search, at each level, up to β most promising states are selected, plus any other state which is as competent as the worst member of these β states. This achieves closure on the worst (i.e. highest) total-cost value being selected. Similarly, in flexible priority beam search, in the first l levels (see Section 8.3.2), at each state, up to α most promising outgoing transitions are selected, plus any transition which has the same priority as the least competent member of these α transitions. At the $l + 1^{th}$ level and onwards, at each state, all the transitions with the same priority as the most promising transition of that particular state are selected (i.e. as if $\alpha = 1$). In other words, in flexible beam searches, tie-breaking is avoided, by making the beam dynamic in size. Note that in flexible priority beam search, in contrast to flexible detailed beam search, if the beam width is stretched, it cannot be readjusted to the intended β.

The benefit of this approach is that there are no selection criteria other than the evaluation function used. This not only leads to more insight in the effectiveness of the function, but in practice it may also mean that smaller beam widths can be used, compared to non-flexible beam search (see, for instance, the results in Chapter 10). The drawback is that the memory requirement is no longer linear in the maximum search depth, since β is only a guideline for the beam width.

8.3.5 G-Synchronised Beam Search

As is described in Section 8.2, the classic beam search algorithms were tailored for the breadth-first exploration strategy. Next, we explain a more general way, where beam search can be used with any best-first exploration strategy. Broadly speaking, we separate the exploration strategy from the pruning phase, where the exploration may be guided with a (possibly different) heuristic function. This is particularly useful when checking reachability properties on-the-fly.

Using the terminology of Chapter 6, we inductively describe G-*synchronised* x-beam search as a two-phase best-first search, where G is the function that guides the exploration and x can be either *detailed*, *priority*, *flexible detailed* or *flexible priority*. Let $\mathscr{H}$ be the current search horizon. In the exploration phase of round i, we need to determine the set of states to be explored $\mathscr{L}_i \subseteq \mathscr{H}$. We do this by first determining an intermediate search horizon $\mathscr{H}'$ by employing the guiding function G as follows: $\mathscr{H}' = \{s \in \mathscr{H} \mid \forall s' \in \mathscr{H}.G(s) \leq G(s')\}$. Subsequently, the pruning phase of x-beam search is applied on $\mathscr{H}'$, leading to $\mathscr{L}_i \subseteq \mathscr{H}'$. According to the pruning phase (which can possibly employ an evaluation function different from G)[4], some of the states in $\mathscr{H}'$ are selected, constituting the set $\mathscr{L}_i$. Finally, the successors of all the states in $\mathscr{L}_i$ are determined and added to $\mathscr{H}$. The next round starts with the search horizon $\mathscr{H} \setminus \mathscr{H}'$ and needs to determine $\mathscr{L}_{i+1}$. Since this technique distinguishes an exploration phase and a pruning phase, it can perfectly combine most exploration strategies (from different best-first searches) with all the variants of beam search introduced earlier.

Modular implementation of synchronised beam search variants can thus be conceived: the first phase takes care of the order in which states need to be considered for pruning and exploration, and the second phase performs the actual pruning and selection. Such a two-phase approach resembles filtered beam search, described by Si Ow and Morton (1988), where classic priority beam search is applied before classic detailed beam search takes place. In Section 6.10, we described a general algorithm, encompassing these two types of searches, called multi-phase best-first search, which can also deal with more than two phases per round.

Using any constant function as G in synchronised detailed beam search clearly results in beam search with breadth-first exploration strategy. Algorithm 21 shows this technique in detail.

To mention a practically interesting candidate for G, we temporarily return to the application of finding schedules for a given problem. This means that we wish to find a path of minimal cost that leads to a particular action or state in a state space. If for every action ℓ, $prio(\ell) = 1$, this problem corresponds to finding the minimal length trace when verifying a reachability property. Recall that the total-cost function in detailed beam search can be decomposed into $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of the trace leading from $\mathscr{I}$ to s. If $G(s) = g(s)$ in G-synchronised detailed beam search, once a goal state (or a complete schedule) is found, searching can safely terminate. This is because in a goal state s, $f(s) = g(s)$, and since the algorithm always follows paths with minimal g (remember that g is monotonic), state s is reached before another state s' iff $g(s) \leq g(s')$. Note that here no state is re-opened, because states with minimal g are taken first and thus a state can be reached again only via paths with greater costs (c.f.

[4] Using different functions for guiding exploration and pruning in principle allows dealing with multi-priced optimisation problems, c.f. Behrmann et al. (2005).

Section 8.3.3).

Both g-synchronised detailed beam search and g-synchronised priority beam search have been used in solving timed scheduling problems, the results of which are reported in Chapter 10, where minimal-time traces to a particular action label are searched for. One can imagine these searches as two-phase best-first searches with a minimal-cost search (Algorithm 18) in the first phase, and a greedy search (best-first search with $f(s) = h(s)$) and a priority beam search in the second phase, respectively. The same pruning algorithm can be used to search for other kinds of traces, such as a shortest trace or a shortest minimal-time trace.

Algorithm 21 G-Synchronised detailed beam search

```
Require: $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$, exploration function $G$, heuristic function $h : \mathcal{S} \to \mathbb{K}$,
  beam width $\beta$, set of goal states $\mathcal{G}$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathcal{H} \leftarrow \{\langle s, 0\rangle \mid s \in \mathcal{I}\}$
  while $\mathcal{H} \neq \emptyset$ do
    $\mathcal{H}' \leftarrow \{\langle s, s.g\rangle \in \mathcal{H} \mid \forall \langle s', s'.g \in \mathcal{H}\rangle . G(s) \leq G(s')\}$
    $\mathcal{H} \leftarrow \mathcal{H} \setminus \mathcal{H}'$
    $\mathcal{L}_i \leftarrow \mathcal{H}'$
    while $|\mathcal{L}_i| > \beta$ do
      $\mathcal{L}_i \leftarrow \mathcal{L}_i \setminus \{\langle s, s.g\rangle \in \mathcal{L}_i \mid s = getf_{max}(\mathcal{L}_i)\}$
    end while
    if $\{s \mid \langle s, s.g\rangle \in \mathcal{L}_i\} \cap \mathcal{G} \neq \emptyset$ then
      return $GeneratePath(\{s \mid \langle s, s.g\rangle \in \mathcal{L}_i\} \cap \mathcal{G})$
    end if
    for all $\langle s, s.g\rangle \in \mathcal{L}_i$ do
      $\mathcal{H} \leftarrow \mathcal{H} \cup nxt_{\mathcal{M}}(s, en_{\mathcal{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathcal{H} \leftarrow \{\langle s, s.g\rangle \in \mathcal{H} \mid \neg\exists g' \leq s.g \in \mathbb{K}.\langle s, g'\rangle \in \bigcup_{j=0}^{i-1} \mathcal{L}_j \wedge \neg\exists g' < s.g \in \mathbb{K}.\langle s, g'\rangle \in \mathcal{H}\}$
  end while
  return $true$
```

8.4 Beam Search in the μCRL Toolset

In the μCRL toolset, we implemented the G-synchronised variants of beam search, presented in this chapter, where G equals the cumulated cost function g, and the exploration phase is performed in an (action-based) minimal-cost manner (see minimal-cost search in Section 7.3.4), therefore these variants can be applied on unweighted state spaces with an action-based representation of costs. In these variants, there is no additional space needed to store (intermediate) cumulated cost results for states, and the duplicate detection can be done straightforwardly, not considering the cumulated costs. Next, we describe how total-cost and priority evaluation functions are represented in the toolset.

In our implementation of priority beam search, priority values are assigned to actions, as opposed to nodes in classic beam search, and are fed to the state space generator in an input file. In this setting, each trace in the state space represents a sequence of jobs (as in the case of the Clinical Chemical Analyser presented in Chapter 10). To be precise, priority values are assigned to action *labels* and are fixed during the search. Therefore, identical action labels have equal priority levels regardless of their source, destination or parameters (if present). By default, all actions have priority zero.

Related to detailed beam search, which uses a total-cost evaluation function, the μCRL toolset can perform a g-synchronised detailed search with $f(s) = h(s)$.[5] The desired estimation function $h : \mathscr{S} \to \mathbb{K}$ can be specified using constants and variables taken from the parameter list of the LPE (see Definition 4) of the specification, combined with the usual operations, i.e. addition, subtraction, multiplication and division. If the estimation function has a sophisticated structure, e.g. depends on some pre-calculated information as in the case of Oechsner and Rose (2005), it should be encoded in the specification. In μCRL, abstract data types are used to specify data structures and functions. This is very expressive and allows creating many useful functions, possibly incorporating pre-calculated data. What remains to be done in the specification is to ascertain that a designated parameter of a process is updated at right moments with the appropriate value of this function. Then the actual estimation function can use this parameter.

In general, G-synchronised searches can be applied, as long as the G-guiding is action-based, like e.g. minimal-cost search, which can keep track of g-values by recording the number of encountered *tick* transitions along the way. Also, flexible versions of the implemented variants are available.

Of course, a total-cost function $f(s) = g(s) + h(s)$ can be achieved by keeping track of cumulated costs of states in a special variable *cost* in the specification, as explained in Section 7.2.

Case studies on timed scheduling problems using the beam search implementations in the μCRL toolset are discussed in Chapter 10.

[5] In fact, note that a guiding function $f(s) = g(s) + h(s)$ has the same effect here, as in each round of the search, g-synchronised detailed beam search considers states with the same g-value, therefore these states can only have different f-values if they have different h-values.

8.5 Memory Management

Memory management is a challenging issue in state space generation. Although beam search reduces memory use due to cutting away parts of the state space, still explored states need to be accessed to guarantee the termination of the exploration in case of cyclic state spaces. Keeping the whole set of visited states in the memory is usually susceptible to early state space explosion. This can be counter-measured by taking into account specific characteristics of the problem at hand and the properties that are to be checked. Below we discuss some possible optimisations when applying beam search:

1. When aiming at a reachability property on-the-fly (such as reachability of a goal state, checking invariants and hunting deadlock states), the memory requirements can be lowered by checking the property while exploring. In that case, once a state satisfying the desired property is reached, the search terminates and the witness trace is reported. This however cannot be extended to arbitrary properties.

2. If there are no cycles in the state space, there is in principle no need to check whether a state has already been visited (in order to guarantee termination). Therefore, only the states from the current level need to be kept and the rest can be removed from memory, i.e. flushed to high latency media such as disks. In this case, however, some states may be revisited due to confluent traces, hence undesirably increasing the search time. Prominent examples of systems with acyclic state spaces are large classes of scheduling problems, which have been traditional targets of beam search, and most security protocols (see Section 8.7). As is demonstrated by Torabi Dashti and Wijs (2007), a known POR algorithm for security protocols can be seen as an instance of beam search.

3. In detailed beam search variants, if each state s has a unique cumulated cost $g(s)$ associated to it, e.g. denoting a notion of progress, and if g is monotonic, then there cannot be any transition from states with a greater cumulated cost to the states with lower cumulated costs: $g(s) < g(s') \implies s \not\to^* s'$. Consequently, states with cumulated costs strictly lower than the cumulated costs of the states to be processed can be removed from memory. This resembles sweep-line state exploration (Christensen et al., 2001).

4. In G-synchronised beam search variants with a monotonic G-function, bit-state hashing (Holzmann, 1998) can be used to reduce memory use. This technique is however inherently incomplete, i.e. may miss parts of the state space, and in particular when used in beam search there is the possibility of ignoring a previously visited state when it is reached via a path with a lower G-value. However, for G-synchronised BS variants with monotonic G, this does not pose a problem, since

re-opening of states is never needed. Note that the approach remains an approximation to beam search and, thus, can be seen as a trade-off between memory usage and having a tight grip on pruning.

8.6 Heuristics and Selecting the Beam Width

Effectiveness of beam search hinges on selecting good heuristic functions. Heuristic functions, as George Polya put it in 1945, are meant “to discover the solution to the present problem” (Polya, 1945), and thus heavily depend on the problem being solved. As the focus of the current chapter is the development of search algorithms working with heuristics, we do not discuss techniques to design the heuristic functions themselves. Developing heuristics constitutes a whole separate body of research and, here, we refer to a few case studies on using heuristics in pruning state spaces: Among others, Groce and Visser (2004), Oechsner and Rose (2005), Si Ow and Morton (1988), and Valente and Alves (2005b) present detailed discussions on pruning heuristics when dealing with Java program analysis, scheduling a wafer stepper machine, and jobshop scheduling problems, respectively. In Chapter 10, we show the effect of using heuristics to schedule a Clinical Chemical Analyser.

Particularly papers on *designing* heuristic functions, such as the work by Edelkamp et al. (2001a), Edelkamp et al. (2004), Groce and Visser (2004), and Kupferschmid et al. (2006), constitute a nice complement to the work we present here, as they explain how to design heuristic functions and we start with the assumption of having a heuristic function. Edelkamp et al. (2001a) and Edelkamp et al. (2004) use guidelines to approximate the distance to deadlocks and violations of invariants and assertions. The objective of Groce and Visser (2004) is to model check Java programs with heuristics constructed using the properties to check, the structure of the programs and additional input of the user. Such functions can naturally be used as input also to the algorithms proposed in the current chapter.

Selecting the beam width β is another challenge in using beam search. The beam width intuitively calibrates the time/memory usage of the algorithm on one hand and the accuracy of the results on the other hand. Therefore, in practice the time/memory limits of a particular experiment determine β. To reduce the sensitivity of the results to the exact value of β, we propose using flexible beam search variants, c.f. the results in Chapter 10. This, however, comes at the price of losing a tight grip on the memory consumption (see also Section 8.3.4).

For more general discussions on selecting β and its relation to the quality of answer we refer to Si Ow and Morton (1988).

8.7 Connections to Other Heuristic Search Algorithms

Having described the beam search spectrum of Figure 8.4, we observe that some existing search techniques fit neatly in there. We here briefly mention some of these connections to other search algorithms. This can be particularly interesting in practice, where one looks for umbrella theories and tools to cover as many existing techniques as possible to ease their use and interoperability. Our main result is that the basic AI search algorithm A^* (Hart et al., 1968) can be seen as an instance of G-synchronised flexible detailed beam search, the bottom right corner of the spectrum of Figure 8.4. First, in order to be able to compare search behaviour of different best-first searches, we formally define the notions of a *guiding signature of a best-first search*, and *strong* and *weak guiding equivalence* of two best-first searches, and we demonstrate the usefulness of these notions, by proving that uniform-cost is strong guiding equivalent to an instance of G-synchronised detailed beam search. Next, we present A^* in pseudo-code, and prove that, given a monotonic guiding function $f(s) = g(s) + h(s)$, it is both weak guiding equivalent to set-based A^* and weak guiding equivalent to f-synchronised flexible detailed beam search.

We observe that the behaviour of a best-first search through a state space $\mathscr{M}$ can be uniquely described by expressing the behaviour of each round in the search.[6] The latter can be done by means of two functions $\xi, \nu : 2^{\mathscr{S}} \to 2^{\mathscr{S}}$, which describe the state space level creation and the search horizon creation in each round of the search algorithm, respectively. This insight is used to define the notion of a guiding signature of a best-first search in Definition 24.

Definition 24 (Guiding signature of a best-first search). *For any $\mathscr{M}$ =($\mathscr{S}$,$\mathscr{A}$,$\mathscr{C}$,$\mathscr{T}$, $\mathscr{I}$), we say that the* guiding signature of a best-first search A *is a tuple* (ξ, ν)*, where* $\xi : 2^{\mathscr{S}\times\mathbb{K}} \to 2^{\mathscr{S}\times\mathbb{K}}$ *is a function describing the creation of a state space level, given a search horizon, in a round of A, and $\nu : 2^{\mathscr{S}\times\mathbb{K}} \to 2^{\mathscr{S}\times\mathbb{K}}$ is a function describing the evolution of a search horizon through a round in A.*

We denote applying ν exactly n times on S as $\nu^n(S)$.

Observe that the search horizon $\mathscr{H}$ of a best-first search with guiding signature (ξ, ν) through a state space $\mathscr{M}$ evolves as follows through the search rounds: $\mathscr{I}, \nu(\mathscr{I}), \nu^2(\mathscr{I})$, etc. The state space levels consecutively created by A are $\mathscr{L}_0 = \xi(\mathscr{I})$, $\mathscr{L}_1 = \xi(\nu(\mathscr{I}))$, $\mathscr{L}_2 = \xi(\nu^2(\mathscr{I}))$, etc.

Next, we define the notion of *strong guiding equivalence*, by which we can compare the behaviour of best-first searches. This is defined in Definition 25.

[6] Actually, iterative best-first searches form an exception to this. Note that a search A and an iterative version of A search in exactly the same way up to the end of the first iteration of the iterative search; after this point A has terminated and the iterative search is in its second iteration. In the current context, though, there is no need to distinguish iterative and non-iterative searches.

Definition 25 (Strong guiding equivalence). *Given two best-first searches A and B with guiding signature* (ξ_A, ν_A) *and* (ξ_B, ν_B)*, respectively. We say that A and B are* strong guiding equivalent *if for any* $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$ *and* $S \subseteq \mathscr{S}$*, we have* $\xi_A(S) = \xi_B(S)$ *and* $\nu_A(S) = \nu_B(S)$*.*

Now, we can prove the following lemma:

Lemma 4. *Given a cumulated-cost function* $g(s)$ *and* $f(s) = g(s)$*, uniform-cost search (see Section 6.4) is strong guiding equivalent to g-synchronised detailed beam search, with* $\beta = \infty$*.*

Proof. Given a state space $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$. Let $\mathscr{H} \subseteq \mathscr{S}$ be the current search horizon in round i of the searches. In order to prove that uniform-cost search (UCS) and g-synchronised detailed beam search (g-BS), with $\beta = \infty$, are strong guiding equivalent, we must show that:

1. $\xi_{\text{UCS}}(\mathscr{H}) = \xi_{g\text{BS}}(\mathscr{H})$;
2. $\nu_{\text{UCS}}(\mathscr{H}) = \nu_{g\text{BS}}(\mathscr{H})$.

From Algorithm 9 with $f(s) = g(s)$ and Algorithm 21 with $G(s) = g(s)$, we observe that:

- $\xi_{\text{UCS}}(S) = \mathscr{L}_i = \{\langle s, s.g\rangle \in S \mid \forall \langle s', s'.g\rangle \in S.f(s) \leq f(s')\}$;
- $\nu_{\text{UCS}}(S) = \{\langle s, s.g\rangle \in S \cup \{s \mid \exists s' \in \xi_{\text{UCS}}(S), \ell \in \mathscr{A}.s' \xrightarrow{\ell} s\} \mid \neg\exists g' \leq s.g \in \mathbb{K}.\langle s, g'\rangle \in \bigcup_{j=0}^{i} \mathscr{L}_j \wedge \neg\exists g' < s.g \in \mathbb{K}.\langle s, g'\rangle \in S \cup \{s \mid \exists s' \in \xi_{\text{UCS}}(S), \ell \in \mathscr{A}.s' \xrightarrow{\ell} s\}\}$;
- $\xi_{g\text{BS}}(S) = \mathscr{L}_i = \mathscr{H}' = \{\langle s, s.g\rangle \in S \mid \forall \langle s', s'.g \in S\rangle.G(s) \leq G(s')\}$ (Since $\beta = \infty$, no pruning is done);
- $\nu_{g\text{BS}}(S) = \{\langle s, s.g\rangle \in S \cup \{s \mid \exists s' \in \xi_{g\text{BS}}(S), \ell \in \mathscr{A}.s' \xrightarrow{\ell} s\} \mid \neg\exists g' \leq s.g \in \mathbb{K}.\langle s, g'\rangle \in (\bigcup_{j=0}^{i-1} \mathscr{L}_j \cup \xi_{g\text{BS}}(S)) \wedge \neg\exists g' < s.g \in \mathbb{K}.\langle s, g'\rangle \in (S \setminus \xi_{g\text{BS}}(S)) \cup \{s \mid \exists s' \in \mathscr{L}_i, \ell \in \mathscr{A}.s' \xrightarrow{\ell} s\}\}$

Clearly, $\xi_{\text{UCS}}(\mathscr{H}) = \xi_{g\text{BS}}(\mathscr{H})$ and therefore also $\nu_{\text{UCS}}(\mathscr{H}) = \nu_{g\text{BS}}(\mathscr{H})$. □

Comparing the standard (not the set-based) A^* with set-based A^* or f-synchronised flexible detailed beam search, though, we observe that these searches are not strong guiding equivalent. However, they are equivalent according to a weaker notion, which we call weak guiding equivalence, defined in Definition 26.

Definition 26 (Weak guiding equivalence). *Given two best-first searches A and B with guiding signature* (ξ_A, ν_A) *and* (ξ_B, ν_B)*, respectively. We say that A and B are* weak guiding equivalent *if for any* $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$ *and* $S \subseteq \mathscr{S}$*, there are* $n, m \geq 0$ *such that* $\xi_A(S) \cup \xi_A(\nu_A(S)) \cup \ldots \cup \xi_A(\nu_A^{n-1}(S)) = \xi_B(S) \cup \xi_B(\nu_B(S)) \cup \ldots \cup \xi_B(\nu_B^{m-1}(S)) \subset \mathscr{S}$ *and* $\nu_A^n(S) = \nu_B^m(S)$*.*

With weak guiding equivalence, we express that, given a search horizon $\mathcal{H}$, two searches A and B can be considered equal if they produce the same set of states in a finite number of levels, *not constituting the full state space*, otherwise any two exhaustive searches would be weak guiding equivalent. Furthermore, the search horizons of A and B after the production of those levels are equal. Note that considering the two equivalence notions in the context of reachability problems, given a state space $\mathcal{M}$, if a search A finds a goal state $s \in \mathcal{G}$ in some $\mathcal{L}_k$, with $k \geq 0$, then a strong guiding equivalent search B through $\mathcal{M}$ also finds a goal state in $\mathcal{L}_k$, and that unlike strong guiding equivalence, weak guiding equivalence with respect to a state space $\mathcal{M}$ does not imply equality of the searches with respect to efficiency in finding a goal state.

In the terminology of Chapter 6, A^* is a search algorithm with a guiding function $f(s) = g(s) + h(s)$, no pruning in the width, and no selection of extra states (see Section 6.8). For the sake of clarity, we present A^* explicitly in pseudo-code in Algorithm 22, based on the general algorithm for best-first search (Algorithm 10).

Algorithm 22 A^* search

```
Require: $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$, heuristic function $h$, selection width $\beta$ (either 1 or $\infty$),
  set of goal states $\mathcal{G}$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathcal{H} \leftarrow \mathcal{I}$
  while $\mathcal{H} \neq \emptyset$ do
    $\mathcal{L}_i \leftarrow select_\beta(\{s \in \mathcal{H} \mid \forall s' \in \mathcal{H}. g(s) + h(s) \leq g(s') + h(s')\})$
    $\mathcal{H} \leftarrow \mathcal{H} \setminus \mathcal{L}_i$
    if $\mathcal{L}_i \cap \mathcal{G} \neq \emptyset$ then
      return $GeneratePath(\mathcal{L}_i \cap \mathcal{G})$
    end if
    for all $s \in \mathcal{L}_i$ do
      $\mathcal{H} \leftarrow \mathcal{H} \cup nxt_{\mathcal{M}}(s, en_{\mathcal{M}}(s))$
    end for
    $i \leftarrow i + 1$
    $\mathcal{H} \leftarrow DuplicateFree(\mathcal{H}, \bigcup_{j=0}^{i-1} \mathcal{L}_j)$
  end while
  return $true$
```

Observe that Algorithm 22 describes both standard A^* and set-based A^*, depending on the value of β (1 and ∞, respectively).

Lemma 5. *Given a monotonic total-cost function $f(s) = g(s) + h(s)$, A^* is weak guiding equivalent to set-based A^*.*

Proof. Given a state space $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$. Let $\mathcal{H} \subseteq \mathcal{S}$ be the current search horizon in round i of the searches. In order to prove that A^* and set-based A^* (sA^*), are weak guiding equivalent, we must show that for some $n, m \geq 0$, $\xi_{A^*}(\mathscr{H}) \cup \xi_{A^*}(\nu_{A^*}(\mathscr{H})) \cup \ldots \cup \xi_{A^*}(\nu_{A^*}^{n-1}(\mathscr{H})) = \xi_{sA^*}(\mathscr{H}) \cup \xi_{sA^*}(\nu_{sA^*}(\mathscr{H})) \cup \ldots \cup \xi_{sA^*}(\nu_{sA^*}^{m-1}(\mathscr{H})) \subset \mathscr{S}$ and $\nu_{A^*}^{n}(\mathscr{H}) = \nu_{sA^*}^{m}(\mathscr{H})$.

From Algorithm 22 we observe that:

- $\xi_{A^*}(S) = \mathscr{L}_i \in \{s \in S \mid \forall s' \in S.f(s) \leq f(s')\}$ (Since $\beta = 1$);
- $\nu_{A^*}(S) = \mathit{DuplicateFree}(S \cup \{s \mid \exists s' \in \xi_{A^*}(S), \ell \in \mathscr{A}.s' \xrightarrow{\ell} s\}, \bigcup_{j=0}^{i} \mathscr{L}_j)$;
- $\xi_{sA^*}(S) = \mathscr{L}_i = \{s \in S \mid \forall s' \in S.f(s) \leq f(s')\}$ (Since $\beta = \infty$);
- $\nu_{sA^*}(S) = \mathit{DuplicateFree}(S \cup \{s \mid \exists s' \in \xi_{sA^*}(S), \ell \in \mathscr{A}.s' \xrightarrow{\ell} s\}, \bigcup_{j=0}^{i} \mathscr{L}_j)$.

First of all, we consider f to be strictly increasing, as a special case of being monotonic, i.e. $\forall s, s' \in \mathscr{S}.s \rightarrow^{+} s' \implies f(s) < f(s')$, with $\rightarrow^{+}$ the transitive closure of $\xrightarrow{\ell}$, for any $\ell \in \mathscr{A}$. Since the exploration of a state s only leads to visiting new states with f-values greater than s, it follows that for $n = \mid \{s \in \mathscr{H} \mid \forall s' \in \mathscr{H}.f(s) \leq f(s')\} \mid$ and $m = 1$, we have $\xi_{A^*}(\mathscr{H}) \cup \xi_{A^*}(\nu_{A^*}(\mathscr{H})) \cup \ldots \cup \xi_{A^*}(\nu_{A^*}^{n-1}(\mathscr{H})) = \xi_{sA^*}(\mathscr{H}) \subset \mathscr{S}$ and $\nu_{sA^*}^{n}(\mathscr{H}) = \nu_{sA^*}^{m}(\mathscr{H})$.

Next, we consider f to be monotonic, but not strictly increasing. Then, A^* will explore all the states in the set $\{s \mid \exists s' \in \{\hat{s} \in \mathscr{H} \mid \forall \hat{s}' \in \mathscr{H}.f(\hat{s}) \leq f(\hat{s}')\}.s' \rightarrow^{*} s \wedge f(s) = f(s')\}$ before any other states. Since A^* explores one state per round, this is done in $\mid \{s \mid \exists s' \in \{\hat{s} \in \mathscr{H} \mid \forall \hat{s}' \in \mathscr{H}.f(\hat{s}) \leq f(\hat{s}')\}.s' \rightarrow^{*} s \wedge f(s) = f(s')\} \mid$ rounds. Likewise, sA^* will explore this set in a number of rounds equal to the longest trace leading from a state $s \in \{s \in \mathscr{H} \mid \forall s' \in \mathscr{H}.f(s) \leq f(s')\}$ to a state $s' \in \{s \mid \exists s' \in \{\hat{s} \in \mathscr{H} \mid \forall \hat{s}' \in \mathscr{H}.f(\hat{s}) \leq f(\hat{s}')\}.s' \rightarrow^{*} s \wedge f(s) = f(s')\}$. If we call the latter number m, and $n = \mid \{s \mid \exists s' \in \{\hat{s} \in \mathscr{H} \mid \forall \hat{s}' \in \mathscr{H}.f(\hat{s}) \leq f(\hat{s}')\}.s' \rightarrow^{*} s \wedge f(s) = f(s')\} \mid$, then we have $\xi_{A^*}(\mathscr{H}) \cup \xi_{A^*}(\nu_{A^*}(\mathscr{H})) \cup \ldots \cup \xi_{A^*}(\nu_{A^*}^{n-1}(\mathscr{H})) = \xi_{sA^*}(\mathscr{H}) \cup \xi_{sA^*}(\nu_{sA^*}(\mathscr{H})) \cup \ldots \cup \xi_{sA^*}(\nu_{sA^*}^{m-1}(\mathscr{H})) \subset \mathscr{S}$ and $\nu_{A^*}^{n}(\mathscr{H}) = \nu_{sA^*}^{m}(\mathscr{H})$. □

In f-synchronised flexible detailed beam search, as in any G-synchronised beam search, a round i in the search consists of two phases. In the first phase of each round, a subset $\mathscr{H}'$ of the search horizon $\mathscr{H}$ is selected, where $\mathscr{H}' = \{s \in \mathscr{H} \mid \forall s' \in \mathscr{H}.f(s) \leq f(s')\}$. In general, in a G-synchronised detailed beam search, the second phase of a round selects a subset of the result of the first phase, using the function f, and prunes the remaining states. However, note that in f-synchronised detailed beam search, selecting a subset of $\mathscr{H}'$ necessarily incorporates tie-breaking, as all the states in $\mathscr{H}'$ have the same f-value. But a flexible variant of f-synchronised detailed beam search avoids tie-breaking, therefore it selects the entire $\mathscr{H}'$ as $\mathscr{L}_i$ in the second phase. In short, f-synchronised flexible detailed beam search selects in each round i all states from $\mathscr{H}$ with the minimal f-value, places them in $\mathscr{L}_i$, expands them, and adds the successor states to $\mathscr{H}$. This gives rise to Lemma 6.

Lemma 6. *Given a monotonic total-cost function $f(s) = g(s) + h(s)$, f-synchronised flexible detailed beam search, with arbitrary $\beta > 0$, is weak guiding equivalent to A^*.*

Proof. Given a state space $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, let $\mathscr{H} \subseteq \mathscr{S}$ be the current search horizon in round i of the searches. In order to prove that A^* and f-synchronised flexible detailed beam search (identified in this proof as fBS), are weak guiding equivalent, we must show that for some $n, m \geq 0$, $\xi_{A^*}(\mathscr{H}) \cup \xi_{A^*}(\nu_{A^*}(\mathscr{H})) \cup \ldots \cup \xi_{A^*}(\nu_{A^*}^{n-1}(\mathscr{H})) = \xi_{f\mathrm{BS}}(\mathscr{H}) \cup \xi_{f\mathrm{BS}}(\nu_{f\mathrm{BS}}(\mathscr{H})) \cup \ldots \cup \xi_{f\mathrm{BS}}(\nu_{f\mathrm{BS}}^{m-1}(\mathscr{H})) \subset \mathscr{S}$ and $\nu_{A^*}^{n}(\mathscr{H}) = \nu_{f\mathrm{BS}}^{m}(\mathscr{H})$.

From Algorithm 22 and Algorithm 21 with $G(s) = f(s)$, we observe that:

- $\xi_{A^*}(S) = \mathscr{L}_i \in \{s \in S \mid \forall s' \in S. f(s) \leq f(s')\}$ (Since $\beta = 1$);
- $\nu_{A^*}(S) = \mathit{DuplicateFree}(S \cup \{s \mid \exists s' \in \xi_{A^*}(S), \ell \in \mathscr{A}. s' \xrightarrow{\ell} s\}, \bigcup_{j=0}^{i} \mathscr{L}_j)$;
- $\xi_{f\mathrm{BS}}(S) = \mathscr{L}_i = \mathscr{H}' = \{\langle s, s.g\rangle \in S \mid \forall \langle s', s'.g \in S\rangle . G(s) \leq G(s')\}$ (Since fBS is flexible, and pruning cannot be done here without tie-breaking, no pruning is done);
- $\nu_{f\mathrm{BS}}(S) = \{\langle s, s.g\rangle \in S \cup \{s \mid \exists s' \in \xi_{f\mathrm{BS}}(S), \ell \in \mathscr{A}. s' \xrightarrow{\ell} s\} \mid \neg\exists g' \leq s.g \in \mathbb{K}. \langle s, g'\rangle \in (\bigcup_{j=0}^{i-1} \mathscr{L}_j \cup \xi_{f\mathrm{BS}}(S)) \wedge \neg\exists g' < s.g \in \mathbb{K}. \langle s, g'\rangle \in (S \setminus \xi_{f\mathrm{BS}}(S)) \cup \{s \mid \exists s' \in \mathscr{L}_i, \ell \in \mathscr{A}. s' \xrightarrow{\ell} s\}\}$

First of all, we consider f to be strictly increasing, as a special case of being monotonic, i.e. $\forall s, s' \in \mathscr{S}. s \rightarrow^+ s' \implies f(s) < f(s')$, with $\rightarrow^+$ the transitive closure of $\xrightarrow{\ell}$, for any $\ell \in \mathscr{A}$. Since the exploration of a state s only leads to visiting new states with f-values greater than s, and $G(s) = f(s)$, it follows that for $n = \mid \{s \in \mathscr{H} \mid \forall s' \in \mathscr{H}. f(s) \leq f(s')\} \mid$ and $m = 1$, we have $\xi_{A^*}(\mathscr{H}) \cup \xi_{A^*}(\nu_{A^*}(\mathscr{H})) \cup \ldots \cup \xi_{A^*}(\nu_{A^*}^{n-1}(\mathscr{H})) = \xi_{f\mathrm{BS}}(\mathscr{H}) \subset \mathscr{S}$ and $\nu_{A^*}^{n}(\mathscr{H}) = \nu_{f\mathrm{BS}}^{m}(\mathscr{H})$.

Next, we consider f to be monotonic, but not strictly increasing. Then, A^* will explore all the states in the set $\{s \mid \exists s' \in \{\hat{s} \in \mathscr{H} \mid \forall \hat{s}' \in \mathscr{H}. f(\hat{s}) \leq f(\hat{s}')\}. s' \rightarrow^* s \wedge f(s) = f(s')\}$ before any other states. Since A^* explores one state per round, this is done in $\mid \{s \mid \exists s' \in \{\hat{s} \in \mathscr{H} \mid \forall \hat{s}' \in \mathscr{H}. f(\hat{s}) \leq f(\hat{s}')\}. s' \rightarrow^* s \wedge f(s) = f(s')\} \mid$ rounds. Likewise, since $G(s) = f(s)$, fBS will explore this set in a number of rounds equal to the longest trace leading from a state $s \in \{s \in \mathscr{H} \mid \forall s' \in \mathscr{H}. G(s) \leq G(s')\}$ to a state $s' \in \{s \mid \exists s' \in \{\hat{s} \in \mathscr{H} \mid \forall \hat{s}' \in \mathscr{H}. G(\hat{s}) \leq G(\hat{s}')\}. s' \rightarrow^* s \wedge G(s) = G(s')\}$. If we call the latter number m, and $n = \mid \{s \mid \exists s' \in \{\hat{s} \in \mathscr{H} \mid \forall \hat{s}' \in \mathscr{H}. f(\hat{s}) \leq f(\hat{s}')\}. s' \rightarrow^* s \wedge f(s) = f(s')\} \mid$, then we have $\xi_{A^*}(\mathscr{H}) \cup \xi_{A^*}(\nu_{A^*}(\mathscr{H})) \cup \ldots \cup \xi_{A^*}(\nu_{A^*}^{n-1}(\mathscr{H})) = \xi_{f\mathrm{BS}}(\mathscr{H}) \cup \xi_{f\mathrm{BS}}(\nu_{f\mathrm{BS}}(\mathscr{H})) \cup \ldots \cup \xi_{f\mathrm{BS}}(\nu_{f\mathrm{BS}}^{m-1}(\mathscr{H})) \subset \mathscr{S}$ and $\nu_{A^*}^{n}(\mathscr{H}) = \nu_{f\mathrm{BS}}^{m}(\mathscr{H})$. □

The major difference between A^* and fBS is that in the latter, all the states of the $\mathscr{H}$ set with the minimal f-value are collected in $\mathscr{L}_i$, and are all expanded in one go (i.e. nothing is pruned, since the search is flexible and the members of $\mathscr{H}'$ all have the same f-value), while in A^* they are expanded one by one. If f is increasing, all the successor states of these states (with minimal f) will have an f-value greater than their parents, therefore A^* will do exactly what f-synchronised flexible detailed beam search does, i.e. it will first explore the members of $\mathscr{H}'$ before considering any other state. In case f is monotonic, but not strictly increasing, the set $\{s \mid \exists s' \in \{\hat{s} \in \mathcal{H} \mid \forall \hat{s}' \in \mathcal{H}. f(\hat{s}) \leq f(\hat{s}')\}. s' \rightarrow^* s\}$ is explored before any other states. This holds for both searches, although they may explore this set in different numbers of rounds.

Finally, note that f-synchronised flexible detailed beam search and set-based A^* are strong guiding equivalent.

We also observe that the POR algorithm of Clarke et al. (2000) for security protocols can be seen as an instance of flexible priority beam search. The main principle of POR is to exploit the commutativity of concurrently executed transitions in order to generate only a sufficient fraction of the state space by exploring a subset of enabled transitions $\mathit{ample}(s) \subseteq \mathit{en}_{\mathcal{M}}(s)$ at each state s. This resembles priority beam search, since at each state, based on the suitability of the enabled transitions, some of the successors are pruned away while generating. However, in contrast to priority beam search, no essential information is lost in POR as the *ample* set is selected such that a certain class of desired properties is preserved. We refer to Clarke et al. (1999) for a general introduction to POR. Torabi Dashti and Wijs (2007) provide a translation from this algorithm to the general pruning framework, and Fokkink et al. (2007) extend the algorithm within this framework to be applicable to branching security protocols.

Felner et al. (2003) extend best-first search to k-best-first search, allowing to compensate for inaccuracies in the evaluation function by selecting in each iteration more than only the best state. Essentially, the difference between k-best-first search and beam search is the decision to keep states not selected in one iteration for the next iteration (this technique is also described in Section 6.7). This makes k-best first search a complete search, but it also means its memory requirement is higher, since there is no pruning done. A trade-off can, however, be achieved, by using inadmissible heuristics, such that fewer states are expanded, but the solution will be near-optimal. This trade-off is also used for weighted A^* by Pohl (1970), and linear-space best-first search by Korf (1993), where the h-function is multiplied by some factor. Moreover, in the latter, the memory requirement is linear in the size of the search depth.

Our extension of g-synchronised beam search can probably best be compared with filtered beam search (Si Ow and Morton, 1988), in the sense that in each iteration, the current set of states undergoes two phases; in filtered beam search, first a priority beam search is applied, and on the outcome of that, detailed beam search is used, this to lessen the computational complexity. In g-synchronised beam search, we first postpone some states, and then prune states from the remaining set. Both searches can be seen as instances of multi-phase best-first search (Section 6.10).

8.8 Related Work

The literature on traditional beam search mainly focuses on how beam search is useful for solving a specific problem and no general framework is presented. For example, Valente and Alves (2005a) provide a relatively small specification of a typical jobshop scheduling problem that does not include data structures, which are often necessary when dealing with practical systems. Oechsner and Rose (2005) use a very specific program, which is only able to simulate the case study presented there. In these papers, beam search is used on a case by case basis, therefore reusing their implementation of beam search on other case studies is not straightforward. We provide a general framework, based on an expressive specification language, instead of case-based tools. This allows us to easily describe complex systems and various problem restrictions.

In many applications of beam search, no restrictions, neither timing nor data, are initially put on scheduling jobs (e.g., in simple jobshop scheduling problems (Pinedo, 1995) or in the case of Oechsner and Rose (2005)). However, in single machine early/tardy jobshop scheduling problems (Valente and Alves, 2005a,c), for instance, there are (timing) restrictions on scheduling jobs. But the violation of these restrictions is usually allowed while penalties are put on them, hence not excluding violations from the search space. These restrictions, in their most general form, can either be hard, meaning that they have to be necessarily met, or soft, that is the violation of these requirements will result in a penalty, but is still allowed. Soft restrictions can simply be modelled by adding extra costs on prohibited actions. We contend that hard restrictions should be specified in the model, because allowing unwanted executions leads to a search space larger than necessary. This, however, requires an expressive specification language, which seems not readily available, e.g., for Valente and Alves (2005a,c), where restrictions are applied on the model after generation. In μCRL, conditions on data and timing restrictions can be specified in a straightforward manner.

The flexible variants of beam search presented in this chapter are remotely similar to *beam search with variable width*, as described by Valente and Alves (2005a). They completely leave out a predefined beam width and introduce deviation parameters so that, broadly speaking, the algorithm calculates how far the evaluation of a node may be from the optimal evaluation value of that level to still be selected for exploration. In comparison, Valente and Alves take away some influence of the evaluation function, to be able to consider more possibilities when searching a tree, whereas we give more influence to the evaluation function and do not allow any other criteria to affect the selection, in order to reestablish the importance of the evaluation function after we moved the searches to the state space setting. There are significant implementation differences between flexible beam search and beam search with variable width of Valente and Alves (2005a). In detailed beam search with variable width, at each level, the largest total-cost value of the states of the level must be known before selection can proceed. This completely disables the second optimisation mentioned in Section 8.3.3. Furthermore, in priority beam search with variable width, the priority threshold has to be separately computed for each node, which can be computationally expensive.

Groce and Visser (2004) use a number of search algorithms to generate state spaces, one of which called beam search. Their beam search, however, deviates from our usage, in that they let $f(s) = h(s)$, making it practically a linear space greedy search; it is, nevertheless, according to the original notion, a beam search. Furthermore, they include duplicate detection, but do not consider other extensions in order to deal more efficiently with arbitrary state spaces, such as a flexible beam width.

Beam search is extended to a complete search by Zhou and Hansen (2005), by using a new data structure, called a *beam stack*. With this, it is possible to achieve a range of searches, from depth-first search ($\beta = 1$) to breadth-first search ($\beta = \infty$). Considering our extensions for arbitrary state spaces, it would be interesting to try to combine these two approaches. Edelkamp and Jabbar (2007) apply a search width k on breadth-first BnB search for priced timed automata. The resulting algorithm is made complete by using iterative broadening. There, k expresses the percentage of states that should be explored per level of the state space; they also allow the selection of more than $k\%$, which is comparable to our flexible beam width. The usage of a heuristic function is not considered; instead, selection is done based on cumulated costs.

Our work is also related to the body of research on DMC, where, to find a counter-example to a functional property (usually belonging to LTL) with a minimal exploration of the state space, heuristics (based on the property) are used to guide the search. Using A^* (Edelkamp et al., 2004) and genetic algorithms (Godefroid and Khurshid, 2002) to guide the search are among notable works in this field. Connections to other heuristic search algorithms are discussed in Section 8.7. In contrast to DMC, we generate a *partial* state space in which an arbitrary property can be checked *afterwards* (the result would of course not be exact, hence being useful mainly in quantitative analyses where a near-optimal solution for a problem often suffices). In this sense, these approaches are different in spirit, addressing rather different problems (i.e. checking qualitative vs. quantitative properties) from different angles (directing the search vs. searching an approximation to the state space). Nonetheless, there are strong similarities as well: A^* can be seen as an instantiation of our extensions of beam search (see lemma 6), and the approach of Godefroid and Khurshid (2002) is similar to ours, as it is in general not guaranteed to explore the whole state space.

8.9 Conclusions

In this chapter, we extended and made available an existing search technique to be used for quantitative analysis within a setting used for system verification. By doing so, we contribute to attempts to achieve a general framework in which different types of analysis, such as functional verification and scheduling, can be performed on a single system specification. Moving beam search to the field of state spaces, we experienced that the algorithm needed to be extended in order to counter a decrease of influence of the heuristic function used. Beam search with a flexible beam width can cope with encountering more than β sufficiently promising states. Using the algorithm for generating state spaces also introduces the well-known issue of re-opening states in DMC. G-synchronised beam search is another introduced extension of beam search, of which the instance g-synchronised beam search, in cases where the g-function is monotonic, avoids re-opening states, by using a two-phase approach; in the first phase, states are ordered based on the cost needed to reach them from $\mathscr{I}$. In the second phase, the states are considered in this order and an estimation function is applied to prune relatively unpromising states. The extensions of beam search give rise to a beam search spectrum. We introduced a way to compare the guiding behaviour of different best-first searches, and presented some example comparisons, relating uniform-cost search and A^* to our spectrum. Our experiments in Chapter 10 show the usefulness and flexibility of the extensions presented in this chapter.

Chapter 9

Distributed Searching

> *Not everything that counts can be counted, and not everything that can be counted counts.*
>
> (*Albert Einstein*)

9.1 Introduction

IN THIS PART OF THE THESIS, we started with an overview on the different types of searches applicable on state spaces. We saw techniques to limit the search to ‘interesting’ parts, depending on the problem to solve and/or the property to check. Some of these techniques can limit a search, such that it remains complete, i.e. its result is guaranteed to be correct (such a technique is used in e.g. depth-first BnB search); other, often more aggressive, limiting techniques do not preserve cost-optimality, making the search a near-optimal one (e.g. pruning as in beam search).

Focussing on beam search, it proves to be very fruitful for finding near-optimal schedules for a given scheduling problem, as examples show in Chapter 10. Sometimes finding a solution may, however, still take quite a lot of time, mostly due to the extra computation needed to evaluate states. Besides that, in specifications of scheduling problems, often complex data structures are used, making the computational complexity to calculate the successor states of a given state s higher. One can improve on these points by moving the beam search techniques to a distributed setting. In this chapter, we first introduce *distributed minimal-cost search*, building on both minimal-cost search from Algorithm 18 and distributed breadth-first state space generation from Algorithms 4 and 5. Next, we propose distributed versions of the beam search variants of Chapter 8, focussing on detailed beam search, since due to its global view when pruning, it is not obvious how a distributed algorithm should function.

9.2 Distributed Minimal-cost Search

In Section 7.3.4, we presented a (sequential) search algorithm to find minimal-cost traces, as defined in Definition 19, called minimal-cost search (or g-synchronised search, according to the terminology of Chapter 8). This search is applicable on state spaces which contain an action-based encoding of costs using the special *tick* transition label. As is the case in general with state space search algorithms, in practice, the capabilities of the search are limited by the available memory and computing power of the machine performing the search. These limitations can be stretched by moving to a distributed setting, i.e. running the search on a cluster of machines. Therefore, we would like to extend minimal-cost search to a version applicable in such a context.

Distributed minimal-cost search maps very well to general distributed breadth-first state space generation from Algorithms 4 and 5, as minimal-cost search is based on breadth-first search. Algorithm 23 describes what happens at the client side. For a precise explanation of the mechanism, we introduce the notion of *cost region* in Definition 27.

Definition 27 (Cost region). *Given a state space $\mathscr{M}$, we say that two states $s, s' \in \mathscr{S}$ are in the same* cost region *iff* $\text{đ}(\mathscr{I}, \{s\}) = \text{đ}(\mathscr{I}, \{s'\})$.

Each client `ID` keeps track of a number of sets. These are:

- $\mathscr{W}^{\mathtt{ID}}$: Contains the states which have to be expanded once the system has moved to a new cost region. For every state $s \in \mathscr{W}^{\mathtt{ID}}$, there exists a state s', already processed by some client in the cluster, such that $s' \xrightarrow{tick} s$. Expanding the states in $\mathscr{W}^{\mathtt{ID}}$ will only commence once no other states in the current cost region can be expanded (signalled by the manager using the `nextcostregion` command).

- $\hat{\mathscr{W}}^{\mathtt{ID}}$: Contains newly found states which have to be expanded in the next cost region (as in $\mathscr{W}^{\mathtt{ID}}$).

- $\mathscr{E}^{\mathtt{ID}}$: Contains all states representing successful termination found by client ID. A state s is an element of $\mathscr{E}$ iff there exists a state s' such that $s' \xrightarrow{finished} s$.

- $\mathscr{L}_i^{\mathtt{ID}}$: Here, all states are received from all the clients in the cluster to be expanded next by client `ID`, in the current cost region.

- $\mathscr{L}_{i+1}^{\mathtt{ID}}$: Right after exploration of the states in $\mathscr{L}_i^{\mathtt{ID}}$, $\mathscr{L}_{i+1}^{\mathtt{ID}}$ contains all the states encountered by client `ID` which can be expanded in the current cost region.

Now, during the generation, a client can receive the following commands from the manager:

- `continue`: In the next round, expand all received states which are situated in the current cost region (i.e. are received in $\mathscr{L}_i^{\mathtt{ID}}$).

- `nextcostregion`: In the next round, expand all received states which are situated in the next cost region (i.e. are received in $\mathscr{W}^{\mathtt{ID}}$).

- `finish`: Stop the search algorithm.

When the client receives a command from the manager other than `finish`, it receives new states to expand from all clients in the cluster. Possibly it has to move to the next cost region, indicated by the `nextcostregion` command. As long as there are states to expand and no successful termination state is found, the client expands states and places the successors in the appropriate sets. Once finished with expanding, the obtained states are distributed, using the hash function $\# : \mathscr{S} \to \{1, \ldots, n\}$ and the functions *SendToClientsNextLevelAndWaiting*(), *RecvFromClientsNextLevel*(), and *RecvFromClientsWaiting*(). The function *SendToMgrGoalStates*() is invoked to send any encountered goal states to the manager. Every client also needs to report in each round whether it has some states waiting to be expanded in the next cost region, which is done via the function *SendToMgrHaveStatesWaiting*(). This tells the manager that even if in a round no new states are encountered, at least the search can continue into the next cost region.

Algorithm 24 shows how the manager behaves during a search. It keeps track of the successful termination states found, whether new states have been found, and whether there are states available for the next cost region. If there are no successful termination states found, but there are new states available, then the clients can move to the next level. If there are no successful termination states and no new states found, but there are states available for the next cost region, then all the clients should move to the next cost region. If successful termination states have been found, or there are no states available anymore, the search has to finish. Finally, if appropriate, the manager will return a minimal-cost path from $\mathscr{I}$ to $\mathscr{E}$. In order to produce this, the function *GeneratePath* incorporates additional communication with the clients in order to gather the appropriate path information; the clients provide this information through the *AnswerTransitionRequestsFromMgr*() function.

9.3 Distributed Detailed Beam Search

Because of the global view of total cost evaluation functions, designing a distributed version of detailed beam search, presented earlier in Chapter 8, is non-trivial. Clients should not select states for further exploration in isolation of each other, but have to communicate.

Let us have a manager and n clients to do a distributed detailed beam search. As described in Chapter 2, we have a hash function $\# : \mathscr{S} \to \mathbb{N}$, which is used to distribute generated states over the clients for future exploration. Let the state space consist of levels $\mathscr{L}_0$, $\mathscr{L}_1$, etc. As detailed beam search is done in a breadth-first manner, each

Algorithm 23 Distributed minimal-cost search - Client Instantiator

```
Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{T}, \mathscr{I})$, client number ID, set of client numbers CIDs, hash function $\# : \mathscr{S} \to$ CIDs
  $i \leftarrow 0$
  $\mathscr{L}_i^{\mathtt{ID}}, \mathscr{E}^{\mathtt{ID}}, \mathscr{W}^{\mathtt{ID}}, \hat{\mathscr{W}}^{\mathtt{ID}} \leftarrow \emptyset$
  for all $s \in \mathscr{I}$ do
    if $\#(s) = \mathtt{ID}$ then
      $\mathscr{L}_i^{\mathtt{ID}} \leftarrow \mathscr{L}_i^{\mathtt{ID}} \cup \{s\}$
    end if
  end for
  repeat
    $\mathscr{L}_{i+1}^{\mathtt{ID}} \leftarrow \emptyset$
    for all $s \in \mathscr{L}_i^{\mathtt{ID}}$ do
      for all $s \xrightarrow{\ell} s' \in en_{\mathscr{M}}(s)$ do
        if $\ell = \mathit{finished}$ then
          $\mathscr{E}^{\mathtt{ID}} \leftarrow \mathscr{E}^{\mathtt{ID}} \cup \{s'\}$
        else if $\ell = \mathit{tick}$ then
          $\hat{\mathscr{W}}^{\mathtt{ID}} \leftarrow \hat{\mathscr{W}}^{\mathtt{ID}} \cup \{s'\}$
        else
          $\mathscr{L}_{i+1}^{\mathtt{ID}} \leftarrow \mathscr{L}_{i+1}^{\mathtt{ID}} \cup \{s'\}$
        end if
      end for
    end for
    $\mathit{SendToMgrGoalStates}(\mathscr{E}^{\mathtt{ID}})$
    $\mathit{SendToMgrNewStatesFound}(|\ \mathscr{L}_{i+1}^{\mathtt{ID}}\ | > 0)$
    $\mathit{SendToMgrHaveStatesWaiting}(|\ \hat{\mathscr{W}}^{\mathtt{ID}} \cup \mathscr{W}^{\mathtt{ID}}\ | > 0)$
    $i \leftarrow i + 1$
    $\mathit{command} \leftarrow \mathit{RecvFromMgr}()$
    if $\mathit{command} \neq$ finish then
      $\mathit{SendToClientsNextLevelAndWaiting}(\mathscr{L}_i^{\mathtt{ID}}, \hat{\mathscr{W}}^{\mathtt{ID}})$
      $\mathscr{L}_i^{\mathtt{ID}} \leftarrow \mathit{RecvFromClientsNextLevel}()$
      $\mathscr{W}^{\mathtt{ID}} \leftarrow \mathscr{W}^{\mathtt{ID}} \cup \mathit{RecvFromClientsWaiting}()$
      if $\mathit{command} =$ nextcostregion then
        $\mathscr{L}_i^{\mathtt{ID}} \leftarrow \mathscr{W}^{\mathtt{ID}}$
        $\mathscr{W}^{\mathtt{ID}} \leftarrow \emptyset$
      end if
      $\mathscr{L}_i^{\mathtt{ID}} \leftarrow \mathscr{L}_i^{\mathtt{ID}} \setminus \bigcup_{j=0}^{i-1} \mathscr{L}_j^{\mathtt{ID}}$
    end if
  until $\mathit{command} =$ finish
  $\mathit{AnswerTransitionRequestsFromMgr}()$
```

Algorithm 24 Distributed minimal-cost search - Manager Instantiator

```
Require: Set of client numbers CIDs
Ensure: If exists, a minimal-cost trace to a goal state is returned
  $\mathcal{E} \leftarrow \emptyset$
  repeat
    $nextlevel, havewaiting \leftarrow false$
    for all ID $\in$ CIDs do
      $\mathcal{E} \leftarrow \mathcal{E} \cup RecvFromClientGoalStates(\mathtt{ID})$
      $nextlevel \leftarrow nextlevel \vee RecvFromClientNewStatesFound(\mathtt{ID})$
      $havewaiting \leftarrow havewaiting \vee RecvFromClientHaveStatesWaiting(\mathtt{ID})$
    end for
    if $\mathcal{E} = \emptyset$ and $nextlevel$ then
      $SendToClients(\mathtt{continue})$
    else if $\mathcal{E} = \emptyset$ and $havewaiting$ then
      $SendToClients(\mathtt{nextcostregion})$
    end if
  until $\mathcal{E} \neq \emptyset$ or $(\neg nextlevel$ and $\neg havewaiting)$
  $SendToClients(\mathtt{finish})$
  if $\mathcal{E} \neq \emptyset$ then
    return $GeneratePath(\mathcal{E})$
  else
    return $false$
  end if
```

level of states $\mathscr{L}_i$ gets distributed over the n clients before exploration, leading to the subsets $\mathscr{L}_i^1$,. . .,$\mathscr{L}_i^n$, such that $\mathscr{L}_i^1 \cup \ldots \cup \mathscr{L}_i^n = \mathscr{L}_i$ for all levels i, where $\mathscr{L}_i^j$ is the subset of $\mathscr{L}_i$ designated to client j by the hash function.

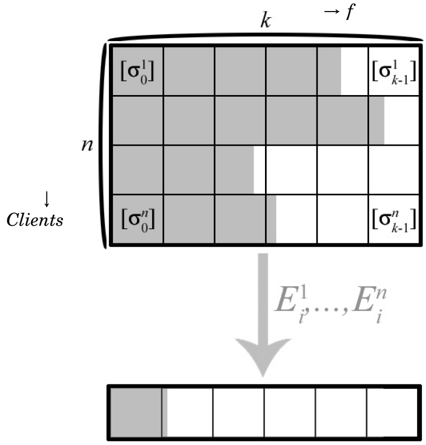

Figure 9.1: Distributing, partitioning, and selecting states

Now, we define function $p_f : 2^{\mathscr{S}} \to 2^{2^{\mathscr{S}}}$, which is used at each level i by each client j. For practical reasons, we assume that k is an upper limit of f. Now, p_f distributes the states from a set $\mathscr{L}_i^j$ over k equivalence classes $[\sigma^j_0], \ldots, [\sigma^j_{k-1}]$, such that $\forall u \in \{0, 1, \ldots, k-1\}. \forall s \in [\sigma^j_u]. f(s) = u$.

We refer to a selection of γ states from a set $\mathscr{L}_i^j$ using evaluation function f as $sel^f_\gamma(\mathscr{L}_i^j) = [\sigma^j_0] \cup \ldots \cup [\sigma^j_r] \cup [\sigma']$, with $r \in \mathbb{N}$ and $r < k-1$, such that $\mid [\sigma^j_0] \cup \ldots \cup [\sigma^j_r] \mid < \gamma$, $[\sigma'] \subseteq [\sigma^j_{r+1}]$ and $\mid [\sigma^j_0] \cup \ldots \cup [\sigma^j_r] \cup [\sigma'] \mid = \gamma$. In practice, $[\sigma'] \subseteq [\sigma^j_{r+1}]$ is composed according to a so-called tie-breaking rule. In the remainder of this chapter, we denote sel^f_γ as sel_γ.

The goal to achieve now for the protocol is the following:

$$\forall i. sel_{\gamma_{i,1}}(\mathscr{L}_i^1) \cup \ldots \cup sel_{\gamma_{i,n}}(\mathscr{L}_i^n) = sel_\beta(\mathscr{L}_i) \tag{9.1}$$

Here, β is the beam width and $\gamma_{i,1}, \ldots, \gamma_{i,n} \in \mathbb{N}$, such that $\gamma_{i,1} + \ldots + \gamma_{i,n} = \beta$. If we could assume that $\gamma_{i,1} = \ldots = \gamma_{i,n}$, then there would be no problem moving the sequential beam search algorithm to a distributed setting. Then, however, besides assuming that the states of each level are evenly distributed over the clients, we also have to assume that the β most promising states of a level are evenly distributed. This we cannot guarantee in general. Instead, we need to move to a more general situation where the $\gamma_{i,j}$'s are unequal to each other. In order to achieve this, extra communication is necessary.

Being in level i, let every client j first determine $sel_\beta(\mathscr{L}_i^j)$, this to be prepared for the worst case scenario where all β most promising states end up at a single client.[1] This is illustrated in the top part of Figure 9.1, where each row in the diagram represents a client, and each column represents an equivalence class. Having constructed the equivalence classes, $sel_\beta(\mathscr{L}_i^j)$ is determined, which, in Figure 9.1, is highlighted in grey for each client. Once this is done, the clients send a set of tuples, each consisting of an evaluation value and the number of states in $sel_\beta(\mathscr{L}_i^j)$ that have this evaluation value to the manager. Formally, the following is sent by each client j, being in level i of the state space:

$$E_i^j = \{(r, \mid [\sigma_r^j] \cap sel_\beta(\mathscr{L}_i^j) \mid) \mid \; 0 \leq r \leq k-1 \wedge \mid [\sigma_r^j] \cap sel_\beta(\mathscr{L}_i^j) \mid \neq 0\} \tag{9.2}$$

All the sets E_i^j sent by the clients are used by the manager to determine a final selection of β states. This is illustrated in the bottom part of Figure 9.1. First, E_i is created as $E_i = \{(j,e,t) \mid \; (e,t) \in E_i^j\}$, where j is the client ID, and e and t correspond to the first and second element in the tuples calculated in Equation 9.2. Similar to p_f, we define a function $p_e : 2^{\mathbb{N}^3} \to 2^{2^{\mathbb{N}^3}}$, which allows us to distribute the elements of the set E_i over k equivalence classes $[e_0], \ldots, [e_{k-1}]$, such that $\forall u \in \{0,1,\ldots,k-1\}.\forall (j,e,t) \in [e_u].e = u$.

For selecting the β best states, we define a function $T_j : 2^{\mathbb{N}^3} \to \mathbb{N}$, which returns the number of states of client j represented in the given evaluation set E_i; more specifically, $T_j(E_i) = \sum_{(j,e,t)\in E'} t$, with $E' = \{(j',e',t') \in E_i \mid \; j' = j\}$. We define $T : 2^{\mathbb{N}^3} \to \mathbb{N}$ as the total number of states represented in the given evaluation set E_i, so $T(E_i) = \sum_{j=1}^{n} T_j(E_i)$. We refer to a selection of β triples from E_i as $evsel_\beta(E_i) = [e_0] \cup \ldots \cup [e_r] \cup [e']$, with $r \in \mathbb{N}$ and $r < k-1$, such that $T([e_0]) + \ldots + T([e_r]) < \beta$, $[e'] = evsubsel_{\beta-(T([e_0])+\ldots+T([e_r]))}([e_{r+1}])$. Here, $evsubsel_{\beta'}([e_u]) = \{(j_0,e_0,t_0)\} \cup \ldots \cup \{(j_{w-1},e_{w-1},t_{w-1})\} \cup \{(j_w,e_w,t'_w)\}$, where $(j_0,e_0,t_0),\ldots,(j_w,e_w,t_w) \in [e_u], t'_w \leq t_w$ and $t_0 + \ldots + t_{w-1} + t'_w = \beta'$. In practice, $[e']$ is composed according to a tie-breaking rule.

Each client j receives a width $\gamma_{i,j}$, constructed by the manager according to Equation 9.3, which the client uses to obtain $sel_{\gamma_{i,j}}(\mathscr{L}_i^j)$. Since $sel_{\gamma_{i,j}}(\mathscr{L}_i^j) \subseteq sel_\beta(\mathscr{L}_i^j)$, this set can be constructed from memory. As it turns out. for this approach only one extra communication round is necessary.

$$\gamma_{i,j} = T_j(evsel_\beta(E_i)) \tag{9.3}$$

One advantage of detailed beam search, as described by Algorithm 20, is that if a

[1] At this point, it is possible to remove all states not in $sel_\beta(\mathscr{L}_i^j)$ from memory.

level contains up to β states, for all states s in the level, $h(s)$, which can be computationally expensive, does not have to be calculated. To achieve this in the distributed version, the manager gets from every client the number of newly generated states. The sum of these numbers equals the complete size of the next level. If it sends this number together with the next `continue` command, the clients know whether or not to prune (see Algorithms 25 and 26).

In general, distributed state space generation algorithms benefit from symmetry. If all clients have to do a similar amount of work, then little to no idle time occurs in any of the clients and therefore no processing power is wasted. However, if we allow unequal $\gamma_{i,j}$s, then the workload of the clients can be very unequal at times. It makes no sense to have clients idle, while they could very well expand states. Exploring more states than originally asked for can in practice, where the accuracy of the evaluation function and the minimally necessary beam width are in general not known, only be seen as an improvement in accuracy.[2] For this reason we decided to create a variant where the manager does not provide every client j with $\gamma_{i,j}$, but a single $\gamma_i = max\{\gamma_{i,1}, \gamma_{i,2}, \ldots, \gamma_{i,n}\}$ is provided to all clients. In this way every client expands the same number of states[3], and we know that the β most promising states are selected. One could argue that another approach is to redistribute the β selected states over the clients, in order to balance the workload. However, then we go against the distribution of the hash function, which means that clients will no longer be able to perform duplicate detection, leading possibly to redundant work.

Algorithms 25 and 26 show what the clients and the manager do in a distributed detailed beam search, respectively. The selection procedure of the manager to obtain the $\gamma_{i,j}$'s, described formally by Equation 9.3, is done in *calcLimit*(). Matching send and receive functions can be identified by their names. Note that duplicate detection is now performed by each client after having received the new set of states to be expanded. This works thanks to the # function, which ensures that a state s is always assigned to the same client j. During the generation, a client can receive the following commands from the manager:

- `continue`: For the next round, receive new states in $\mathscr{L}_i$ and expand them.
- `finish`: Stop the search algorithm.

9.4 Other Beam Search Variants

In Chapter 8, besides detailed beam search, some other variants have been introduced. Priority beam search is performed using a priority evaluation function $prio : \mathscr{A} \rightarrow \mathbb{Z}$,

[2] There are results where a bigger beam width does not correspond to a greater accuracy, such as in Oechsner and Rose (2005) and in Chapter 10. However, this phenomenon mainly occurs when using relatively small beam widths (compared to the size of the state space), and can therefore be ignored for bigger cases.

[3] The exception to this is when a client has fewer states available than it is told to expand.

which assigns priorities to transitions. A fixed number of outgoing transitions is selected *per state*, meaning that pruning is performed on a state by state basis. This makes an adaptation to a distributed setting straightforward. Since each selection does not consider the outgoing transitions of other states, extra communication with other clients for the pruning phase is not needed. We can take the standard distributed breadth-first state space generation algorithms (Algorithms 4 and 5), and insert an evaluation and selection step at the point where a state is expanded.

Minimal-cost search can be combined with beam search, resulting in g-synchronised beam search, which is presented in Chapter 8 as an instance of G-synchronised beam search, where G can be any reasonable guiding function. Compared to regular beam search, now only states with equal g-values are considered at the same time and states are selected purely on their h-value. This variant not only allows finding minimal-cost solutions within the beam before any other solutions. If one uses additional actions to model costs, it also removes the necessity to store the g-value of every state, since revisiting a state necessarily means having found a less efficient trace compared with a previous trace to the state. Although not worked out explicitly in this chapter, one can imagine combining the techniques of distributed minimal-cost search from Algorithms 23 and 24 with distributed detailed beam search from Algorithms 25 and 26, resulting in a g-synchronised detailed beam search with action-based encoding of costs.

Finally, two other variants are flexible priority beam search and flexible detailed beam search, also introduced in Chapter 8. Flexible priority beam search behaves as regular priority beam search, but at each state it also selects any transition which has the same priority as the least competent member of the usually selected set. This search can be implemented in a distributed setting, since the local view characteristic is not lost. Similarly, in flexible detailed beam search we achieve at each level closure on the worst evaluation value still selected. Distributed detailed beam search, as presented by Algorithms 25 and 26, can be made flexible by redefining some functions. First, we say that function $sel_\gamma(\mathscr{L}_i^j)$ selects at least γ states, where $sel_\gamma(\mathscr{L}_i^j) = [\sigma_0^j] \cup \ldots \cup [\sigma_r^j] \cup [\sigma_{r+1}^j]$, with $r \in \mathbb{N}$ and $r < k-1$, such that $\mid [\sigma_0^j] \cup \ldots \cup [\sigma_r^j] \mid < \gamma$ and $\mid [\sigma_0^j] \cup \ldots \cup [\sigma_r^j] \cup [\sigma_{r+1}^j] \mid \geq \gamma$. Likewise, we redefine $evsel_\beta(E_i) = [e_0] \cup \ldots \cup [e_r] \cup [e_{r+1}]$, with $r \in \mathbb{N}$ and $r < k-1$, such that $T([e_0]) + \ldots + T([e_r]) < \beta$ and $T([e_0]) + \ldots + T([e_{r+1}]) \geq \beta$.

9.5 Related Work

Distributed state space generation has appeared in various forms and in various settings, we will just mention a few here. An early approach, not limited to any specific input language, was proposed by Ciardo et al. (1998). Dill (1996) presents a distributed generation algorithm for the Murϕ verifier. Based on this technique, a distributed UPPAAL has been developed by Behrmann et al. (2000). An implementation of a dis-

Algorithm 25 Distributed detailed beam search - Client Instantiator

```
Require: $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$, heuristic function $h : \mathcal{S} \to \mathbb{K}$, beam width $\beta$, set of goal
    states $\mathcal{G}$, client number $\mathtt{ID}$, set of client numbers $\mathtt{CID}s$, hash function $\# : \mathcal{S} \to \mathtt{CID}s$
  $levelsize \leftarrow | \mathcal{I} |$
  $i \leftarrow 0$
  $\mathcal{L}_i^{\mathtt{ID}} \leftarrow \emptyset$
  for all $s \in \mathcal{I}$ do
    if $\#(s) = \mathtt{ID}$ then
      $\mathcal{L}_i^{\mathtt{ID}} \leftarrow \mathcal{L}_i^{\mathtt{ID}} \cup \{\langle s, 0 \rangle\}$
    end if
  end for
  repeat
    $\mathcal{L}_{i+1}^{\mathtt{ID}} \leftarrow \emptyset$
    if $levelsize > \beta$ then
      while $| \mathcal{L}_i^{\mathtt{ID}} | > \beta$ do
        $\mathcal{L}_i^{\mathtt{ID}} \leftarrow \mathcal{L}_i^{\mathtt{ID}} \setminus \{\langle s, s.g \rangle \in \mathcal{L}_i^{\mathtt{ID}} \mid s = getf_{max}(\mathcal{L}_i^{\mathtt{ID}})\}$
      end while
      $SendToMgrEvalInfo(E_i^{\mathtt{ID}})$, with $E_i^{\mathtt{ID}}$ as Equation 9.2, $sel_\beta(\mathcal{L}_i^j) = \mathcal{L}_i^{\mathtt{ID}}$
      $\gamma_{i,\mathtt{ID}} \leftarrow RecvFromMgrLimit()$
      while $| \mathcal{L}_i^{\mathtt{ID}} | > \gamma_{i,\mathtt{ID}}$ do
        $\mathcal{L}_i^{\mathtt{ID}} \leftarrow \mathcal{L}_i^{\mathtt{ID}} \setminus \{\langle s, s.g \rangle \in \mathcal{L}_i^{\mathtt{ID}} \mid s = getf_{max}(\mathcal{L}_i^{\mathtt{ID}})\}$
      end while
    end if
    $SendToMgrGoalStates(\{s \mid \langle s, s.g \rangle \in \mathcal{L}_i^{\mathtt{ID}}\} \cap \mathcal{G})$
    for all $s \in \mathcal{L}_i^{\mathtt{ID}}$ do
      $\mathcal{L}_{i+1}^{\mathtt{ID}} \leftarrow \mathcal{L}_{i+1}^{\mathtt{ID}} \cup\ nxt_{\mathcal{M}}(s, en_{\mathcal{M}}(s))$
    end for
    $SendToMgrSizeNextLevel(| \{\langle s, s.g \rangle \in \mathcal{L}_{i+1}^{\mathtt{ID}} \mid \neg\exists g' < s.g \in \mathbb{K}.\langle s, g' \rangle \in \mathcal{L}_{i+1}^{\mathtt{ID}}\} |)$
    $i := i + 1$
    $(command, levelsize) \leftarrow RecvFromMgr()$
    if $command \neq \mathtt{finish}$ then
      $SendToClientsNextLevel(\{\langle s, s.h \rangle \in \mathcal{L}_i^{\mathtt{ID}} \mid \neg\exists g' < s.g \in \mathbb{K}.\langle s, g' \rangle \in \mathcal{L}_i^{\mathtt{ID}}\})$
      $\mathcal{L}_i^{\mathtt{ID}} \leftarrow RecvFromClientsNextLevel()$
      $\mathcal{L}_i^{\mathtt{ID}} \leftarrow \{\langle s, s.g \rangle \in \mathcal{L}_i^{\mathtt{ID}} \mid \neg\exists g' \leq s.g \in \mathbb{K}.\langle s, g' \rangle \in \bigcup_{j=0}^{i-1} \mathcal{L}_j^{\mathtt{ID}} \wedge \neg\exists g' < s.g \in \mathbb{K}.\langle s, g' \rangle \in$
      $\mathcal{L}_i^{\mathtt{ID}}\}$
    end if
  until $command = \mathtt{finish}$
  $AnswerTransitionRequestsFromMgr()$
```

Algorithm 26 Distributed detailed beam search - Manager Instantiator

```
Require: $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{C}, \mathcal{T}, \mathcal{I})$, beam width $\beta$, set of client numbers CIDs
Ensure: If found, a trace to a goal state is returned
  $levelsize \leftarrow |\mathcal{I}|$
  $\mathcal{E} \leftarrow \emptyset$
  repeat
    if $levelsize > \beta$ then
      $SendToClientsLimit(calcLimit(RecvFromClientsEvalInfo()))$, see Equation 9.3
    end if
    $levelsize \leftarrow 0$
    for all ID $\in$ CIDs do
      $\mathcal{E} \leftarrow \mathcal{E} \cup RecvFromClientGoalStates(\mathtt{ID})$
    end for
    for all ID $\in$ CIDs do
      $levelsize \leftarrow levelsize + RecvFromClientSizeNextLevel(\mathtt{ID})$
    end for
    if $\mathcal{E} \neq \emptyset$ or $levelsize = 0$ then
      $SendToClients(\mathtt{finish}, 0)$
    else
      $SendToClients(\mathtt{continue}, levelsize)$
    end if
  until $\mathcal{E} \neq \emptyset$ or $levelsize = 0$
  if $\mathcal{E} \neq \emptyset$ then
    return $GeneratePath(\mathcal{E})$
  else
    return $true$
  end if
```

tributed state space exploration algorithm based on the SPIN model checker exists, described by Lerda and Sista (1999). Garavel et al. (2001) present a method to generate state spaces in a distributed way by means of the CADP model checker. All these approaches, however, focus on exhaustive state space generation, and not on heuristically pruning parts of the state space on-the-fly in order to solve a particular kind of problem. Jabbar and Edelkamp (2006) developed a distributed, external version of A^*, thereby combining the fields of distributed, directed and external model checking.

Attempts to create a distributed beam search can be found outside of model checking (Bisiani, 1992). In those settings, one usually works with search trees which have a much smaller average branching factor (the number of outgoing transitions per state) compared to an average state space. Because of this, Bisiani (1992) concludes that small beam widths, usually not bigger than 10, suffice, making a distributed beam search counter-productive due to the communication overhead (a similar result can be found in Section 10.4). In model checking, however, we wish to deal with arbitrary state spaces, where the average branching factor can be much greater, thereby, for bigger instances, making a distributed beam search effective.

9.6 Conclusions

We presented distributed versions of minimal-cost search and detailed beam search. Due to the global view of detailed beam search, designing distributed detailed beam search was non-trivial. Furthermore, we described how the other beam search variants presented in Chapter 8 can be transformed to work in a distributed setting. In the next chapter, the results are reported of a number of experiments with both distributed minimal-cost search and distributed (flexible) detailed beam search.

Chapter 10

Search Experiments in Weighted State Spaces

> *The answer to the ultimate question of Life, the Universe, and Everything is... 42.*
>
> (*Deep Thought*)

10.1 Introduction

SEVERAL SEARCH TECHNIQUES have been described in Chapters 7, 8, and 9, to deal with scheduling problems using model checkers. In the first section of this chapter, we look at a very basic scheduling problem, which has been used already several times in the literature to demonstrate scheduling techniques. We limit ourselves there to demonstrating how to specify the problem and report a minimal-cost solution obtained using minimal-cost search. In the next two sections of this chapter, we present a number of relatively small problems, which nevertheless nicely represent the class of problems these searches are meant to be applied to. In particular, these problems produce state spaces with interesting structures: they contain cycles, deadlocks (meaning unsuccessful terminations of attempts to solve the problem), and confluence of traces (i.e. there are states with multiple incoming transitions). Therefore, these problems show the effectiveness of our techniques to some extent. We describe the problems, and report experimental results, which were obtained by using the μCRL toolset version 2.17.13 (and, in one case, also SPIN version 4.2.7). The results are analysed, and conclusions are drawn.

It should be stressed that the main targets for the search techniques are industrial case studies. Therefore, in Section 10.5, we present an industrial case study of a Clinical Chemical Analyser in detail. For that case, we applied a whole range of searches described in the previous chapters, using the μCRL toolset, to solve problem instances of the general scheduling problem of this machine. From these results, we can draw conclusions on how to schedule the machine in general.

10.2 A Five Tasks Scheduling Problem

In order to facilitate comparison, we will look at the small static scheduling problem originally presented by Niebert et al. (2000), later adapted by Behrmann et al. (2001b). A number of tasks (a_1, a_2, c, b_1, and b_2) need to be performed in a specific order. All tasks need to be performed precisely once, except task c, which can be performed zero or more times. The order is as follows: After task a_1, one should perform a_2, followed by (zero or more times) c. Then task b_1 needs to be executed, finishing with task b_2. The system is free to decide for itself how long it wants to delay after having performed a task. Tasks themselves are considered here to take no time to execute. There are three timing constraints however:

1. The time between execution of a_1 and execution of b_1 should be at least 2 time units;

2. The time between execution of a_2 or the last execution of c and execution of b_1 should be no more than 1 time unit;

3. The time between execution of a_2 and execution of b_2 should be at least 3 time units.

We created a μCRL specification $\mathcal{M} = (\mathcal{D}, \mathcal{F}, \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{I})$ with $\mathbb{N}, \mathbb{T} \in \mathcal{D}$, $+ : \mathbb{T} \times \mathbb{T} \to \mathbb{T}, + : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \in \mathcal{F}$, $a_1, a_2, c, b_1, b_2, \mathit{tick}, \mathit{finished} \in \mathcal{A}$, and $\mathcal{C} = \emptyset$. Furthermore, $\mathcal{P}$ constitutes of a single process S, presented in Figure 10.1. We use three counters, x, y and z, to ensure timing constraints 1, 2 and 3, respectively. The parameter n is used to encode which actions are enabled when. All parameters initially have the value 0, in other words, $\mathcal{I} = S(0,0,0,0)$. Originally, Niebert et al. (2000) presented this problem as a timed automaton. To facilitate comparison, Figure 10.2 displays the problem in this manner.

Using the μCRL toolset, we can search for a minimal-cost (in this case, minimal-time) trace using minimal-cost search in the resulting state space $\mathscr{M}$. This delivers the following trace, where $\mathscr{I} = \{s_0\}$, which takes three time units to execute: $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{\mathit{tick}} s_3 \xrightarrow{c} s_4 \xrightarrow{\mathit{tick}} s_5 \xrightarrow{b_1} s_6 \xrightarrow{\mathit{tick}} s_7 \xrightarrow{b_2} s_8 \xrightarrow{\mathit{finished}} s_9$. It took the state space generator less than three seconds to generate the necessary part of the state space and present a minimal-time trace.

The result is a different one from the one given by Behrmann et al. (2001b), but the execution times of the traces are the same. The only difference is due to the freedom to delay after a task is done. Because of this, there are several minimal-time traces present in the state space.

$$
\begin{aligned}
&S(x:\mathbb{T}, y:\mathbb{T}, z:\mathbb{T}, n:\mathbb{N}) = \\
&\quad a_1 \cdot S(x,y,z,n+1) \triangleleft n = 0 \triangleright \delta + \\
&\quad \mathit{tick} \cdot S(x+1,y,z,n) \triangleleft n = 1 \triangleright \delta + \\
&\quad a_2 \cdot S(x,y,z,n+1) \triangleleft n = 1 \triangleright \delta + \\
&\quad \mathit{tick} \cdot S(x+1,y+1,z+1,n) \triangleleft n = 2 \triangleright \delta + \\
&\quad c \cdot S(x,0,z,n) \triangleleft n = 2 \triangleright \delta + \\
&\quad b_1 \cdot S(x,y,z,n+1) \triangleleft n = 2 \wedge x \geq 2 \wedge y \leq 1 \triangleright \delta + \\
&\quad \mathit{tick} \cdot S(x,y,z+1,n) \triangleleft n = 3 \triangleright \delta + \\
&\quad b_2 \cdot S(x,y,z,n+1) \triangleleft n = 3 \wedge z \geq 3 \triangleright \delta + \\
&\quad \mathit{tick} \cdot S(x,y,z,n) \triangleleft n = 4 \triangleright \delta + \\
&\quad \mathit{finished} \cdot \checkmark \triangleleft n = 4 \triangleright \delta
\end{aligned}
$$

Figure 10.1: A μCRL process describing a five tasks scheduling problem

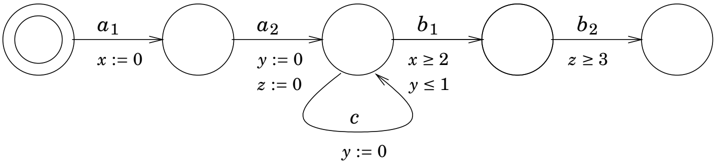

Figure 10.2: A timed automaton describing a five tasks scheduling problem

10.3 Cannibals and Missionaries

In this section, we report our experimental results on solving the *Cannibals and Missionaries* (CM) problem (see, e.g., Lim (1992)), which belongs to the class of *river crossing problems* (Dudeney, 1958). We use a number of μCRL implementations of searches and a SPIN implementation of the depth-first branch-and-bound algorithm to solve this problem. First, we describe the problem. After that, we list the search techniques used and discuss the results, shown in Table 10.1.

10.3.1 Description of the Problem

In the Cannibals and Missionaries problem, C missionaries and C cannibals stand on the left bank of a river that they wish to cross, with $C \in \mathbb{N}$. There is a boat available which can ferry up to B people across ($B \in \mathbb{N}$). The goal is to find a schedule for ferrying all the cannibals and all the missionaries safely across, i.e. the cannibals never outnumber the missionaries, on a shore or in the boat. The boat can only move if it contains at least one person. On top of that we associate costs with moving the boat (1 time unit per passenger), and desire to find a minimal-cost path towards the goal.

10.3.2 Results

Table 10.1 provides all the obtained results. The experiments have been performed on a single machine with a 64 bit Athlon 2.2 GHz CPU and 1 GB RAM, running SUSE Linux 9.2, using both the μCRL toolset version 2.17.13 and SPIN version 4.2.7. Appendix B provides the specifications and the commands used to invoke the searches.

In μCRL, we first applied the minimal-cost search, denoted MCS in Table 10.1. As a reminder, MCS is an exhaustive search method, where the states in the state space are ordered based on the cost needed to reach them from the initial state. This search was used to find the minimum number of time units needed to solve the problem (shown in the *Result* column). The execution times of the searches are displayed in the corresponding *Time* column in the format 'minutes:seconds'.

For comparison reasons, we also performed experiments with SPIN. In those cases, we followed the technique of Ruys (2003), a prominent technique to use heuristics within SPIN. The idea is that the LTL formula that is checked is modified during verification to reflect the best solution found so far. This can effectively implement a BnB mechanism in SPIN, denoted *DFS BnB Prop* in Table 10.1. This algorithm can avoid exhaustive searches, but nevertheless is complete, as is the minimal-cost search used for μCRL. In these experiments, the LTL property is $\Diamond(q \geq U)$ (with $\Diamond$ the eventuality operator), where q denotes the total cost of a trace and U denotes an upper bound on this cost, provided by the user. In the standard depth-first search, U is fixed during the search, while in the BnB search C-code in the specification updates U to the best (smallest) total cost found so far in the search. The search through a trace stops only when either the property does not hold or a deadlock state is found. In the latter case, we have slightly adjusted the technique of Ruys (2003) to deal with unsuccessful terminations: U is updated only on successful terminations. Since μCRL and SPIN seem to count states in different ways, we remark that the numbers of states of the experiments using different tools cannot fairly be compared.

A problem in setting up the SPIN experiments is choosing U. Apparently a very high U has a negative effect on the search performance. Choosing it too high can happen easily. Here we always chose U fairly close to the final result, but not so close, that updating it for every case was necessary. For the cases up and including $(50, 20)$, we took $U = 200$, after that switching to $U = 300$. The $(300, 10)$ case with $U = 1000$ was the last case we could actually study, i.e. SPIN could not deal with bigger cases. As the experiments showed that choosing U has a big effect on the efficiency of the SPIN searches, one approach can be to use the result of a beam search as the initial value of U. Further integration of these techniques remains to be investigated.

Contrary to the results of Ruys (2003), here the depth-first BnB search technique does not prune much, compared to standard depth-first search. This can be due to the differences in the nature of the problems that are studied. We believe that in this experiment there are many more possible schedules with very long traces, in comparison to the Travelling Salesman Problem, analysed by Ruys (2003). The fact that the first successful termination in general was found quite late in the search, means that pruning could only be started at a fairly late stage. Before that stage, of course, depth-first BnB search searched exactly the same amount of states as depth-first search. It remains to be investigated, whether compression techniques or approximation techniques such as bit-state hashing have a positive effect on the search capabilities in this specific case.

As a final step, we used g-synchronised flexible detailed beam search (g-SFDBS), which is a combination of the techniques proposed in Chapter 8, with $h(s) = C(s) + M(s) + (\langle C(s) \neq M(s) \rangle \times (2 \times C))$ as the heuristic part of the search, where $C(s)$ and $M(s)$ are the numbers of cannibals and missionaries on the left bank in state s, respectively, C is the total number of cannibals (or missionaries) in the problem, and $\langle C(s) \neq M(s) \rangle$ equals 1 if $C(s) \neq M(s)$, and 0 otherwise. The intuition behind this heuristic is that, first, we want to minimise $C(s)$ and $M(s)$, hence the first part of the function. Second, we support having an equal number of cannibals and missionaries on the left bank as an easy way to avoid deadlock states where $C(s) > M(s)$, hence putting an extra penalty on such states.

Our experiments showed that in practice there are so many unsuccessful termination states in the specification that some deadlock avoidance in the heuristic function is needed. Without it, we often experienced unsuccessful searches in which the entire beam got trapped in deadlocks. With deadlock avoidance, flexible beam search proved to be applicable using a fairly stable beam width of 20, partially showing the suitability of the heuristics used. In Table 10.1, the T column under g-SFDBS shows the minimum number of time units needed to solve the problem approximated by this search. The results show an example of what can be achieved when near-optimal solutions are acceptable, i.e. when we give up completeness.

Let us take a closer look at a particular problem instance, using the 3D interactive state space visualisation tool LTSVIEW (see bibliography). Figure 10.3 shows us, on the left side, the complete state space of the (50,10) instance of the Cannibals and Missionaries problem, in other words, the case where there are 50 missionaries and 50 cannibals, and the boat can contain up to 10 people. The initial state is at the top of this structure. As it turns out, there is exactly one state representing successful termination, therefore all possible successful traces end up in this state. The state is situated in the small cone near the bottom of the image, in the center of the black circle. When we search the state space with minimal-cost search, which is situated on the right of the figure, we observe that everything needs to be searched, except for the part at the bottom, which is at a greater depth than the cone containing the goal state.

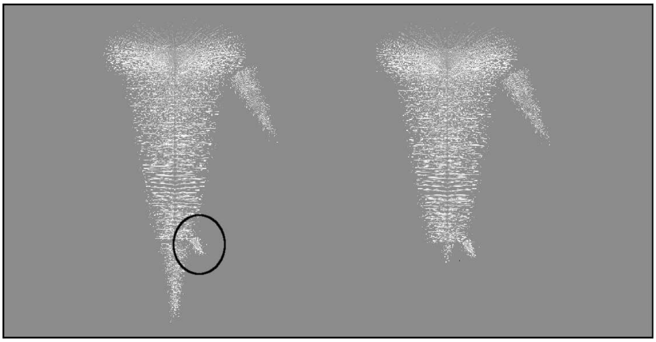

Figure 10.3: Breadth-first and minimal-cost search of the (50,10) CM problem

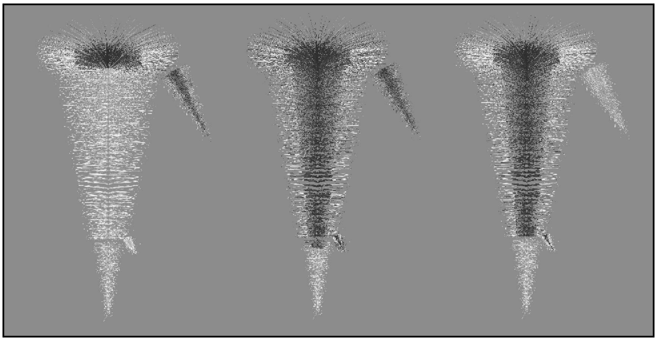

Figure 10.4: g-Synchronised detailed beam search of the (50,10) CM problem: 1. $\beta = 10$ without deadlock avoidance; 2. $\beta = 100$ without deadlock avoidance; 3. $\beta = 10$ with deadlock avoidance

If we consider the same problem instance using beam search, we get the results shown in Figure 10.4. The importance of including some notion of deadlock avoidance in the estimation function becomes apparent here. On the left, we see the parts of the state space, which are searched using a g-synchronised detailed beam search with $\beta = 10$ and $h(s) = C(s) + M(s)$, displayed in dark grey. It is particularly interesting to note that, using this estimation function, the search quickly gets trapped in the 'deadlock cone' at the top, i.e. a part of the state space, which only leads to failure. We can compensate for this behaviour, by increasing the beam width. If we take $\beta = 100$, we find the goal state. This case is shown in the middle of the figure. In this case, however, it should be clear that we are not so successful at pruning. Besides, as it turns out in practice, the beam width needs to be increased in size considerably when dealing with increasingly big problem instances. On the right, we can see a g-synchronised detailed beam search with $\beta = 10$ and $h(s) = C(s) + M(s) + (\langle C(s) \neq M(s) \rangle \times (2 \times C))$. Now, with some form of deadlock avoidance in the estimation function, we are able to find the goal state with a beam width of 10, and furthermore are able to completely avoid the deadlock cone.

Our g-SFDBS algorithm should ideally be compared with other heuristic state space generation tools, such as HSF-SPIN, which is SPIN augmented by Edelkamp et al. (2001b) with A^* and greedy best-first search. We, however, leave this as future work.

Table 10.1: Cannibals and Missionaries experimental results. n.s.: no solution exists (Lim, 1992); o.o.t.: out of time (set to 12 hours); o.o.m.: out of memory (set to 900 MB)

Problem	Result	μCRL MCS		μCRL g-SFDBS				SPIN DFS		SPIN DFS BnB Prop.	
(C,B)	T	# States	Time	T	β	# States	Time	# States	Time	# States	Time
(3,2)	18	147	00:03.80	18	3	142	00:03.73	28,535	00:00.32	26,062	00:00.29
(10,3)	n.s.	396	00:03.94	n.s.	10	396	00:03.90	76,432	00:00.42	76,432	00:00.41
(10,4)	44	1,378	00:04.38	46	10	1,129	00:04.42	253,815	00:01.60	251,400	00:01.53
(20,4)	104	2,537	00:05.32	106	10	2,191	00:05.38	445,801	00:02.66	408,053	00:02.34
(50,10)	142	25,868	00:11.22	148	10	8,035	00:08.47	3,703,900	00:27.33	3,472,070	00:23.69
(50,20)	116	90,355	00:20.15	120	15	17,361	00:11.45	12,647,000	02:05.25	12,060,300	01:49.59
(100,10)	292	49,141	00:19.65	296	10	16,274	00:14.46	14,709,600	02:49.32	13,849,300	02:23.34
(100,30)	222	366,608	01:05.79	228	15	61,380	00:32.06	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(300,10)	892	143,549	01:01.94	896	10	49,514	00:48.47	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(300,30)	680	1,008,436	04:10.72	684	15	205,556	02:30.11	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(500,50)	1,076	4,365,536	21:40.52	1,080	20	685,293	10:33.28	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(500,100)	1,036	17,248,979	77:16.36	1,040	20	1,170,242	16:47.10	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(1000,50)	2,160	8,551,996	70:00.14	2,168	20	1,397,100	37:02.03	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(1000,250)	o.o.t.	o.o.t.	o.o.t.	2,032	20	5,317,561	240:22.11	o.o.m.	o.o.m.	o.o.m.	o.o.m.

10.4 The Zebra Finch Problem

Next, we look at some experimental results of attempts to solve instances of what we call the *Zebra Finch problem*. We based this problem on a combination of several river crossing problems, such as *five jealous husbands* and *soldiers and children* (Dudeney, 1958). First, we describe the problem, and then we provide the results obtained using the techniques described in earlier chapters.

10.4.1 Description of the Problem

Zebra Finches, *Taeniopygia guttata* (Vieillot, 1817), are small birds living in Central Australia (Zann, 1996). They are found in large colonies of pairs inhabiting open steppes with scattered bushes and trees. These birds can react aggressively towards each other, for instance when a jealous male bird tries to keep other male birds away from his mate. When young birds reach an age where they can live outside the nest, they are quickly adopted by the group.

Figure 10.5: A pair of Zebra Finches

We consider a group consisting of n pairs and m young, sitting in a tree on an open steppe. They want to migrate to some bushes up ahead, but they have to travel in smaller groups, since there are some hawks flying in the distance, which can spot a group of more than k adult finches. Once a group has reached the bushes, at least one of the Zebra Finches needs to fly back, in order to signal that a new group can travel. On top of this there are two other conditions:

1. Considering the jealous nature of the male Zebra Finches, no female finch may ever be either in the tree, the travelling group or the bushes in the presence of other male birds, unless her partner is also present.
2. The young in the colony have to be guided by at least one adult finch, so the travelling group cannot consist of only young finches. In limiting the group size, two young are equivalent to one adult.

Finally, some costs are related to the travelling from tree to bushes and back:

- A group consisting of only adults needs 1 time unit to travel the distance, independent of the size of the group;
- If the number of young in the group does not exceed the number of adults, the time needed to travel is 2 time units (each adult needs to take care of at most one young);
- When, in the group, the number of young exceeds the number of adults, the travel takes 3 time units, since at least one adult takes care of more than one young.

We specify the problem allowing all possible actions at all times. It demonstrates the techniques' ability to deal with arbitrary state spaces; problem instances lead to state spaces containing both cycles (while forming the group and when birds fly away and back again), and deadlocks (violations of the 'jealous male' condition).

10.4.2 Results

In Table 10.2, we present some results we found for instances of the Zebra Finch problem. We used minimal-cost search, g-synchronised detailed beam search (g-SDBS), and its flexible variant (g-SFDBS), where for the last two cases we defined $h(s)$ for each state s as the number of finches still in the tree, thereby encouraging fast removal and discouraging the returning of finches. Problem instances are described by providing n, m and k. For each search, the total execution time of the result found is given as T. Furthermore, the number of states searched to find the solution is provided and the time needed to find it is displayed in the format 'hours:minutes:seconds'. Searches not performed are marked with hyphens, and when a search is done in a distributed setting, an asterisk is placed after the number of states. Sequential searches were performed using a machine with a 64bit Athlon 2.2Ghz processor, 1 GB of memory and running Suse 9.3, while 16 of these machines together performed the distributed searches.

The minimal-cost search tells us that as the problem instances get bigger, the state spaces grow very rapidly. The beam searches on the other hand show a much nicer increase in states from instance to instance. Looking at the (50,50,10) instance though, we see an unwanted effect in the regular g-synchronised detailed beam search, already briefly referred to in Section 9.3, namely that increasing β not necessarily means getting a better result. The main cause for this is tie-breaking, i.e. the fact that pruning is sometimes not being done only based on f, but also on other criteria, simply because more than β states turn out to be promising enough. Although this mainly has a noticeable effect in smaller instances, it is undesired and does not occur in its flexible variant. Because of this, the flexible search provides better insight into the effectiveness of the estimation function used.

Furthermore, it is interesting to note that for smaller instances, the distributed algorithm performs worse than the sequential version, which can be seen in the (50,50,20) case, where we performed both a sequential and a distributed search. The $\mathscr{L}_i$ sets in the state space are all relatively small, making the communication overhead of the distributed algorithm noticeable. This seems to be directly related to the argument found in the literature, for instance by Bisiani (1992), against distributed beam search in a more traditional setting, mentioned in more detail in Section 9.5. Besides that, note that the result obtained with the distributed search is better than the one of the sequential search, even though the beam widths are equal. This, again, is due to tie-breaking, which, in a distributed environment, can happen at multiple places in a single level, instead of only at one point. In the flexible search, where tie-breaking is avoided altogether, this behaviour does not appear.

The (100,100,50) and the (100,100,80) case have a big difference in execution time, while the number of states in the latter case is even smaller. However, although the number of expanded states is smaller in the (100,100,80) case, the number of encountered and evaluated states is much greater. This is directly related to the maximum size of the travelling group k.

Finally, as stated earlier in Section 8.3.4, for the flexible search, overall β is more stable compared to the non-flexible search. Due to this, in the two biggest cases, the flexible search is even more efficient than the non-flexible one. The stability of β means in general that, given some search results, it is easier to determine β for a new flexible search than for a new non-flexible one.

Table 10.2: Zebra Finch problem experimental results

Instance			μCRL MCS			μCRL g-SDBS				μCRL g-SFDBS			
n	m	k	T	# States	Time	β	T	# States	Time	β	T	# States	Time
10	5	5	19	228,737	00:00:29	400	19	58,272	00:00:14	400	19	67,804	00:00:18
10	10	5	21	513,123	00:01:07	400	21	65,605	00:00:18	400	21	85,633	00:00:24
10	10	8	10	2,020,061	00:04:28	450	10	48,669	00:00:19	400	10	69,550	00:00:21
50	50	5	121	18,157,429	00:48:13	1,000	121	641,315	00:04:49	400	121	298,065	00:02:31
50	50	10	41	475,744,120 *	05:13:26	1,000	43	637,285	00:07:28	1,000	41	702,844	00:13:10
50	50	10	-	-	-	1,500	44	946,660	00:13:37	1,500	41	1,088,042	00:20:09
50	50	10	-	-	-	4,000	43	2,560,051	00:46:22	4,000	41	2,881,512	00:51:28
50	50	20	-	-	-	5,000	24	3,478,600	01:14:00	1,500	22	1,521,829	00:54:07
50	50	20	-	-	-	5,000	20	3,095,782 *	02:01:05	4,000	20	2,579,479 *	01:48:16
100	100	10	-	-	-	5,000	87	6,009,134 *	01:39:52	4,000	87	5,318,589 *	06:22:54
100	100	20	-	-	-	5,000	41	5,884,895 *	00:42:48	4,000	42	5,433,733 *	04:02:26
100	100	50	-	-	-	20,000	17	27,366,213 *	02:57:21	4,000	18	41,611,293 *	06:16:29
100	100	80	-	-	-	20,000	10	24,796,756 *	25:31:24	10,000	9	16,044,286 *	05:05:49
200	200	50	-	-	-	50,000	35	135,964,662 *	37:25:42	6,000	35	27,324,012 *	08:32:35

10.5 A Clinical Chemical Analyser

10.5.1 Introduction

In this section, we describe and analyse an industrial case study of a Clinical Chemical Analyser. The Clinical Chemical Analyser (CCA) is used to automatically analyse patient samples (blood, plasma or urine). TNO Industry, in cooperation with the Eindhoven University of Technology (TU/e), has been involved in the redesign of the CCA. The project charter was originally drawn up by Vital Scientific, a customer of TNO, to examine the possibility of a 100% throughput increase.

At TU/e, several projects have been devoted to the CCA. First, the basic outline for the hardware was explored by Vervoort (1999), while, in a parallel project, the scheduler was developed by Spronk (1999). Then, the hardware for a CCA mock-up was designed by Hesen (2000). Currently, a new scheduler is being designed by Weber (2003). The fact that a schedule providing optimal performance of the CCA still has not been found raised the idea to look at this problem using a modelling language.

The section is set up as follows: First, we give an introduction to the CCA. After that, we discuss the CCA μCRL specifications we used for the CCA case study, followed by the results obtained by applying the sequential and distributed implementations of minimal-cost search to instances of the general scheduling problem of the CCA. Following, we conducted more experiments, applying the implementations of some of the beam search variants, presented in Chapter 8, on the CCA specifications. Finally, we compare the experimental results and draw conclusions.

10.5.2 Description of the Problem

What follows is a description of the scaled-down CCA as we used it for the research described in this section. Note that this is based on the design as given to us by mechanical engineers. Improving the design is regarded outside the scope of this work.

Figure 10.6 shows the setup of the CCA; there is a cuvette rotor containing 11 cuvettes, which are indexed from 0 to 10 counter-clockwise (this in contrast with both the CCA mock-up, which has 45 cuvettes, and the real CCA, which has 120 cuvettes). There are three cranks, which are able to perform actions on these cuvettes: The reagent crank can add a reagent from the reagent rotor to a cuvette, the sample crank can add a patient sample from the sample rotor to a cuvette, and the emptying crank can empty a cuvette. Besides that there is a mixing crank, but it is unimportant for the scheduling problem, which will become clear later on.

The use of the machine is to process test recipes. Each available patient sample should be processed according to one of three possible test recipes.

In Table 10.3, the three recipes are depicted. In recipe 1, first a reagent (R_1), and later a sample (S) is added to a cuvette. After that, the cuvette is emptied (E). Recipe 2 is an extension of recipe 1 in the sense that after having added a sample to the cuvette a

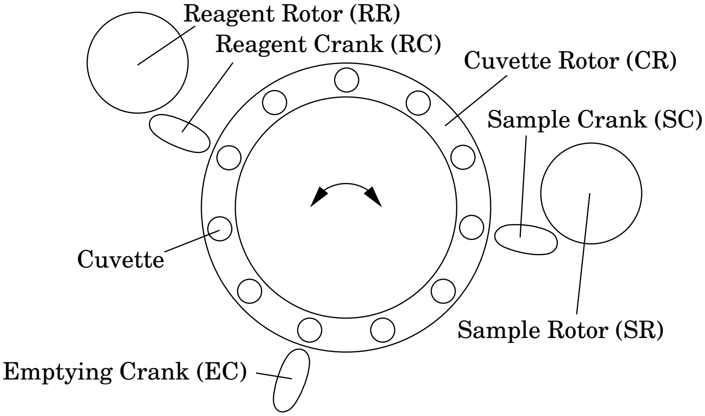

Figure 10.6: The scaled-down CCA

Table 10.3: Recipes for the CCA

Description	Recipe
1-reagent	$R_1 \to \Delta t_1 \to S \to \Delta t_2 \to E$
2-reagent	$R_1 \to \Delta t_1 \to S \to \Delta t_3 \to R_2 \to \Delta t_4 \to E$
3-reagent	$R_1 \to \Delta t_1 \to S \to \Delta t_5 \to R_2 \to \Delta t_6 \to R_3 \to \Delta t_7 \to E$

second reagent (R_2) must be added. Finally, recipe 3 requires even a third reagent (R_3) to be added to the cuvette. This adding of fluids cannot be done at any time however. The Δ occurrences in Table 10.3 represent delays of certain lengths (measured in time units). The values of $t_1, ..., t_7$ are limited to the following possibilities: $t_1 \geq 15, t_2 \leq 105, 3 \leq t_3 \leq 27, t_4 \leq 105 - t_3, 6 \leq t_5 \leq 21, 9 \leq t_6 \leq 42, t_7 \leq 105 - t_5 - t_6$.[1]

The CCA consists of a number of independently working parts (cranks and rotors) which have to be controlled using a set of low-level actions. In order to avoid problems, these actions are used as the building blocks for higher level instructions, so-called *operations*. Careful design of the operations has led to the property, that no errors occur within them. These are the operations available:

- $R_i(j)$: Reagent i of a test is added to cuvette j;
- $S(i)$: The sample for cuvette i is added;

[1] A time unit in the scaled-down CCA specification corresponds with a duration of 4 seconds in the actual CCA.

- $E(i)$: Cuvette i is emptied.

Finally, a number of operations together form a *cycle*, which is the basic building block for a schedule. There are three types of cycles, the 12, 16 and 24-cycles, differing in the number of time units they require for execution. In the 12-cycles round 1 of operations occurs, in the 16-cycles rounds 1 and 2 occur, and in the 24-cycles all three rounds occur. The rounds being (in this order):

1. Given an empty cuvette i, the first reagent of a test can be added to this cuvette. At the same time, if possible, the sample for the test in cuvette $i-5$ can be added. Finally, also at the same time, if cuvette $i+3$ contains a finished test, the cuvette can be emptied.

2. If a cuvette j ($i \neq j$) is ready to receive a second or a third reagent, this reagent can be added.

3. If a cuvette k ($i \neq k, j \neq k$) is ready to receive a third reagent, this reagent can be added.

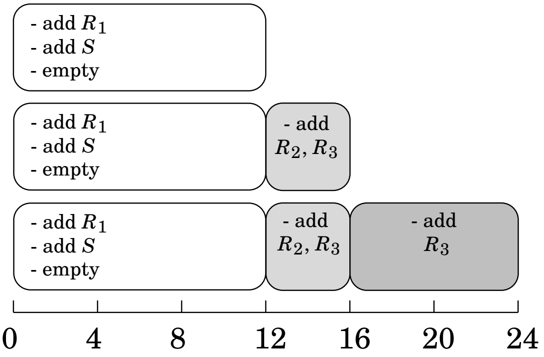

Figure 10.7: The 12, 16, and 24-cycles for the CCA

In Figure 10.7, the three types of cycles are visualised. All of them start with round 1, where the available operations (listed using hyphens) can be performed in parallel. After that, in the case of 16 and 24-cycles, a second round is entered. In 24-cycles even a third round appears. This mandatory ordering in rounds means that even in a cycle, in which only a second and/or a third reagent is added, round 1 appears, even though no operation (or only an empty operation) is performed in this round.

The cycles can be named by listing the operations that occur in each round. We do not list the E operations though, since emptying cuvettes is done whenever possible. For instance, in the 12-cycle $R_1(i)$, round 1 from the list above is carried out without adding a sample. When rounds 2 and 3 occur in a cycle, it will always be after having done round 1. Also for these rounds the necessary cuvette indices are given. For instance, cycle $R_1SR_2(i,j)$ first performs round 1, with a first reagent being added to cuvette i and a sample being added at the same time to cuvette $i-5$, after which a second reagent is added to cuvette j in round 2. In the real machine it happens to be the case that there is no cycle which only empties a cuvette. This is important to know when looking at the results of the case study, in particular Section 10.5.8.

It was previously mentioned that there is a mixing crank. Mixing should happen every time an extra fluid is added to a cuvette. This, however, is not part of the scheduling problem, because mixing is done within the operations.

The scheduling problem is now the following: given a batch of tests to be processed, provide a sequence of cycles that enables the CCA to process the tests in the minimum time possible.

10.5.3 Creating the Specification of the CCA

For the scheduling problem of the CCA, it is not necessary to specify all the parts of the machine at a very detailed level. It suffices to concentrate on a process which allows every valid sequence of cycle commands to happen. Invalid sequences would consist of cycles applied to inappropriate cuvettes or cycles applied too soon or too late. It has to be stressed that we therefore incorporate explicitly the timing constraints, as seen in Section 10.5.2, in the specification.

When designing, it is important to choose the parameters in a smart way. The more information you store, the larger the resulting state space will be, therefore any unnecessary information must be avoided. We decide not to use test IDs; to solve the problem we do not need to link an individual sample with some particular reagents. We can assume that the reagent and sample rotors provide the right reagents and samples when required. Furthermore the number of samples and second and third reagents that still need to be added is not needed; it is clear what must be added when looking at the rotor and the number of unprocessed first reagents. That leaves us with the following:

- The cuvette list, consisting of 11 tuples. Each tuple stores which fluids are currently in the corresponding cuvette, which type of test is in the cuvette, and how much time is left before a new fluid may be added.
- How many 1-reagent tests should still be started.
- How many 2-reagent tests should still be started.
- How many 3-reagent tests should still be started.

When specifying, it becomes clear how convenient the use of abstract data types is. The rotor can be specified using a specially taylored list data type, and we can define functions to quickly check the status of the rotor (e.g. whether there are any tests ready to receive a sample, or whether a certain test is finished). This makes working with complex data structures easier.

We decide to build the specification in an incremental way; first, we build a specification dealing only with 1-reagent tests and 12-cycles. It consists of a single process which has the 12-cycles as actions, together with the necessary guards and recursive calls, placed in alternative composition. The guards are there to check whether a chosen cuvette is indeed ready to receive a certain fluid and whether the timing constraints are met. Note that it is not necessary to keep track of the overall execution time in this specification, as each action requires a delay of three time units; in such a case a minimal-time trace in a state space is also the shortest trace. Therefore we can do a normal breadth-first search for the *finished* action.

Using the specification in practice, though, on a number of test batches, we find that the freedom to place new tests anywhere on the rotor leads to a state space explosion. Therefore, we decide to build a second specification allowing new tests to be placed only in the next empty cuvette, looking counter-clockwise. Since the cranks are placed in such a way that, rotating one cuvette at a time, a sample can be added to a cuvette the moment it reaches the sample crank, this restriction will not lead to a suboptimal solution. In fact, Section 10.5.4 shows that this is indeed the case, for a test batch of five products.

Next, we build a third specification with a process using all possible cycles together with the necessary guards, placed in alternative composition. We also use this specification to find schedules for different test batches. The results can be found in the following sections. After that, we create a fourth specification, which is much more restricted in its possibilities; we put a strategy in it to cope with a batch of tests. We attach priorities to cycles, such that the specification will always execute the enabled cycle with the highest priority. In short, the strategy is to always perform as many operations in parallel as possible and to get the first reagents of the tests as quickly as possible on the rotor. Using the same batches of tests as input for this specification, we get the same results as we get using the strategy-free specification (in cases where the complete state space of the latter specification can be generated at least). This tells us that the strategy used in the strategy specification is a good one for the test batches used.

The distributed state space generator of the μCRL toolset makes it possible to generate state spaces using a cluster of computers. In this case study, it becomes clear quite soon that an increase of the size of the test batch results in a big growth of the state spaces of most of the specifications. For some of the test batches a minimal-time trace cannot be found without distributed state space generation.

10.5.4 Results Using Exhaustive State Space Search

Tables 10.4 and 10.5 summarise our findings when applying exhaustive breadth-first search. We used the sequential implementation for the small cases, and switched to the distributed implementation for the bigger cases (indicated with an asterisk). Table 10.4 considers the simpler case where all test batches consist of a number of 1-reagent tests. In this setting, only 12-cycles are needed. In Table 10.5, all cycles are incorporated. In both cases, we consider the specification with and without a built-in strategy.

The tables should be read as follows: In every row, a test batch is specified. In Table 10.4, the number of tests is displayed, in Table 10.5, the descriptions are of the form (a, b, c), where a, b and c indicate the number of 1-reagent, 2-reagent and 3-reagent tests, respectively. The results are in the following format: r/s, where r and s equal the number of time units and the number of cycles in the minimal-time trace, respectively. Searches not performed are marked with hyphens. Also, the number of states in the different state spaces is given. Finally, the time needed to find the results is given in the format 'minutes:seconds'.

From the numbers, it is clear that the state spaces grow rapidly in size when using bigger test batches. In the specifications without a strategy this is due to the fact that from every state the system can do any of the valid actions. In Table 10.4, in case of the 12-cycles specification, the size is increasing so rapidly, that already with 10 tests we had to conclude this would not be promising to continue. The restricted specification was sufficient for us to find minimal-time traces for all configurations.

Table 10.4: Exhaustive search results for the CCA with only 12-cycles

Spec. **# Tests**	12-cycles	# States	12-cycles restr.	# States
5	30/10	416,352 *	30/10	447
10	-	-	45/15	9,878
15	-	-	60/20	528,699
20	-	-	75/25	8,403,885
30	-	-	105/35	222,613,811 *

Table 10.5 contains the results we obtained when using specifications with the three types of tests. When using 10 tests, we are not able to get minimal-time traces anymore using the general specification. Although generating the state spaces takes a lot of time and effort, it is still possible. The problem is the fact that CADP, which is used to obtain minimal-time traces from the state spaces, needs the chunks of the state space, obtained from a distributed state space generation, to be merged into a single state space, since it only works sequentially at the moment. In the (6,2,2) test batch, the resulting state space takes about 30 Gigabytes of disk space, and is too big to handle

Table 10.5: Exhaustive search results for the CCA

Spec. **# Tests**	All cycles	# States	Runtime	Strategy	# States	Runtime
(3,1,1)	36/11	1,148	00:07.41	36/11	222	00:02.64
(1,3,1)	39/11	5,352	00:27.50	39/11	290	00:02.84
(1,1,3)	45/12	16,380	01:16.99	45/12	273	00:02.84
(6,2,2)	-	-	-	51/15	11,477	00:44.92
(3,5,2)	-	-	-	55/15	29,929	01:56.82
(1,2,7)	-	-	-	73/17	23,895	01:34.84
(7,4,4)	-	-	-	75/21	5,300,625 *	83:48.21
(4,8,3)	-	-	-	77/21	3,959,283 *	63:31.45
(2,5,8)	-	-	-	91/22	1,951,446 *	31:37:53

afterwards. In the strategy specification, the size increase is mainly due to the nondeterminism concerning adding new tests (more precisely, deciding which test type should be added at which point). One can therefore decide to create another strategy specification, which applies a fixed order of tests concerning their type (i.e. first adding 3-reagent tests).

10.5.5 Results Using On-the-fly Searching

We also used minimal-cost search to find minimal-time traces for the strategy specification, using five and ten products (in the varying type combinations). Table 10.6 contains the results of these tests. Please note that the number of states in this table cannot be straightforwardly compared to the numbers in Tables 10.4 and 10.5. This is because for the on-the-fly searching we added the necessary *tick* actions to the specification, resulting in more states in the state spaces.

In the cases of five products, we find that the state spaces still need to be generated almost completely in order to find the solutions. When moving to bigger test configurations though, the payoff becomes considerate; in the (6,2,2) test batch, a minimal-time trace can be found halfway through the state space generation.

The results of using minimal-cost search are twofold: on the one hand, we are able to find minimal-time traces with less effort; more specifically, since we can find these traces on-the-fly, merging the state space chunks into a single state space and subsequently searching for a specific action using CADP can be avoided. This already saves us a lot of time. On the other hand, it still proves very difficult to get results for bigger test configurations, as seen in Table 10.6. The state space for the (6,2,2) test batch is very big and takes hours to generate. It has to be said that, although difficult, getting

Table 10.6: Minimal-cost search results for the CCA

Spec. **# Tests**	All cycles	# States	Runtime
(3,1,1)	36/11	3,375 (of 4,001)	00:10.35
(1,3,1)	39/11	13,194 (of 15,091)	00:30.48
(1,1,3)	45/12	34,142 (of 39,132)	01:10.97
(6,2,2)	51/15	341,704,322 * (of 677,470,840)	1524:56.00
(3,5,2)	-	-	-
(1,2,7)	-	-	-
(7,4,4)	-	-	-
(4,8,3)	-	-	-
(2,5,8)	-	-	-

a minimal-time trace is only possible using on-the-fly searching, since converting the state space after generation and then searching the state space using a model checker provides technical difficulties (as mentioned earlier in Section 10.5.4). For bigger test configurations, we are currently unable to find minimal-time traces, since we encounter technical bottlenecks, such as the speed of communication between the computers in the cluster we use. Other problems stem from this particular case study and specification, not from the search algorithm.

10.5.6 Results Using Beam Search

Applying detailed and priority beam search to the CCA case study proves to be very fruitful. It is possible to prune away traces, which are not promising, very effectively, and it turns out to be very interesting to try and see how much can be pruned without removing all optimal solutions. Of course, one can only know if all optimal solutions are pruned if the total cost of these solutions is known. Using previous results (Tables 10.4, 10.5 and 10.6), the beam widths needed to get optimal solutions can be determined for those particular problem instances. These beam width values provide an indication of how big the beam widths will have to be for even bigger instances.

In Table 10.7, the results are given which are obtained using a g-synchronised detailed beam search through the state spaces. The estimation function h we use counts the number of fluids that still have to be added to the rotor. Worst case, a given partial schedule can always be extended using n cycles, where n is the remaining number of fluids. Note that, in order to use this function, we have to add an extra parameter to the specification described in Section 10.5.3, to be able to keep track of the total number of fluids left.

As can be seen, almost at all times, we are able to deal with the test batches using a standalone computer. Notice that in the first number of configurations we are able to provide the number of states in the complete state space, thereby showing how much we can prune. As is shown with the (6,2,2) configuration, the number of pruned states can become considerate, in this particular case more than 99.9% of the state space. Looking at the results, we see that the needed beam width differs from test to test. This makes it hard to predict the needed beam width for larger test configurations. The larger you choose the beam width, the higher the probability that the solution found is a minimal-time trace, so when choosing a beam width value, one should determine how much time and effort is reasonable to put into finding a solution.

The beam width is not growing in relation to the number of fluids in a test configuration. Probably this is due to the ordering of states while searching. Sometimes the generator is forced to make some selections which are not based on the evaluation values of the states (tie-breaking), due to the hard limit of states per level set by the beam width. In those cases, the order in which the states are encountered plays a role.

The execution times become very long already when dealing with 10 tests, no doubt because of the evaluation procedure. It seems interesting to try to optimise this procedure in the future, since a lot of time could be gained then.

Table 10.8 shows us results obtained by performing a g-synchronised priority beam search. Again, here we were able to find solutions for the test configurations using a standalone computer. The used priority function *prio* stimulates to perform as many operations in parallel as possible. To facilitate comparison, we searched for solutions for all the test configurations with $\alpha, l = 1$, a search which could in fact be called g-synchronised heuristic breadth-first search, and, in most cases, with $\alpha, l > 1$. This shows the effect of raising the widening factor and choosing the stabilisation level further down the state space.

Finally, in Table 10.9, our experiences with flexible priority beam search are shown. The execution times of searches applied on batches up to 10 tests are very promising. The major advantage of flexible beam search is that determining the beam width for each individual configuration is no longer an issue. In all the cases the beam width is initially set to 1, i.e. $\alpha^l = 1$, and is increased automatically where needed during exploration. When dealing with batches bigger than 10 tests, we see that the execution time and the number of states searched rapidly increase. This shows the drawback of a flexible search: it avoids tie-breaking, as mentioned already several times in this chapter, but the result of this is that the space and computation time requirements are no longer linear to the maximum search depth.

Note that we do not conduct any tests using flexible detailed beam search. Although we have implemented it in the toolset, we do not think that, in the CCA case study, it will show a much better performance than detailed beam search. More on this is mentioned in Section 10.5.7.

Table 10.7: *g*-Synchronised detailed beam search results for the CCA

Spec. # Tests	All cycles	β	# States	Runtime
(3,1,1)	36/11	25	1,461 (of 4,001)	00:03.43
(1,3,1)	39/11	41	2,234 (of 15,091)	00:03.93
(1,1,3)	45/12	19	1,598 (of 39,132)	00:03.46
(6,2,2)	51/15	81	7,408 (of 677,470,840)	00:07.76
(3,5,2)	55/15	765	67,470	00:49.45
(1,2,7)	73/17	75,000	6,708,705	84:38.41
(7,4,4)	75/21	35,000	3,801,607	41:01.80
(4,8,3)	77/21	50,000	5,837,325	85:41.60
(2,5,8)	-	-	-	-

10.5.7 Comparisons

Taking a closer look at the minimal-time traces found, we conclude the following: Concerning the 12-cycles specifications, the minimal-time traces are straightforward. The first five reagents need to be added without adding a sample, because of the incubation times. After that, a reagent can be added together with a sample, until there are no reagents left to add and the final five samples can be added. Having a batch of i products will therefore lead to a minimal-time trace of $i+5$ cycles, which will take $3\times(i+5)$ time units, since every cycle takes three time units.

For the more general case, using 12, 16, and 24-cycles, it is more difficult to observe a pattern, though. There does not seem to be any advantage gained by adding the reagents for the different kinds of tests in a certain order (for instance first adding all the reagents for the 3-reagent tests). Besides that, there does not have to be any pattern shared by the particular minimal-time traces found here; it could very well be the case that there are several minimal-time traces coexisting in the same state space. We only get to see one though, which shows a possible solution, not necessarily a mandatory one.

Next, we compare the results of the different search techniques used. The first observation is, that when analysing the results of Table 10.5, the chosen strategy seems to be a good one, at least for the test configurations we used. Therefore, it seems to be a good approach to try to put the first reagents of tests as quickly as possible on the rotor and to try to do as much as possible in each cycle.

Table 10.6 tells us that for the smaller configurations (5 tests) the minimal-time traces present are not much shorter than the longest traces in the state spaces. We get this from the fact that only a small part of each state space is left unexplored when finding a minimal-time trace. An explanation for this may be the fact that with 5 tests,

Table 10.8: g-Synchronised priority beam search results for the CCA

Spec. # Tests	All cycles	α	l	# States	Runtime
(3,1,1)	37/12	1	1	48 (of 4,001)	00:03.03
(3,1,1)	36/11	2	5	179 (of 4,001)	00:03.52
(1,3,1)	39/11	1	1	50 (of 15,091)	00:03.08
(1,1,3)	45/12	1	1	57 (of 39,132)	00:03.08
(6,2,2)	52/16	1	1	67 (of 677,470,840)	00:02.63
(6,2,2)	51/15	2	9	479 (of 677,470,840)	00:03.06
(3,5,2)	58/18	1	1	74	00:02.65
(3,5,2)	55/15	3	13	4,125	00:13.47
(1,2,7)	73/17	1	1	90	00:02.99
(7,4,4)	84/30	1	1	107	00:03.14
(7,4,4)	75/21	3	25	151,379	08:14.66
(4,8,3)	88/30	1	1	112	00:03.14
(4,8,3)	77/21	3	25	148,015	08:28.38
(2,5,8)	106/32	1	1	132	00:05.55
(2,5,8)	94/25	3	25	150,088	09:40.77

not a lot of freedom is given to the system to do actions, which leads to inefficient traces. When moving to the (6,2,2) configuration, a lot is gained though. Already halfway through the state space search do we encounter a minimal-time trace. This encourages us to believe that the on-the-fly searching method can help more and more with even bigger configurations.

The problem with the on-the-fly searching method, of course, is that still the amount of states that have to be explored grows rapidly when increasing the number of fluids in a configuration. At this moment, we are not able to deal with configurations bigger than (6,2,2), but once the hardware gets improved and our generator gets optimised we will be able to in the future.

When using a g-synchronised priority beam search in the CCA case study, it turns out that the search progresses much faster compared to using a detailed beam search. Furthermore, in all cases, we are able to find the optimal solutions with smaller beam widths, when compared to using a detailed beam search. It shows that the evaluation function used for the detailed beam search can be improved. We have not tried to improve the total-cost evaluation function yet. It turns out that this particular scheduling problem is well solvable by assigning priorities to actions. This is already noticable by the effectiveness of the strategy specifications. Based on these results, we decide not to perform any tests using flexible detailed beam search. At least, the findings here are

Table 10.9: g-Synchronised flexible priority beam search results for the CCA

Spec. **# Tests**	All cycles	# States	Runtime
(3,1,1)	36/11	821 (of 4,001)	00:03.70
(1,3,1)	39/11	1,133 (of 15,091)	00:04.06
(1,1,3)	45/12	1,145 (of 39,132)	00:04.03
(6,2,2)	51/15	45,402 (of 677,470,840)	02:33.65
(3,5,2)	55/15	128,373	06:44.93
(1,2,7)	73/17	122,449	04:02.94
(7,4,4)	75/21	20,666,509	872:55.71
(4,8,3)	-	-	-
(2,5,8)	-	-	-

in contrast with experiences in other settings, for instance the results found by Valente and Alves (2004), who state that in general, detailed beam searches, with their global view on the state space, perform more accurately than priority beam searches. One has to note, though, that due to these different settings, comparisons cannot be easily made.

Solutions are found quicker using beam search than using on-the-fly searching, but of course, when applied to bigger cases for which a minimal-time trace has not been found yet, this is at the expense of finding near-optimal solutions.

Using the (g-synchronised) flexible priority beam search, we find that, with $\alpha^l = 1$ and the right priority assignments, the obtained results are the same as the ones obtained from the strategy specification during the earlier testing. The flexible beam search technique, therefore, saves the user the effort of separately specifying a specification with a built-in strategy, if such a specification is only needed to place an ordering on actions. This is not only convenient, but also removes the possibility of errors or unwanted behaviour, which may appear when writing a specification with a strategy. Besides that, it makes changing a strategy during testing very straightforward. Of course, this comes at a cost; finding a solution using flexible priority beam search takes more time than finding the same solution using a specification with a built-in strategy, due to the evaluation procedure.

Compared to the other beam search variants used, we no longer have the problem of determining the beam width for each test batch when using flexible beam search.

10.5.8 Other Findings

Looking at the $(4, 8, 3)$ batch within the strategy specification produces some strange results; the state space turns out to be of infinite size. Since this is unexpected, we look at it in more detail, and find a trace of infinite size showing that it would be wise to have a cycle which only empties a cuvette, if one wants to exclude the possibility for the scheduler to create an invalid schedule. The trace in question will now be presented, where we always indicate the type of the test subjected to an operation, using a superscript i for an i-reagent test. Furthermore, ϵ is the 12-cycle in which no operation at all is executed; basically it is a delay. This is the trace:

$$\begin{aligned}[&R^3_1(0), R^3_1(1), R^3_1(2), R^1_1(3), R^1_1(4), R^1_1S^3(5), R^1_1S^3(6), R^2_1S^3R^3_2(7,0), R^2_1S^3R^1_2(8,1),\\ &R^2_1S^3R^1_2(9,2), R^2_1S^1R^3_3(10,0), S^1R^3_3(0,1), R^2_1R^3_3(3,2), R^2_1(6), S^2(1), R^2_1S^2R^2_2(4,7),\\ &S^2R^2_2(2,8), R^2_1R^2_2(5,8), S^2(8), S^2R^2_2(9,3), S^2R^2_2(0,4), S^2R^2_2(10,6), S^2R^2_2(3,5),\\ &R^2_2(9), \epsilon, \epsilon, \epsilon, \ldots]\end{aligned}$$

In this trace, all the cuvets get filled with tests in such a way that there is never a completed test at the emptying position. In the end, the rotor is filled entirely with completed tests, but nothing can be removed, because there is no cycle in which only a removal operation is done.

10.6 Conclusions

The modelling language μCRL is well-suited for modelling scheduling problems. The data support it has is very convenient when working with complex data structures, as in the case of the CCA. In this regard, no changes had to be made to the current μCRL toolset. In other regards, the μCRL toolset had to be extended with search algorithms other than breadth-first search. Although not a necessity, a useful feature in the modelling language μCRL would be a priority operator, which could be used to assign priorities to actions.

Furthermore, it often suffices to model a single process, as all the experiments in this chapter show. This applies to scheduling problems in general, since the nature of this kind of problems is to find, within all possible sequences of commands, actions, etc., a minimal-time trace leading to a successful termination.

The number of possibilities can grow very rapidly though, when increasing the size of the problem instance. Particularly in case of the CCA, we already encountered technical problems concerning the size of the state space when working with 10 products in a test batch. It is possible, however, to limit the specification in certain ways to make this state space smaller. For the CCA, we restricted new tests to be added to the first empty cuvette on the rotor (counter-clock wise) available.

Another way is to build a specification with a built-in strategy. By introducing a strategy, the number of possible execution sequences can be brought down a lot, depending on the level of non-determinism still in the specification. A specification with a built-in strategy can be used to compare a certain strategy with the general specification. Besides that, it can serve to determine an upper-bound for a bounded search through a state space of a general specification. Note that using a specification with a built-in strategy does not guarantee finding minimal-cost traces, depending on the effectiveness of the strategy chosen.

In a distributed setting we are able to deal with bigger problem instances. When performing minimal-cost search in this setting, we found that the larger the state space, the higher the percentage of states which can be avoided in the search.

We showed that g-synchronised (flexible) detailed beam search is suitable for finding near-optimal solutions for instances of the Cannibals and Missionaries problem and the Zebra Finch problem, which are two problems in the class of river crossing problems. In these particular problems, the state spaces incorporate cycles, confluence of traces, and unsuccessful termination states, thereby they are useful examples to demonstrate that the beam search variants of Chapter 8 can deal with arbitrary state spaces. Moving to a distributed setting with the beam searches has proven to be worthwhile. Of course, there is some communication overhead when searching in a distributed way, but when dealing with big problem instances and beam widths, overall there is a gain in processing time, compared to a sequential search. In the case of the CCA, we used both g-synchronised detailed and g-synchronised priority beam search; the latter turning out to be more effective in this particular case study, meaning that using priority beam search, smaller beam widths are needed to get similar solutions in shorter execution times. This can be due to the fact that the CCA scheduling problem seems to be well solvable by assigning priorities to actions, as can already be seen by the effectiveness of specifications with a built-in strategy. Beam search allows one to make a trade-off between computation time and the quality of the solutions to find. Having both detailed and priority beam search to work with, even increases the possibilities for such a trade-off. If one wants a certain level of quality, however, choosing the right beam width becomes a problem.

Because of this, in Chapter 8, we proposed extensions of both detailed and priority beam search, called flexible beam search, in which the actual beam width can change while searching, in order to keep track of all actions with a sufficient priority in each level (i.e. avoiding tie-breaking). The experiments suggest that from case to case, the beam widths of flexible beam searches do not have to be increased often. The experimental results for the CCA suggest that flexible priority beam search removes the necessity to create additional specifications with built-in strategies, if they are only needed to assign priorities to actions. Flexible priority beam search combines the ease of use of beam search, meaning that no additional specifications have to be created to use it, with the flexibility of a specification with a built-in strategy, meaning that there is no limit to the number of states or transitions expanded per level. The major benefits of flexible beam searches are the relative ‘stability’ of the beam widths (i.e. when increasing the size of the test configuration, the beam width can often be left unchanged) and the avoidance of tie-breaking, but this comes at a price, namely that the space and computation time requirements of these searches are not linear to the maximum search depth.

As a side note, in Section 10.5.8, we showed an example of gaining results not related to the scheduling problem in question. When generating a state space you may notice some unexpected behaviour, which could lead to more insight into the system.

Part III

Comparing Timed Behaviour

IN WHICH A DEFINITION OF TIMED BRANCHING BISIMILARITY IS EXTENDED, THE EXTENDED NOTION IS PROVED TO BE AN EQUIVALENCE, AND A ROOTED VERSION IS PROVED TO BE A CONGRUENCE

This part is an extended version of the work by Fokkink et al. (2005)

Chapter 11

Is Timed Branching Bisimilarity a Congruence Indeed?

> *Time present and time past*
> *Are both perhaps present in time future,*
> *And time future contained in time past.*
>
> (*T.S. Eliot*)

11.1 Introduction

BRANCHING BISIMILARITY (Van Glabbeek and Weijland, 1989, 1996a) is a widely used concurrency semantics for process algebras that include the silent step τ. Two processes are branching bisimilar if they can be related by some branching bisimulation relation. See the work of Van Glabbeek (1994) for a clear account on the strong points of branching bisimilarity.

Over the years, process algebras such as CCS, CSP and ACP have been extended with a notion of time. As a result, the concurrency semantics underlying these process algebras have been adapted to cope with the presence of time. Klusener (1991, 1992, 1993) was the first to extend the notion of a branching bisimulation relation to a setting with time. The main complication is that while a process can let time pass without performing an action, such idling may mean that certain behavioural options in the future are being discarded. Klusener pioneered how this aspect of timed processes can be taken into account in a branching bisimulation context. Based on his work, Van der Zwaag (2001, 2002) and Baeten and Middelburg (2002) proposed new notions of a timed branching bisimulation relation.

A key property for a semantics is that it is an equivalence. In general, for concurrency semantics in the presence of τ, reflexivity and symmetry are easy to see, but transitivity is much more difficult. In particular, the transitivity proof for branching bisimilarity of Van Glabbeek and Weijland (1989) turned out to be flawed, because the transitive closure of two branching bisimulation relations need not be a branching bisimulation relation. Basten (1996) pointed out this flaw, and proposed a new transitivity proof for branching bisimilarity, based on the notion of a *semi*-branching bisimulation relation. Such relations are preserved under transitive closure, and the notions of branching bisimilarity and semi-branching bisimilarity coincide.

In a setting with time, proving equivalence of a concurrency semantics becomes even more complicated, compared to the untimed case. Still, equivalence properties for timed semantics are often claimed, but hardly ever proved. Klusener (1993), Van der Zwaag (2001, 2002), and Baeten and Middelburg (2002) claim equivalence properties without an explicit proof, although in all cases it is stated that such proofs do exist.

Related to this, it is an interesting question whether a rooted version of timed branching bisimilarity is a congruence over a basic timed process algebra containing parallelism, successful termination and deadlock (such as Baeten and Bergstra's $\mathrm{BPA}^{ur}_{\rho\delta}$ (Baeten and Bergstra, 1991), which is basic real time process algebra with time stamped urgent actions). Similar to equivalence, congruence properties for timed branching bisimilarity are often claimed, but hardly ever proved. One such congruence proof is provided by Klusener (1993). Considering other timed settings, Reniers and Van Weerdenburg (2007) provide a congruence proof for a setting with an untimed τ-step, which makes it possible for them, unlike for us, to follow the format of the usual congruence proof for untimed branching bisimilarity. Trčka (2007) proved timed branching bisimilarity to be a congruence over a timed process algebra in a setting with discrete, relative time.

In the current chapter, first of all, we study in how far the notion of timed branching bisimilarity of Van der Zwaag constitutes an equivalence relation. We give a counter-example to show that in case of a dense time domain, his notion is not transitive. We proceed to present a stronger version of Van der Zwaag's definition (stronger in the sense that it relates fewer processes), and prove that this adapted notion does constitute an equivalence relation, even when the time domain is dense. Our proof follows the approach of Basten. Next, we show that in case of a discrete time domain, Van der Zwaag's notion of timed branching bisimilarity and our new notion coincide. So in particular, in case of a discrete time domain, Van der Zwaag's notion does constitute an equivalence relation.

In Appendix C, we show that our counter-example for transitivity also applies to the notion of timed branching bisimilarity by Baeten & Middelburg in case of a dense time domain (see Baeten and Middelburg (2002, Section 6.4.1)). So that notion does not constitute an equivalence relation as well.

Following the equivalence proof, we prove that a rooted version of the stronger version of timed branching bisimilarity is a congruence over a basic timed process algebra containing parallelism, successful termination and deadlock. In a number of ways, our proof differs from the usual congruence proof for untimed branching bisimilarity. For example, due to the presence of successful termination, there is a large number of cases. In fact, the congruence proof for the parallel composition operator is restricted to a setting without successful termination, since the number of cases in a proof considering successful termination is just too large. Furthermore, Example 7 demonstrates that the standard approach for untimed branching bisimilarity, i.e. take the smallest congruence closure and prove that this yields a branching bisimulation, falls short in a timed setting.

This chapter is organized as follows. Section 11.2 contains the preliminaries, describing the notion of a *timed labelled transition system*, i.e. a timed state space. Section 11.3 features a counter-example to show that the notion of timed branching bisimilarity by Van der Zwaag is not an equivalence relation in case of a dense time domain. A new definition of timed branching bisimulation is proposed in Section 11.4, and we prove that our notion of timed branching bisimilarity is an equivalence indeed. In Section 11.5 we prove that in case of a discrete time domain, our definition and Van der Zwaag's definition of timed branching bisimilarity coincide. Section 11.7 gives suggestions for future work. In Appendix C, we show that our counter-example for transitivity also applies to the notion of timed branching bisimilarity by Baeten and Middelburg (2002).

11.2 Timed Labelled Transition Systems

Let $\mathscr{A}$ be a non-empty set of visible actions, and τ a special action to represent internal events, with $\tau \notin \mathscr{A}$. We use $\mathscr{A}_\tau$ to denote $\mathscr{A} \cup \{\tau\}$.

The time domain $\mathbb{T}$ is a totally ordered set with a least element 0. We say that $\mathbb{T}$ is *discrete* if for each pair $u, v \in \mathbb{T}$ there are only finitely many $w \in \mathbb{T}$ such that $u < w < v$.

Definition 28 (Timed labelled transition system (Van der Zwaag, 2001)). *A* timed labelled transition system *(TLTS) (Groote et al., 2002) is a triple $(\mathscr{S}, \mathscr{T}, \mathscr{U})$, where:*

1. *$\mathscr{S}$ is a set of states, including a special state $\surd$ to represent successful termination;*
2. *$\mathscr{T} \subseteq \mathscr{S} \times \mathscr{A}_\tau \times \mathbb{T} \times \mathscr{S}$ is a set of transitions;*
3. *$\mathscr{U} \subseteq \mathscr{S} \times \mathbb{T}$ is a* delay relation*, which satisfies:*
    - *if $\mathscr{T}(s, \ell, u, r)$, then $\mathscr{U}(s, u)$;*
    - *if $u < v$ and $\mathscr{U}(s, v)$, then $\mathscr{U}(s, u)$.*

Transitions (s, ℓ, u, s') express that state s evolves into state s' by the execution of action ℓ at (absolute) time u. It is assumed that the execution of transitions does not consume any time. A transition (s, ℓ, u, s') is denoted by $s \xrightarrow{\ell}_u s'$. If $\mathscr{U}(s, u)$, then state s can let time pass until time u; these predicates are used to express time deadlocks.

11.3 Van der Zwaag's Timed Branching Bisimulation

Van Glabbeek and Weijland (1996a) introduced the notion of a *branching bisimulation* relation for untimed LTSs. Intuitively, a τ-transition $s \xrightarrow{\tau} s'$ is invisible if it does not lose possible behaviour (i.e., if s and s' can be related by a branching bisimulation relation). See the work of Van Glabbeek (1994) for a lucid exposition on the motivations behind the definition of a branching bisimulation relation.

The reflexive transitive closure of $\xrightarrow{\tau}$ is denoted by $\Rightarrow$.

Definition 29 (Branching bisimulation (Van Glabbeek and Weijland, 1996a)). *Assume an untimed LTS. A symmetric binary relation $B \subseteq \mathscr{S} \times \mathscr{S}$ is a* branching bisimulation *if $s \; B \; t$ implies:*

1. *if $s \xrightarrow{\ell} s'$, then*

   i either $\ell = \tau$ and $s' \; B \; t$,

   ii or $t \Rightarrow \hat{t} \xrightarrow{\ell} t'$ with $s \; B \; \hat{t}$ and $s' \; B \; t'$;

2. *if $s \downarrow$, then $t \Rightarrow t' \downarrow$ with $s \; B \; t'$.*

Two states s and t are branching bisimilar, *denoted by $s \leftrightarroweq_b t$, if there is a branching bisimulation B with $s \; B \; t$.*

Van der Zwaag (2001) defined a timed version of branching bisimulation, which takes into account time stamps of transitions and ultimate delays $\mathscr{U}(s,u)$.

For $u \in \mathbb{T}$, the reflexive transitive closure of $\xrightarrow{\tau}_u$ is denoted by $\Rightarrow_u$.

Definition 30 (Timed branching bisimulation (Van der Zwaag, 2001)). *Assume a TLTS $(\mathscr{S}, \mathscr{T}, \mathscr{U})$. A collection B of symmetric binary relations $B_u \subseteq \mathscr{S} \times \mathscr{S}$ for $u \in \mathbb{T}$ is a* timed branching bisimulation *if $s \; B_u \; t$ implies:*

1. *if $s \xrightarrow{\ell}_u s'$, then*

   i either $\ell = \tau$ and $s' \; B_u \; t$,

   ii or $t \Rightarrow_u \hat{t} \xrightarrow{\ell}_u t'$ with $s \; B_u \; \hat{t}$ and $s' \; B_u \; t'$;

2. *if $s \downarrow$, then $t \Rightarrow_u t' \downarrow$ with $s \; B_u \; t'$;*

3. *if $u \leq v$ and $\mathscr{U}(s,v)$, then for some $n \geq 0$ there are $t_0, \ldots, t_n \in \mathscr{S}$ with $t = t_0$ and $\mathscr{U}(t_n, v)$, and $u_0 < \cdots < u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $t_i \Rightarrow_{u_i} t_{i+1}$, $s \; B_{u_i} \; t_{i+1}$ and $s \; B_{u_{i+1}} \; t_{i+1}$.*

Two states s and t are timed branching bisimilar at u, *denoted by $s \leftrightarroweq_{tb}^{Z,u} t$, if there is a timed branching bisimulation B with $s \; B_u \; t$. States s and t are* timed branching bisimilar, *denoted by $s \leftrightarroweq_{tb}^{Z} t$,*[1] *if they are timed branching bisimilar at all $u \in \mathbb{T}$.*

[1] The superscript Z refers to van der Zwaag, to distinguish it from the adaptation of his definition of timed branching bisimulation that we will define later.

Transitions can be executed at the same time consecutively. By the first clause in Definition 30, the behaviour of a state at some point in time is treated like untimed behaviour. The second clause deals with successful termination.[2] By the last clause, time passing in a state s is matched by a related state t with a "τ-path" where all intermediate states are related to s at times when a τ-transition is performed.[3]

In the following examples, $\mathbb{Z}_{\geq 0} \subseteq \mathbb{T}$, and, for any states $s_0, s_1 \in \mathscr{S}$, if $s_0 \xrightarrow{\ell}_u s_1$, then $\mathscr{U}(s_0, u)$ and for all $v > u$, $\neg\mathscr{U}(s_0, v)$.

Example 1. *Consider the following two TLTSs: $s_0 \xrightarrow{a}_2 s_1 \xrightarrow{b}_1 s_2$, $t_0 \xrightarrow{a}_2 t_1$, $\mathscr{U}(s_1, 1)$, and $\mathscr{U}(t_1, 1)$. We have $s_0 \mathrel{\underline{\leftrightarrow}}^Z_{tb} t_0$, since $s_0\ B_w\ t_0$ for $w \geq 0$, $s_1\ B_w\ t_1$ for $w > 1$, and $s_2\ B_w\ t_1$ for $w \geq 0$ is a timed branching bisimulation.*

Example 2. *Consider the following two TLTSs: $s_0 \xrightarrow{a}_1 s_1 \xrightarrow{\tau}_2 s_2 \xrightarrow{b}_3 s_3$, $t_0 \xrightarrow{a}_1 t_1 \xrightarrow{b}_3 t_2$, $\mathscr{U}(s_3, 4)$, and $\mathscr{U}(t_2, 4)$. We have $s_0 \mathrel{\underline{\leftrightarrow}}^Z_{tb} t_0$, since $s_0\ B_w\ t_0$ for $w \geq 0$, $s_1\ B_w\ t_1$ for $w \leq 2$, $s_2\ B_w\ t_1$ for $w \geq 0$, and $s_3\ B_w\ t_2$ for $w \geq 0$ is a timed branching bisimulation.*

Example 3. *Consider the following two TLTSs: $s_0 \xrightarrow{a}_u s_1 \xrightarrow{\tau}_v s_2 \downarrow$ and $t_0 \xrightarrow{a}_u t_1 \downarrow$. If $u = v$, we have $s_0 \mathrel{\underline{\leftrightarrow}}^Z_{tb} t_0$, since $s_0\ B_w\ t_0$ for $w \geq 0$, $s_1\ B_u\ t_1$, and $s_2\ B_w\ t_1$ for $w \geq 0$ is a timed branching bisimulation. If $u \neq v$, we have $s_0 \not\mathrel{\underline{\leftrightarrow}}^Z_{tb} t_0$, because s_1 and t_1 are not timed branching bisimilar at time u; namely, t_1 has a successful termination, and s_1 cannot simulate this at time u, as it cannot do a τ-transition at time u.*

Example 4. *Consider the following two TLTSs: $s_0 \xrightarrow{\tau}_u s_1 \xrightarrow{a}_v s_2 \downarrow$ and $t_0 \xrightarrow{a}_v t_1 \downarrow$. If $u = v$, we have $s_0 \mathrel{\underline{\leftrightarrow}}^Z_{tb} t_0$, since $s_0\ B_w\ t_0$ for $w \geq 0$, $s_1\ B_w\ t_0$ for $w \geq 0$, and $s_2\ B_w\ t_1$ for $w \geq 0$ is a timed branching bisimulation. If $u \neq v$, we have $s_0 \not\mathrel{\underline{\leftrightarrow}}^Z_{tb} t_0$, because s_0 and t_0 are not timed branching bisimilar at time $\frac{u+v}{2}$.*[4]

Van der Zwaag (2001, 2002) wrote about his definition: "It is straightforward to verify that branching bisimilarity is an equivalence relation." However, we found that in general this is not the case. A counter-example is presented below. Note that it uses a non-discrete time domain.

Example 5. *Let p, q, and r defined as in Figures 11.1, 11.2 and 11.3, with $\mathbb{T} = \mathbb{Q}_{\geq 0}$. We depict $s \xrightarrow{a}_u s'$ as $s \xrightarrow{a(u)} s'$.*

[2] Van der Zwaag does not take into account successful termination, so the second clause is missing in his definition.

[3] In the definition of Van der Zwaag, instead of $u \leq v$ and $n \geq 0$, $u < v$ and $n > 0$ are written, respectively. The change is needed in order to deal correctly with the deadlock process $\delta(u)$ and the parallel composition operator $\|$ later on, when we come to the congruence proof in Section 11.6. According to the old definition, $\delta(1) \mathrel{\underline{\leftrightarrow}}^{Z,2}_{tb} \delta(2)$, but then, since $a(2) \parallel \delta(1) \not\mathrel{\underline{\leftrightarrow}}^{Z,2}_{tb} a(2) \parallel \delta(2)$, the congruence proof would be broken. Instead, it is desirable that $\delta(1) \not\mathrel{\underline{\leftrightarrow}}^{Z,2}_{tb} \delta(2)$. Van der Zwaag did not consider deadlock explicitly; in the absence of deadlock, the two definitions (with '$u < v$' and '$u \leq v$') coincide.

[4] $s_0 \mathrel{\underline{\leftrightarrow}}^Z_{tb} t_0$ would hold for $u \leq v$ if in Definition 30 we would require that they are timed branching bisimilar at 0 (instead of at all $u \in \mathbb{T}$).

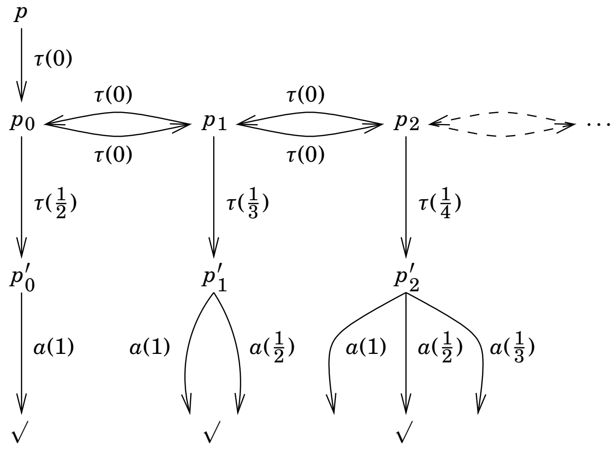

Figure 11.1: A timed process p

$p \underline{\leftrightarrow}^Z_{tb} q$, since $p\ B_w\ q$ for $w \geq 0$, $p_i\ B_w\ q_i$ for $w \leq \frac{1}{i+2}$, and $p'_i\ B_w\ q_i$ for $w > 0$ (for $i \geq 0$) is a timed branching bisimulation.

Moreover, $q \underline{\leftrightarrow}^Z_{tb} r$, since $q\ B_w\ r$ for $w \geq 0$, $q_i\ B_w\ r_i$ for $w \geq 0$, $q_i\ B_0\ r_j$, and $q_i\ B_w\ r_\infty$ for $w = 0 \vee w > \frac{1}{i+2}$ (for $i, j \geq 0$) is a timed branching bisimulation. (Note that q_i and r_∞ are not *timed branching bisimilar in the time interval $\langle 0, \frac{1}{i+2}]$.)*

However, $p \not\underline{\leftrightarrow}^Z_{tb} r$, due to the fact that none of the p_i can simulate r_∞. Namely, r_∞ can idle until time 1; p_i can only simulate this by executing a τ at time $\frac{1}{i+2}$, but the resulting process $\sum_{n=1}^{i+1} a(\frac{1}{n})$ is not timed branching bisimilar to r_∞ at time $\frac{1}{i+2}$, since

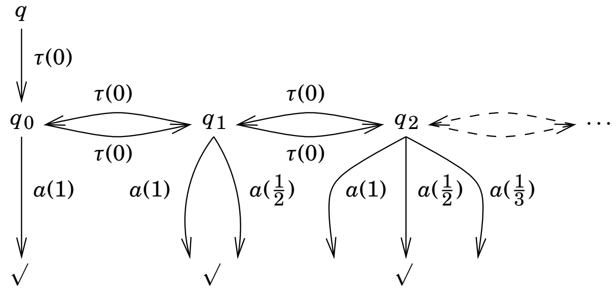

Figure 11.2: A timed process q

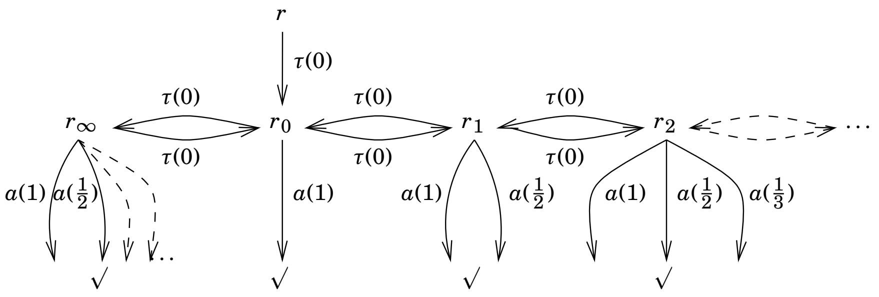

Figure 11.3: A timed process r

only the latter can execute action a at time $\frac{1}{i+2}$.

11.4 A Strengthened Timed Branching Bisimulation

In this section, we propose a way to fix the definition of Van der Zwaag (see Definition 30). Our adaptation requires the *stuttering property* (Van Glabbeek and Weijland, 1996a) (see Definition 33) at all time intervals. That is, in the last clause of Definition 30, we require that $s\ B_w\ t_{i+1}$ for $u_i \leq w \leq u_{i+1}$. Hence, we achieve a stronger version of Van der Zwaag's definition. We prove that this new notion of timed branching bisimilarity is an equivalence relation.

11.4.1 Timed Branching Bisimulation

Definition 31 (Timed branching bisimulation). *Assume a TLTS $(\mathscr{S}, \mathscr{T}, \mathscr{U})$. A collection B of binary relations $B_u \subseteq \mathscr{S} \times \mathscr{S}$ for $u \in \mathbb{T}$ is a* timed branching bisimulation *if $s\ B_u\ t$ implies:*

1. *if $s \xrightarrow{\ell}_u s'$, then*

   i either $\ell = \tau$ and $s'\ B_u\ t$,

   ii or $t \Rightarrow_u \hat{t} \xrightarrow{\ell}_u t'$ with $s\ B_u\ \hat{t}$ and $s'\ B_u\ t'$;

2. *if $t \xrightarrow{\ell}_u t'$, then*

   i either $\ell = \tau$ and $s\ B_u\ t'$,

   ii or $s \Rightarrow_u \hat{s} \xrightarrow{\ell}_u s'$ with $\hat{s}\ B_u\ t$ and $s'\ B_u\ t'$;

3. *if $s \downarrow$, then $t \Rightarrow_u t' \downarrow$ with $s\ B_u\ t'$;*

4. *if $t \downarrow$, then $s \Rightarrow_u s' \downarrow$ with $s' \ B_u \ t$;*

5. *if $u \leq v$ and $\mathcal{U}(s, v)$, then for some $n \geq 0$ there are $t_0, \ldots, t_n \in \mathcal{S}$ with $t = t_0$ and $\mathcal{U}(t_n, v)$, and $u_0 < \cdots < u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $t_i \Rightarrow_{u_i} t_{i+1}$ and $s \ B_w \ t_{i+1}$ for $u_i \leq w \leq u_{i+1}$;*

6. *if $u \leq v$ and $\mathcal{U}(t, v)$, then for some $n \geq 0$ there are $s_0, \ldots, s_n \in \mathcal{S}$ with $s = s_0$ and $\mathcal{U}(s_n, v)$, and $u_0 < \cdots < u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $s_i \Rightarrow_{u_i} s_{i+1}$ and $s_{i+1} \ B_w \ t$ for $u_i \leq w \leq u_{i+1}$.*

Two states s and t are timed branching bisimilar at *u, denoted by $s \leftrightarrows_{tb}^{u} t$, if there is a timed branching bisimulation B with $s \ B_u \ t$. States s and t are* timed branching bisimilar*, denoted by $s \leftrightarrows_{tb} t$, if they are timed branching bisimilar at all $u \in \mathbb{T}$.*

It is not hard to see that the union of timed branching bisimulations is again a timed branching bisimulation.

Note that states q and r from Example 5 are not timed branching bisimilar according to Definition 31. Namely, none of the q_i can simulate r_∞ in the time interval $\langle 0, \frac{1}{i+2}]$, so that the stuttering property is violated.

Starting from this point, we focus on timed branching bisimulation as defined in Definition 31. We did not define this new notion of timed branching bisimulation as a symmetric relation (like in Definition 30), in view of the equivalence proof that we are going to present. Namely, in general the relation composition of two symmetric relations is not symmetric. Clearly any symmetric timed branching bisimulation is a timed branching bisimulation. Furthermore, it follows from Definition 31 that the inverse of a timed branching bisimulation is again a timed branching bisimulation, so the union of a timed branching bisimulation and its inverse is a symmetric timed branching bisimulation. Hence, Definition 31 and the definition of timed branching bisimulation as a symmetric relation give rise to the same notion.

Example 6. *Consider the following two TLTSs: $s_0 \xrightarrow{a}_1 s_1$ and $t_0 \xrightarrow{a}_1 t_1$, with $\mathcal{U}(s_1, 0)$ and $\mathcal{U}(t_1, 1)$ We have $s_0 \not\leftrightarrows_{tb} t_0$, because s_1 and t_1 are not timed branching bisimilar at time 1; namely, t_1 can delay until time 1, and s_1 can neither delay until time 1, nor simulate this by doing τ-transitions at time 1 to a state which can delay until time 1. (Note that s_0 and t_0 are timed branching bisimilar according to the original definition of Van der Zwaag; see footnote 3).*

11.4.2 Timed Semi-branching Bisimulation

Basten (1996) showed that the relation composition of two (untimed) branching bisimulations is not necessarily again a branching bisimulation. Figure 11.4 illustrates an example, showing that the relation composition of two timed branching bisimulations is not always a timed branching bisimulation. It is a slightly simplified version of an example from Basten (1996), here applied at time 0. Clearly, B and D are timed branching bisimulations. However, $B \circ D$ is not, and the problem arises at the transition $r_0 \xrightarrow{\tau}_0 r_1$. According to case 1 of Definition 30, since $r_0\,(B \circ D)\,t_0$, either $r_1\,(B \circ D)\,t_0$, or $r_0\,(B \circ D)\,t_1$ and $r_1\,(B \circ D)\,t_2$, must hold. But neither of these cases hold, so $B \circ D$ is not a timed branching bisimulation.

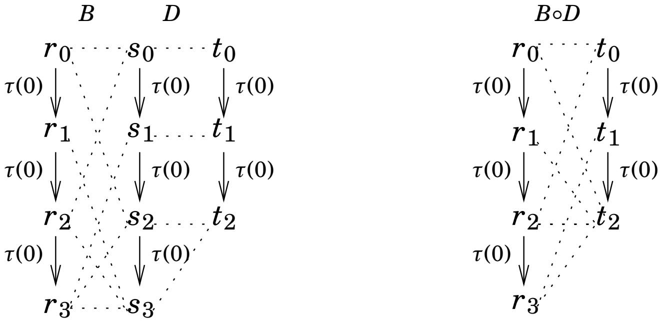

Figure 11.4: Composition does not preserve timed branching bisimulation

Semi-branching bisimulation (Van Glabbeek and Weijland, 1996a) relaxes case 1i of Definition 29: if $s \xrightarrow{\tau} s'$, then it is allowed that $t \Rightarrow t'$ with $s\ B\ t'$ and $s'\ B\ t'$. Basten proved that the relation composition of two semi-branching bisimulations is again a semi-branching bisimulation. It is easy to see that semi-branching bisimilarity is reflexive and symmetric. Hence, semi-branching bisimilarity is an equivalence relation. Then he proved that semi-branching bisimilarity and branching bisimilarity coincide, that means two states in an (untimed) LTS are related by a branching bisimulation relation if and only if they are related by a semi-branching bisimulation relation. We mimic the approach of Basten (1996) to prove that timed branching bisimilarity is an equivalence relation.

Definition 32 (Timed semi-branching bisimulation). *Assume a TLTS $(\mathscr{S}, \mathscr{T}, \mathscr{U})$. A collection B of binary relations $B_u \subseteq \mathscr{S} \times \mathbb{T} \times \mathscr{S}$ for $u \in \mathbb{T}$ is a* timed semi-branching bisimulation *if $s\ B_u\ t$ implies:*

1. *if $s \xrightarrow{\ell}_u s'$, then*

   i *either $\ell = \tau$ and $t \Rightarrow_u t'$ with $s\ B_u\ t'$ and $s'\ B_u\ t'$,*

   ii *or $t \Rightarrow_u \hat{t} \xrightarrow{\ell}_u t'$ with $s\ B_u\ \hat{t}$ and $s'\ B_u\ t'$;*

2. *if $t \xrightarrow{\ell}_u t'$, then*

   i *either $\ell = \tau$ and $s \Rightarrow_u s'$ with $s'\ B_u\ t$ and $s'\ B_u\ t'$,* *ii or* $s \Rightarrow_u \hat{s} \xrightarrow{\ell}_u s'$ *with* $\hat{s}\ B_u\ t$ *and* $s'\ B_u\ t'$*;*

3. *if* $s\downarrow$*, then* $t \Rightarrow_u t' \downarrow$ *with* $s\ B_u\ t'$*;*

4. *if* $t\downarrow$*, then* $s \Rightarrow_u s' \downarrow$ *with* $s'\ B_u\ t$*;*

5. *if* $u \leq v$ *and* $\mathcal{U}(s,v)$*, then for some* $n \geq 0$ *there are* $t_0,\ldots,t_n \in \mathcal{S}$ *with* $t = t_0$ *and* $\mathcal{U}(t_n,v)$*, and* $u_0 < \cdots < u_n \in \mathbb{T}$ *with* $u = u_0$ *and* $v = u_n$*, such that for* $i < n$*,* $t_i \Rightarrow_{u_i} t_{i+1}$ *and* $s\ B_w\ t_{i+1}$ *for* $u_i \leq w \leq u_{i+1}$*;*

6. *if* $u \leq v$ *and* $\mathcal{U}(t,v)$*, then for some* $n \geq 0$ *there are* $s_0,\ldots,s_n \in \mathcal{S}$ *with* $s = s_0$ *and* $\mathcal{U}(s_n,v)$*, and* $u_0 < \cdots < u_n \in \mathbb{T}$ *with* $u = u_0$ *and* $v = u_n$*, such that for* $i < n$*,* $s_i \Rightarrow_{u_i} s_{i+1}$ *and* $s_{i+1}\ B_w\ t$ *for* $u_i \leq w \leq u_{i+1}$*.*

Two states s *and* t *are* timed semi-branching bisimilar at u *if there is a timed semi-branching bisimulation* B *with* $s\ B_u\ t$*. States* s *and* t *are* timed semi-branching bisimilar*, denoted by* $s \leftrightarrows_{tsb} t$*, if they are timed semi-branching bisimilar at all* $u \in \mathbb{T}$*.*

It is not hard to see that the union of timed semi-branching bisimulations is again a timed semi-branching bisimulation. Furthermore, any timed branching bisimulation is a timed semi-branching bisimulation.

Definition 33 (Stuttering property (Van Glabbeek and Weijland, 1996a)). *A timed semi-branching bisimulation* B *is said to satisfy the* stuttering property *if:*

1. $s\ B_u\ t$*,* $s'\ B_u\ t$ *and* $s \xrightarrow{\tau}_u s_1 \xrightarrow{\tau}_u \cdots \xrightarrow{\tau}_u s_n \xrightarrow{\tau}_u s'$ *implies that* $s_i\ B_u\ t$ *for* $1 \leq i \leq n$*;*

2. $s\ B_u\ t$*,* $s\ B_u\ t'$ *and* $t \xrightarrow{\tau}_u t_1 \xrightarrow{\tau}_u \cdots \xrightarrow{\tau}_u t_n \xrightarrow{\tau}_u t'$ *implies that* $s\ B_u\ t_i$ *for* $1 \leq i \leq n$*.*

Lemma 7. *Any timed semi-branching bisimulation satisfying the stuttering property is a timed branching bisimulation.*

Proof. Let B be a timed semi-branching bisimulation that satisfies the stuttering property. We prove that B is a timed branching bisimulation.

Let $s\ B_u\ t$. We only consider case 1i of Definition 32, because cases 1ii, 2ii and 3-6 are the same for both timed semi-branching and branching bisimulation. Moreover, case 2i can be dealt with in a similar way as case 1i. So let $s \xrightarrow{\tau}_u s'$ and $t \Rightarrow_u t'$ with $s\ B_u\ t'$ and $s'\ B_u\ t'$. We distinguish two cases.

1. $t = t'$. Then $s'\ B_u\ t$, which agrees with case 1i of Definition 31.

2. $t \neq t'$. Then $t \Rightarrow_u t'' \xrightarrow{\tau}_u t'$. Since B satisfies the stuttering property, $s\ B_u\ t''$. This agrees with case 1ii of Definition 31.

□

11.4.3 Timed Branching Bisimilarity is an Equivalence

Our equivalence proof consists of the following main steps:

1. We first prove that the relation composition of two timed semi-branching bisimulation relations is again a semi-branching bisimulation relation (Proposition 1).
2. Then we prove that timed semi-branching bisimilarity is an equivalence relation (Theorem 1).
3. Finally, we prove that the largest timed semi-branching bisimulation satisfies the stuttering property (Proposition 2).

According to Lemma 7, any timed semi-branching bisimulation satisfying the stuttering property is a timed branching bisimulation. So by the 3rd point, two states are related by a timed branching bisimulation if and only if they are related by a timed semi-branching bisimulation.

Lemma 8. *Let B be a timed semi-branching bisimulation, and $s\ B_u\ t$.*

1. $s \Rightarrow_u s' \implies (\exists t' \in \mathscr{S} : t \Rightarrow_u t' \wedge s'\ B_u\ t')$;
2. $t \Rightarrow_u t' \implies (\exists s' \in \mathscr{S} : s \Rightarrow_u s' \wedge t'\ B_u\ s')$.

Proof. We prove the first part, by induction on the number of τ-transitions at u from s to s'.

1. *Base case:* The number of τ-transitions at u from s to s' is zero. Then $s = s'$. Take $t' = t$. Clearly $t \Rightarrow_u t'$ and $s'\ B_u\ t'$.
2. *Inductive case:* $s \Rightarrow_u s'$ consists of $n \geq 1$ τ-transitions at u. Then there exists an $s'' \in \mathscr{S}$ such that $s \Rightarrow_u s''$ in $n-1$ τ-transitions at u, and $s'' \xrightarrow{\tau}_u s'$. By the induction hypothesis, $t \Rightarrow_u t''$ with $s''\ B_u\ t''$. Since $s'' \xrightarrow{\tau}_u s'$ and B is a timed semi-branching bisimulation:
   - either $t'' \Rightarrow_u t'$ and $s''\ B_u\ t'$ and $s'\ B_u\ t'$;
   - or $t'' \Rightarrow_u \hat{t} \xrightarrow{\tau}_u t'$ with $s''\ B_u\ \hat{t}$ and $s'\ B_u\ t'$.

   In both cases $t \Rightarrow_u t'$ with $s'\ B_u\ t'$.

The proof of the second part is similar. □

Proposition 1. *The relation composition of two timed semi-branching bisimulations is again a timed semi-branching bisimulation.*

Proof. Let B and D be timed semi-branching bisimulations. We prove that the composition of B and D (or better, the compositions of B_u and D_u for $u \in \mathbb{T}$) is a timed semi-branching bisimulation. Suppose that $r\ B_u\ s\ D_u\ t$ for $r,s,t \in \mathscr{S}$. We check that the conditions of Definition 32 are satisfied with respect to the pair r,t. We distinguish four cases.

- $r \xrightarrow{\tau}_u r'$ and $s \Rightarrow_u s'$ with $r\ B_u\ s'$ and $r'\ B_u\ s'$. Since $s\ D_u\ t$ and $s \Rightarrow_u s'$, Lemma 8 yields that $t \Rightarrow_u t'$ with $s'\ D_u\ t'$. Hence, $r\ B_u\ s'\ D_u\ t'$ and $r'\ B_u\ s'\ D_u\ t'$.

- $r \xrightarrow{\ell}_u r'$ and $s \Rightarrow_u s'' \xrightarrow{\ell}_u s'$ with $r\ B_u\ s''$ and $r'\ B_u\ s'$. Since $s\ D_u\ t$ and $s \Rightarrow_u s''$, Lemma 8 yields that $t \Rightarrow_u t''$ with $s''\ D_u\ t''$. Since $s'' \xrightarrow{\ell}_u s'$ and $s''\ D_u\ t''$:
  - Either $\ell = \tau$ and $t'' \Rightarrow_u t'$ with $s''\ D_u\ t'$ and $s'\ D_u\ t'$. Then $t \Rightarrow_u t'$ with $r\ B_u\ s''\ D_u\ t'$ and $r'\ B_u\ s'\ D_u\ t'$.
  - Or $t'' \Rightarrow_u t''' \xrightarrow{\ell}_u t'$ with $s''\ D_u\ t'''$ and $s'\ D_u\ t'$. Then $t \Rightarrow_u t''' \xrightarrow{\ell}_u t'$ with $r\ B_u\ s''\ D_u\ t'''$ and $r'\ B_u\ s'\ D_u\ t'$.

- $r\downarrow$. Since $r\ B_u\ s$, $s \Rightarrow_u s' \downarrow$ with $r\ B_u\ s'$. Since $s\ D_u\ t$ and $s \Rightarrow_u s'$, Lemma 8 yields that $t \Rightarrow_u t''$ with $s'\ D_u\ t''$. Since $s'\downarrow$ and $s'\ D_u\ t''$, $t'' \Rightarrow_u t' \downarrow$ with $s'\ D_u\ t'$. Hence, $t \Rightarrow_u t' \downarrow$ with $r\ B_u\ s'\ D_u\ t'$.

- $u \le v$ and $\mathcal{U}(r,v)$. Since $r\ B_u\ s$, for some $n \ge 0$ there are $s_0,\ldots,s_n \in \mathscr{S}$ with $s = s_0$ and $\mathcal{U}(s_n,v)$, and $u_0 < \cdots < u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that $s_i \Rightarrow_{u_i} s_{i+1}$ and $r\ B_w\ s_{i+1}$ for $u_i \le w \le u_{i+1}$ and $i < n$.

  For $i \le n$ we show that for some $m_i \ge 0$ there are $t^i_0,\ldots,t^i_{m_i} \in \mathscr{S}$ with $t = t^0_0$ and $\mathcal{U}(t^n_{m_n},v)$, and $v^i_0 \le \cdots \le v^i_{m_i} \in \mathbb{T}$ with (A$_i$) $u_{i-1} = v^i_0$ (if $i > 0$) and (B$_i$) $u_i = v^i_{m_i}$, such that:

  (C$_i$) $t^i_j \Rightarrow_{v^i_j} t^i_{j+1}$ for $j < m_i$;

  (D$_i$) $t^{i-1}_{m_{i-1}} \Rightarrow_{u_{i-1}} t^i_0$ (if $i > 0$);

  (E$_i$) $s_i\ D_{u_{i-1}}\ t^i_0$ (if $i > 0$);

  (F$_i$) $s_i\ D_w\ t^i_{j+1}$ for $v^i_j \le w \le v^i_{j+1}$ and $j < m_i$.

  We apply induction with respect to i.
  - *Base case:* $i = 0$.

    Let $m_0 = 0$, $t^0_0 = t$ and $v^0_0 = u_0$. Note that B$_0$, C$_0$ and F$_0$ hold.
  - *Inductive case:* $0 < i \le n$.

    Suppose that $m_k, t^k_0,\ldots,t^k_{m_k}, v^k_0,\ldots,v^k_{m_k}$ have been defined for $0 \le k < i$. Moreover, suppose that B$_k$, C$_k$ and F$_k$ hold for $0 \le k < i$, and that A$_k$, D$_k$ and E$_k$ hold for $0 < k < i$.

F$_{i-1}$ for $j = m_{i-1} - 1$ together with B$_{i-1}$ yields $s_{i-1}\ D_{u_{i-1}}\ t^{i-1}_{m_{i-1}}$. Since $s_{i-1} \Rightarrow_{u_{i-1}} s_i$, Lemma 8 implies that $t^{i-1}_{m_{i-1}} \Rightarrow_{u_{i-1}} t'$ with $s_i\ D_{u_{i-1}}\ t'$. We define $t^i_0 = t'$ [then D$_i$ and E$_i$ hold] and $v^i_0 = u_{i-1}$ [then A$_i$ holds]. $s_i \Rightarrow_{u_i} \cdots \Rightarrow_{u_{n-1}} s_n$ with $\mathscr{U}(s_n, v)$ implies that $\mathscr{U}(s_i, u_i)$. Since $s_i\ D_{u_{i-1}}\ t^i_0$, according to case 5 of Definition 32, for some $m_i > 0$ there are $t^i_1, \ldots, t^i_{m_i} \in \mathscr{S}$ with $\mathscr{U}(t^i_{m_i}, u_i)$, and $v^i_1 < \cdots < v^i_{m_i} \in \mathbb{T}$ with $v^i_0 < v^i_1$ and $u_i = v^i_{m_i}$ [then B$_i$ holds], such that for $j < m_i$, $t^i_j \Rightarrow_{v^i_j} t^i_{j+1}$ [then C$_i$ holds] and $s_i\ D_w\ t^i_{j+1}$ for $v^i_j \le w \le v^i_{j+1}$ [then F$_i$ holds].

Concluding, for $i < n$, $r\ B_{u_i}\ s_{i+1}\ D_{u_i}\ t^{i+1}_0$ and $r\ B_w\ s_{i+1}\ D_w\ t^{i+1}_{j+1}$ for $v^{i+1}_j \le w \le v^{i+1}_{j+1}$ and $j < m_i$. Since $v^i_j \le v^i_{j+1}$, $v^i_{m_i} = u_i = v^{i+1}_0$, $t = t^0_0$, $u = u_0 = v^0_0$, $t^i_j \Rightarrow_{v^i_j} t^i_{j+1}$, $t^i_{m_i} \Rightarrow_{u_i} t^{i+1}_0$, and $\mathscr{U}(t^n_{m_n}, v)$, we are done.

So cases 1,3,5 of Definition 32 are satisfied. Similarly it can be checked that cases 2,4,6 are satisfied. So the composition of B and D is again a timed semi-branching bisimulation. □

Theorem 1. *Timed semi-branching bisimilarity, $\leftrightarroweq_{tsb}$, is an equivalence relation.*

Proof. Reflexivity: Obviously, the identity relation on $\mathscr{S}$ is a timed semi-branching bisimulation.
Symmetry: Let B a timed semi-branching bisimulation. Obviously, B^{-1} is also a timed semi-branching bisimulation.
Transitivity: This follows from Proposition 1. □

Proposition 2. *The largest timed semi-branching bisimulation satisfies the stuttering property.*

Proof. Let B be the largest timed semi-branching bisimulation on $\mathscr{S}$. Let $s \xrightarrow{\tau}_u s_1 \xrightarrow{\tau}_u \cdots \xrightarrow{\tau}_u s_n \xrightarrow{\tau}_u s'$ with $s\ B_u\ t$ and $s'\ B_u\ t$. We prove that $B' = B \cup \{(s_i, t) \mid 1 \le i \le n\}$ is a timed semi-branching bisimulation.

We check that all cases of Definition 32 are satisfied for the relations $s_i B'_u t$, for $1 \le i \le n$. First we check that the transitions of s_i are matched by t. Since $s \Rightarrow_u s_i$ and $s\ B_u\ t$, by Lemma 8 $t \Rightarrow_u t'$ with $s_i\ B_u\ t'$.

- If $s_i \xrightarrow{\ell}_u s''$, then it follows from $s_i\ B_u\ t'$ that:
  - Either $\ell = \tau$ and $t' \Rightarrow_u t''$ with $s_i\ B_u\ t''$ and $s''\ B_u\ t''$. Since $t \Rightarrow_u t' \Rightarrow_u t''$, this agrees with case 1i of Definition 32.
  - Or $t' \Rightarrow_u t''' \xrightarrow{\ell}_u t''$ with $s_i\ B_u\ t'''$ and $s''\ B_u\ t''$. Since $t \Rightarrow_u t' \Rightarrow_u t'''$, this agrees with case 1ii of Definition 32.

- If $s_i \downarrow$, then it follows from $s_i \; B_u \; t'$ that $t' \Rightarrow_u t'' \downarrow$ with $s_i \; B_u \; t''$. Since $t \Rightarrow_u t' \Rightarrow_u t''$, this agrees with case 3 of Definition 32.

- If $u \leq v$ and $\mathcal{U}(s_i, v)$, then it follows from $s_i \; B_u \; t'$ that for some $n \geq 0$ there are $t_0, \ldots, t_n \in \mathcal{S}$ with $t' = t_0$ and $\mathcal{U}(t_n, v)$, and $u_0 < \cdots < u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $t_i \Rightarrow_{u_i} t_{i+1}$ and $s_i \; B_w \; t_i$ for $u_i \leq w \leq u_{i+1}$. Since $t \Rightarrow_u t' \Rightarrow_u t_1$, this agrees with case 5 of Definition 32.

Next we check that the transitions of t are matched by s_i.

- If $t \xrightarrow{\ell}_u t'$, then it follows from $s' \; B_u \; t$ that:

  - Either $\ell = \tau$ and $s' \Rightarrow_u s''$ with $s'' \; B_u \; t$ and $s'' \; B_u \; t'$. Since $s_i \Rightarrow_u s' \Rightarrow_u s''$, this agrees with case 2i of Definition 32.

  - Or $s' \Rightarrow_u s''' \xrightarrow{\ell}_u s''$ with $s''' \; B_u \; t$ and $s'' \; B_u \; t'$. Since $s_i \Rightarrow_u s' \Rightarrow_u s'''$, this agrees with case 2ii of Definition 32.

- If $t \downarrow$, then it follows from $s' \; B_u \; t$ that $s' \Rightarrow_u s'' \downarrow$ with $s'' \; B_u \; t$. Since $s_i \Rightarrow_u s' \Rightarrow_u s''$, this agrees with case 4 of Definition 32.

- If $u \leq v$ and $\mathcal{U}(t, v)$, then it follows from $s' \; B_u \; t$ that for some $n \geq 0$ there are $s'_0, \ldots, s'_n \in \mathcal{S}$ with $s' = s'_0$ and $\mathcal{U}(s_n, v)$, and $u_0 < \cdots < u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $s'_i \Rightarrow_{u_i} s'_{i+1}$ and $s'_{i+1} \; B_w \; t$ for $u_i \leq w \leq u_{i+1}$. Since $s_i \Rightarrow_u s' \Rightarrow_u s'_1$, this agrees with case 6 of Definition 32.

Hence B' is a timed semi-branching bisimulation. Since B is the largest, and $B \subseteq B'$, we find that $B = B'$. So B satisfies the first requirement of Definition 33.

Since B is the largest timed semi-branching bisimulation and $\leftrightarrows_{tsb}$ is an equivalence, B is symmetric. Then B also satisfies the second requirement of Definition 33. Hence B satisfies the stuttering property. □

As a consequence, the largest timed semi-branching bisimulation is a timed branching bisimulation (by Lemma 7 and Proposition 2). Since any timed branching bisimulation is a timed semi-branching bisimulation, we have the following two corollaries.

Corollary 1. *Two states are related by a timed branching bisimulation if and only if they are related by a timed semi-branching bisimulation.*

Corollary 2. *Timed branching bisimilarity, $\leftrightarrows_{tb}$, is an equivalence relation.*

We note that for each $u \in \mathbb{T}$, timed branching bisimilarity at time u is also an equivalence relation.

11.5 Discrete Time Domains

Theorem 2. *In case of a discrete time domain, $\underline{\leftrightarrow}_{tb}^{Z}$ and $\underline{\leftrightarrow}_{tb}$ coincide.*

Proof. Clearly $\underline{\leftrightarrow}_{tb} \subseteq \underline{\leftrightarrow}_{tb}^{Z}$. We prove that $\underline{\leftrightarrow}_{tb}^{Z} \subseteq \underline{\leftrightarrow}_{tb}$. Suppose B is a timed branching bisimulation relation according to Definition 30. We show that B is a timed branching bisimulation relation according to Definition 31. B satisfies cases 1-4 of Definition 31, since they coincide with cases 1-2 of Definition 30. We prove that case 5 of Definition 31 is satisfied.

Let $s\ B_u\ t$ and $\mathcal{U}(s,v)$ with $u \le v$. Let $u_0 < \cdots < u_n \in \mathbb{T}$ with $u_0 = u$ and $u_n = v$, where $u_1, \ldots, u_{n-1}$ are all the elements from $\mathbb{T}$ that are between u and v. (Here we use that $\mathbb{T}$ is discrete.) We prove by induction on n that there are $t_0, \ldots, t_n \in \mathscr{S}$ with $t = t_0$ and $\mathcal{U}(t_n, v)$, such that for $i < n$, $t_i \Rightarrow_{u_i} t_{i+1}$ and $s\ B_w\ t_{i+1}$ for $u_i \le w \le u_{i+1}$.

- *Base case:* $n = 0$. Then $u = v$. By case 3 of Definition 30, $\mathcal{U}(t,u)$.
- *Inductive case:* $n > 0$. Since $\mathcal{U}(s,v)$, clearly also $\mathcal{U}(s,u_1)$. By case 3 of Definition 30 there is a $t_1 \in \mathscr{S}$ such that $t \Rightarrow_u t_1$, $s\ B_u\ t_1$ and $s\ B_{u_1}\ t_1$. Hence, $s\ B_w\ t_1$ for $u \le w \le u_1$. By induction, $s\ B_{u_1}\ t_1$ together with $\mathcal{U}(s,v)$ implies that there are $t_2, \ldots, t_n \in \mathscr{S}$ with $\mathcal{U}(t_n, v)$, such that for $1 \le i < n$, $t_i \Rightarrow_{u_i} t_{i+1}$, $s\ B_{u_i}\ t_{i+1}$ and $s\ B_{u_{i+1}}\ t_{i+1}$. Hence, $s\ B_w\ t_{i+1}$ for $u_i \le w \le u_{i+1}$.

We conclude that case 5 of Definition 31 holds. Similarly it can be proved that B satisfies case 6 of Definition 31. Hence B is a timed branching bisimulation relation according to Definition 31. So $\underline{\leftrightarrow}_{tb}^{Z} \subseteq \underline{\leftrightarrow}_{tb}$. □

11.6 Rooted Timed Branching Bisimilarity as a Congruence

11.6.1 Rooted Timed Branching Bisimilarity

In this section, we prove that a rooted version of the timed branching bisimulation as defined in Definition 31 is a congruence over a given basic process algebra with sequential, alternative, and parallel composition. Like (untimed) branching bisimilarity, timed branching bisimilarity is not a congruence over most process algebras from the literature. A rootedness condition has been introduced for branching bisimilarity to remedy this imperfection (Bergstra and Klop, 1985; Milner, 1989). First, we provide a related definition of rooted timed branching bisimulation in Definition 34. Following, we introduce the transition rules of a basic process algebra, encompassing atomic actions, including τ and δ, and the alternative, sequential, and parallel composition process operators. Next, we define what a congruence is. After that, the congruence proof is presented.

Definition 34 (Rooted timed branching bisimulation). *Assume a TLTS* $(\mathcal{S}, \mathcal{T}, \mathcal{U})$. *A binary relation* $B \subseteq \mathcal{S} \times \mathcal{S}$ *is a* rooted timed branching bisimulation *if* $s\ B\ t$ *implies:*

1. *if* $s \xrightarrow{\ell}_u s'$, *then* $t \xrightarrow{\ell}_u t'$ *with* $s' \leftrightarroweq^u_{tb} t'$;
2. *if* $t \xrightarrow{\ell}_u t'$, *then* $s \xrightarrow{\ell}_u s'$ *with* $s' \leftrightarroweq^u_{tb} t'$;
3. $s \downarrow$ *iff* $t \downarrow$;
4. $\mathcal{U}(s,u)$ *iff* $\mathcal{U}(t,u)$.

Two states s and t are rooted timed branching bisimilar, *denoted by* $s \leftrightarroweq_{rtb} t$, *if there is a rooted timed branching bisimulation B with* $s\ B\ t$.

Note that $\leftrightarroweq_{rtb} \subseteq \leftrightarroweq_{tb}$. A rooted timed branching bisimulation relation is a timed branching bisimulation relation, where in cases 1 to 4 of Definition 31 '$\Rightarrow_u$' constitutes zero τ-steps, and in cases 5 and 6 $n = 0$.

11.6.2 A Basic Process Algebra

In the following, x, y are variables, p, q, r are process terms, and s, t are process terms or $\surd$, with $\surd$ a special state representing successful termination.

Here, we present a basic process algebra, which we will use in subsequent sections in our congruence proof. It is based on the process algebra BPA$_{\rho\delta U}$ (Baeten and Bergstra, 1991). We consider the following transition rules for the process algebra used, where the synchronisation of two actions a and b resulting in an action c, denoted by $a \mid b = c$, is defined by means of a function $\mathbb{C} : \mathcal{A} \times \mathcal{A} \to \mathcal{A} \cup \{\tau, \delta\}$. Whenever two actions a and b should never synchronise, we define that $a \mid b = \delta$.

$$\text{Termination}: \frac{}{\surd \downarrow} \qquad \text{Atomic}: \frac{}{a(u) \xrightarrow{a}_u \surd}$$

$$\text{Alt1}: \frac{x \xrightarrow{a}_u x'}{x + y \xrightarrow{a}_u x'} \qquad \text{Alt2}: \frac{x \xrightarrow{a}_u \surd}{x + y \xrightarrow{a}_u \surd}$$

$$\text{Alt3}: \frac{y \xrightarrow{a}_u y'}{x + y \xrightarrow{a}_u y'} \qquad \text{Alt4}: \frac{y \xrightarrow{a}_u \surd}{x + y \xrightarrow{a}_u \surd}$$

$$\text{Seq1}: \frac{x \xrightarrow{a}_u x'}{x \cdot y \xrightarrow{a}_u x' \cdot y} \qquad \text{Seq2}: \frac{x \xrightarrow{a}_u \surd}{x \cdot y \xrightarrow{a}_u y}$$

$$\text{Par1}: \frac{x \xrightarrow{a}_u x' \quad \mathcal{U}(y,u)}{x \parallel y \xrightarrow{a}_u x' \parallel y} \qquad \text{Par2}: \frac{y \xrightarrow{a}_u y' \quad \mathcal{U}(x,u)}{x \parallel y \xrightarrow{a}_u x \parallel y'}$$

$$\text{Par3}: \frac{x \xrightarrow{a}_u \surd \quad \mathcal{U}(y,u)}{x \parallel y \xrightarrow{a}_u y} \qquad \text{Par4}: \frac{y \xrightarrow{a}_u \surd \quad \mathcal{U}(x,u)}{x \parallel y \xrightarrow{a}_u x}$$

$$\text{Par5}: \frac{x \xrightarrow{a}_u x' \quad y \xrightarrow{b}_u y' \quad a \mid b = c \quad c \neq \delta}{x \parallel y \xrightarrow{c}_u x' \parallel y'}$$

$$\text{Par6}: \frac{x \xrightarrow{a}_u \surd \quad y \xrightarrow{b}_u y' \quad a \mid b = c \quad c \neq \delta}{x \parallel y \xrightarrow{c}_u y'}$$

$$\text{Par7}: \frac{x \xrightarrow{a}_u x' \quad y \xrightarrow{b}_u \surd \quad a \mid b = c \quad c \neq \delta}{x \parallel y \xrightarrow{c}_u x'}$$

$$\text{Par8}: \frac{x \xrightarrow{a}_u \surd \quad y \xrightarrow{b}_u \surd \quad a \mid b = c \quad c \neq \delta}{x \parallel y \xrightarrow{c}_u \surd}$$

$$\begin{array}{l}
\mathcal{U}(\surd, 0) \\
\mathcal{U}(a(u), v) \textit{ if } v \leq u \\
\mathcal{U}(\delta(u), v) \textit{ if } v \leq u \\
\mathcal{U}(x \cdot y, v) \Leftrightarrow \mathcal{U}(x, v) \\
\mathcal{U}(x + y, v) \Leftrightarrow \mathcal{U}(x, v) \vee \mathcal{U}(y, v) \\
\mathcal{U}(x \parallel y, v) \Leftrightarrow \mathcal{U}(x, v) \wedge \mathcal{U}(y, v)
\end{array}$$

Definition 35 (Congruence). *An equivalence relation $R \subseteq \mathcal{S} \times \mathcal{S}$ over a process algebra with sequential composition (‘·’), alternative composition (‘+’) , and parallel composition (‘‖’) is called a* congruence *if $s_1\ R\ t_1$ and $s_2\ R\ t_2$ implies:*

1. $s_1 \cdot s_2\ R\ t_1 \cdot t_2$;
2. $s_1 + s_2\ R\ t_1 + t_2$;
3. $s_1 \parallel s_2\ R\ t_1 \parallel t_2$.

11.6.3 Congruence Proof for Sequential Composition

First, we prove that rooted timed branching bisimilarity is a congruence for the sequential composition operator.

We give an example to show that if $p_0 \leftrightarrow^u_{tb} q_0$ and $p_1 \leftrightarrow^u_{tb} q_1$, then not necessarily $p_0 \cdot p_1 \leftrightarrow^u_{tb} q_0 \cdot q_1$.

Example 7. *Let $p_0 = q_0 = a(1)$, $p_1 = \tau(0) \cdot b(1)$, $q_1 = b(1)$, and $u = 0$. Clearly, $a(1) \leftrightarrow^0_{tb} a(1)$. Also, $\tau(0) \cdot b(1) \leftrightarrow^0_{tb} b(1)$. However, clearly, $a(1) \cdot \tau(0) \cdot b(1) \leftrightarrow^0_{tb} a(1) \cdot b(1)$ does not hold.*

From Example 7, it follows that the standard approach to prove that untimed rooted branching bisimilarity is a congruence, i.e. take the smallest congruence closure and prove that this yields a branching bisimulation (see Fokkink (2000a)), fails for timed rooted branching bisimilarity when considering sequential composition. This motivates the usage of $p_1 \leftrightarrows_{rtb} q_1$ in Definition 36.

Definition 36 (Relation C_u). *Let $C_u \subseteq \mathscr{S} \times \mathscr{S}$ for $u \in \mathbb{T}$ denote the smallest relation such that:*

1. $\leftrightarrows_{tb}^{u} \subseteq C_u$;
2. *if* $p_0\ C_u\ q_0$ *and* $p_1 \leftrightarrows_{rtb} q_1$, *then* $p_0{\cdot}p_1\ C_u\ q_0{\cdot}q_1$;
3. *if* $p_0\ C_u \surd$ *and* $p_1 \leftrightarrows_{rtb} q_1$, *then* $p_0{\cdot}p_1\ C_u\ q_1$;
4. *if* $\surd\ C_u\ q_0$ *and* $p_1 \leftrightarrows_{rtb} q_1$, *then* $p_1\ C_u\ q_0{\cdot}q_1$.

Proposition 3. *The relations C_u constitute a timed branching bisimulation.*

Proof. The proof consists of three parts (plus three symmetric parts).

A *If $s\ C_u\ t$ and $s \xrightarrow{\ell}_u s'$, then we must prove, that*

  i *either $\ell = \tau$ and $s'\ C_u\ t$,*

  ii *or $t \Rightarrow_u \hat{t} \xrightarrow{\ell}_u t'$ with $s\ C_u\ \hat{t}$ and $s'\ C_u\ t'$.*

We apply induction on the structure of s and t. Since $s\ C_u\ t$, by Definition 36, we can distinguish four cases.

Firstly, $s \leftrightarrows_{tb}^{u} t$. The proof obligation follows directly from the definition of timed branching bisimilarity.

Secondly, $s = p_0{\cdot}p_1$ and $t = q_0{\cdot}q_1$, with $p_0\ C_u\ q_0$ and $p_1 \leftrightarrows_{rtb} q_1$. Since $p_0{\cdot}p_1 \xrightarrow{\ell}_u s'$, by the transition rules, we can distinguish two cases:

1. Let $p_0 \xrightarrow{\ell}_u p_0'$ and $s' = p_0'{\cdot}p_1$. Since $p_0\ C_u\ q_0$, by Definition 36, we can distinguish four cases:

   (a) $p_0 \leftrightarrows_{tb}^{u} q_0$. Since $p_0 \xrightarrow{\ell}_u p_0'$, by Definition 31, we can again distinguish two cases:

   i $\ell = \tau$ and $p_0' \leftrightarrows_{tb}^{u} q_0$. Then $p_0'{\cdot}p_1\ C_u\ q_0{\cdot}q_1$.

   ii $q_0 \Rightarrow_u \hat{q} \xrightarrow{\ell}_u t_0$, with $p_0 \leftrightarrows_{tb}^{u} \hat{q}$ and $p_0' \leftrightarrows_{tb}^{u} t_0$. Then,

   * either $t_0 \neq \surd$ and $q_0{\cdot}q_1 \Rightarrow_u \hat{q}{\cdot}q_1 \xrightarrow{\ell}_u t_0{\cdot}q_1$, $p_0{\cdot}p_1\ C_u\ \hat{q}{\cdot}q_1$ and $p_0'{\cdot}p_1\ C_u\ t_0{\cdot}q_1$;

- $*$ or $t_0 = \surd$ and $q_0{\cdot}q_1 \Rightarrow_u \hat{q}{\cdot}q_1 \xrightarrow{\ell}_u q_1$, $p_0{\cdot}p_1\ C_u\ \hat{q}{\cdot}q_1$ and $p_0'{\cdot}p_1\ C_u\ q_1$.

(b) $p_0 = p_{00}{\cdot}p_{01}$ and $q_0 = q_{00}{\cdot}q_{01}$ with $p_{00}\ C_u\ q_{00}$ and $p_{01} \mathrel{\underline{\leftrightarrow}}_{rtb} q_{01}$. Since $p_{00}{\cdot}p_{01} \xrightarrow{\ell}_u p_0'$, by the transition rules, either $p_{00} \xrightarrow{\ell}_u p_{00}'$ with $p_0' = p_{00}'{\cdot}p_{01}$, or $p_{00} \xrightarrow{\ell}_u \surd$ and $p_0' = p_{01}$.

In the first case, since $p_{00}\ C_u\ q_{00}$ and $p_{00} \xrightarrow{\ell}_u p_{00}'$, by induction,

i either $\ell = \tau$ and $p_{00}'\ C_u\ q_{00}$. Then $p_{00}'{\cdot}p_{01}{\cdot}p_1\ C_u\ q_{00}{\cdot}q_{01}{\cdot}q_1$.

ii or $q_{00} \Rightarrow_u \hat{q}_{00} \xrightarrow{\ell}_u t_{00}$, with $p_{00}\ C_u\ \hat{q}_{00}$ and $p_{00}'\ C_u\ t_{00}$. Then,

- $*$ either $t_{00} \neq \surd$ and $q_{00}{\cdot}q_{01} \Rightarrow_u \hat{q}_{00}{\cdot}q_{01} \xrightarrow{\ell}_u t_{00}{\cdot}q_{01}$, so $q_{00}{\cdot}q_{01}{\cdot}q_1 \Rightarrow_u \hat{q}_{00}{\cdot}q_{01}{\cdot}q_1 \xrightarrow{\ell}_u t_{00}{\cdot}q_{01}{\cdot}q_1$. Furthermore $p_{00}{\cdot}p_{01}{\cdot}p_1\ C_u\ \hat{q}_{00}{\cdot}q_{01}{\cdot}q_1$ and $p_{00}'{\cdot}p_{01}{\cdot}p_1\ C_u\ t_{00}{\cdot}q_{01}{\cdot}q_1$.
- $*$ or $t_{00} = \surd$ and $q_{00}{\cdot}q_{01} \Rightarrow_u \hat{q}_{00}{\cdot}q_{01} \xrightarrow{\ell}_u q_{01}$, so $q_{00}{\cdot}q_{01}{\cdot}q_1 \Rightarrow_u \hat{q}_{00}{\cdot}q_{01}{\cdot}q_1 \xrightarrow{\ell}_u q_{01}{\cdot}q_1$. Furthermore $p_{00}{\cdot}p_{01}{\cdot}p_1\ C_u\ \hat{q}_{00}{\cdot}q_{01}{\cdot}q_1$ and $p_{00}'{\cdot}p_{01}{\cdot}p_1\ C_u\ q_{01}{\cdot}q_1$.

In the second case, since $p_{00}\ C_u\ q_{00}$ and $p_{00} \xrightarrow{\ell}_u \surd$, by induction,

i either $\ell = \tau$ and $\surd\ C_u\ q_{00}$. Then $p_{01}{\cdot}p_1\ C_u\ q_{00}{\cdot}q_{01}{\cdot}q_1$.

ii or $q_{00} \Rightarrow_u \hat{q}_{00} \xrightarrow{\ell}_u t_{00}$ with $p_{00}\ C_u\ \hat{q}_{00}$ and $\surd\ C_u\ t_{00}$. Then,

- $*$ either $t_{00} \neq \surd$ and $q_{00}{\cdot}q_{01}{\cdot}q_1 \Rightarrow_u \hat{q}_{00}{\cdot}q_{01}{\cdot}q_1 \xrightarrow{\ell}_u t_{00}{\cdot}q_{01}{\cdot}q_1$. Furthermore $p_{00}{\cdot}p_{01}{\cdot}p_1\ C_u\ \hat{q}_{00}{\cdot}q_{01}{\cdot}q_1$ and $p_{01}{\cdot}p_1\ C_u\ t_{00}{\cdot}q_{01}{\cdot}q_1$.
- $*$ or $t_{00} = \surd$ and $q_{00}{\cdot}q_{01}{\cdot}q_1 \Rightarrow_u \hat{q}_{00}{\cdot}q_{01}{\cdot}q_1 \xrightarrow{\ell}_u q_{01}{\cdot}q_1$. Furthermore $p_{00}{\cdot}p_{01}{\cdot}p_1\ C_u\ \hat{q}_{00}{\cdot}q_{01}{\cdot}q_1$ and $p_{01}{\cdot}p_1\ C_u\ q_{01}{\cdot}q_1$.

(c) $p_0 = p_{00}{\cdot}p_{01}$ with $p_{00}\ C_u\ \surd$ and $p_{01} \mathrel{\underline{\leftrightarrow}}_{rtb} q_0$. Since $p_{00}{\cdot}p_{01} \xrightarrow{\ell}_u p_0'$, by the transition rules, either $p_{00} \xrightarrow{\ell}_u p_{00}'$ with $p_0' = p_{00}'{\cdot}p_{01}$, or $p_{00} \xrightarrow{\ell}_u \surd$ and $p_0' = p_{01}$.
In the first case, since $p_{00}\ C_u\ \surd$, by induction, $\ell = \tau$ and $p_{00}'\ C_u\ \surd$. Then $p_{00}'{\cdot}p_{01}{\cdot}p_1\ C_u\ q_0{\cdot}q_1$.

In the second case, since $p_{00}\ C_u\ \surd$, by induction, $\ell = \tau$. Moreover, $p_{01}{\cdot}p_1\ C_u\ q_0{\cdot}q_1$.

(d) $q_0 = q_{00}{\cdot}q_{01}$ with $\surd\ C_u\ q_{00}$ and $p_0 \mathrel{\underline{\leftrightarrow}}_{rtb} q_{01}$. Since $p_0 \mathrel{\underline{\leftrightarrow}}_{rtb} q_{01}$ and $p_0 \xrightarrow{\ell}_u p_0'$, by induction, $q_{01} \xrightarrow{\ell}_u q_{01}'$ with $p_0' \mathrel{\underline{\leftrightarrow}}_{tb}^u q_{01}'$. Then $p_0'{\cdot}p_1\ C_u\ q_{00}{\cdot}q_{01}'{\cdot}q_1$.

2. Let $p_0 \xrightarrow{\ell}_u \surd$ and $s' = p_1$. Since $p_0\ C_u\ q_0$, by induction,

i either $\ell = \tau$ and $\surd\ C_u\ q_0$. Then $p_1\ C_u\ q_0{\cdot}q_1$.

ii or $q_0 \Rightarrow_u \hat{q}_0 \xrightarrow{\ell}_u t_0$ with $p_0\ C_u\ \hat{q}_0$ and $\surd\ C_u\ t_0$. Then,

- $*$ either $t_0 \neq \surd$ and $q_0{\cdot}q_1 \Rightarrow_u \hat{q}_0{\cdot}q_1 \xrightarrow{\ell}_u t_0{\cdot}q_1$. Furthermore $p_0{\cdot}p_1\ C_u\ \hat{q}_0{\cdot}q_1$ and $p_1\ C_u\ t_0{\cdot}q_1$.
- $*$ or $t_0 = \surd$ and $q_0{\cdot}q_1 \Rightarrow_u \hat{q}_0{\cdot}q_1 \xrightarrow{\ell}_u q_1$. Furthermore $p_0{\cdot}p_1\ C_u\ \hat{q}_0{\cdot}q_1$ and $p_1\ C_u\ q_1$.

Thirdly, $s = p_0{\cdot}p_1$, with $p_0\ C_u\ \surd$ and $p_1 \mathrel{\underline{\leftrightarrow}}_{rtb} t$. Since $p_0{\cdot}p_1 \xrightarrow{\ell}_u s'$, by the transition rules, we can distinguish two cases:

1. $p_0 \xrightarrow{\ell}_u p_0'$ and $s' = p_0'{\cdot}p_1$. Since $p_0\ C_u\ \surd$, by induction, $\ell = \tau$ and $p_0'\ C_u\ \surd$. Then $p_0'{\cdot}p_1\ C_u\ t$.
2. $p_0 \xrightarrow{\ell}_u \surd$ and $s' = p_1$. Since $p_0\ C_u\ \surd$, by induction, $\ell = \tau$. And $p_1 \mathrel{\underline{\leftrightarrow}}_{rtb} t$ clearly implies $p_1\ C_u\ t$.

Fourthly, $t = q_0{\cdot}q_1$, with $\surd\ C_u\ q_0$ and $s \mathrel{\underline{\leftrightarrow}}_{rtb} q_1$. Since $\surd\downarrow$, by induction, $q_0 \Rightarrow_u t_0 \downarrow$. Clearly, $t_0 = \surd$. Since $s \mathrel{\underline{\leftrightarrow}}_{rtb} q_1$, $s \xrightarrow{\ell}_u s'$ implies $q_1 \xrightarrow{\ell}_u t'$ with $s' \mathrel{\underline{\leftrightarrow}}_{tb}^u t'$. So $q_0{\cdot}q_1 \Rightarrow_u q_1 \xrightarrow{\ell}_u t'$.

B *If $s\ C_u\ t$ and $s\downarrow$, then we must prove, that $t \Rightarrow_u t' \downarrow$, with $s\ C_u\ t'$.*

We apply induction on the structure of s and t. Since $s\ C_u\ t$, by Definition 36, we can distinguish four cases:

Firstly, $s \mathrel{\underline{\leftrightarrow}}_{tb}^u t$. The proof obligation follows directly from the definition of timed branching bisimilarity.

Secondly, $s = p_0{\cdot}p_1$ and $t = q_0{\cdot}q_1$, with $p_0\ C_u\ q_0$ and $p_1 \mathrel{\underline{\leftrightarrow}}_{rtb} q_1$. This case is vacuous, since $s = p_0{\cdot}p_1$ contradicts $s\downarrow$.

Thirdly, $s = p_0{\cdot}p_1$, with $p_0\ C_u\ \surd$ and $p_1 \mathrel{\underline{\leftrightarrow}}_{rtb} t$. This case is vacuous, since $s = p_0{\cdot}p_1$ contradicts $s\downarrow$.

Fourthly, $t = q_0{\cdot}q_1$, with $\surd\ C_u\ q_0$ and $s \mathrel{\underline{\leftrightarrow}}_{rtb} q_1$. This case is vacuous, since $s\downarrow$ and $s \mathrel{\underline{\leftrightarrow}}_{rtb} q_1$ implies $q_1\downarrow$, which is not possible.

C *If $s\ C_u\ t$ and $u \leq v$ and $\mathcal{U}(s,v)$, then we must prove, that for some $n \geq 0$ there are $t_0,\ldots,t_n \in \mathcal{S}$ with $t = t_0$ and $\mathcal{U}(t_n,v)$, and $u_0,\ldots,u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $t_i \Rightarrow_{u_i} t_{i+1}$ and $s\ C_w\ t_{i+1}$ for $u_i \leq w \leq u_{i+1}$.*

We apply induction on the structure of s and t. Since $s\ C_u\ t$, by Definition 36, we can distinguish four cases:

Firstly, $s \underline{\leftrightarrow}^u_{tb} t$. The proof obligation follows directly from the definition of timed branching bisimilarity.

Secondly, $s = p_0{\cdot}p_1$ and $t = q_0{\cdot}q_1$, with $p_0\ C_u\ q_0$ and $p_1 \underline{\leftrightarrow}_{rtb} q_1$. Since $\mathscr{U}(p_0{\cdot}p_1, v)$, also $\mathscr{U}(p_0, v)$. Since $p_0\ C_u\ q_0$ and $u \leq v$, by induction, for some $n \geq 0$ there are $\hat{q}_0, \ldots, \hat{q}_n \in \mathscr{S}$ with $q_0 = \hat{q}_0$ and $\mathscr{U}(\hat{q}_n, v)$, and $u_0, \ldots, u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $\hat{q}_i \Rightarrow_{u_i} \hat{q}_{i+1}$ and $p_0\ C_w\ \hat{q}_{i+1}$ for $u_i \leq w \leq u_{i+1}$. Clearly, $t = \hat{q}_0{\cdot}q_1$ and $\mathscr{U}(\hat{q}_n{\cdot}q_1, v)$, and for $i < n$, $\hat{q}_i{\cdot}q_1 \Rightarrow_{u_i} \hat{q}_{i+1}{\cdot}q_1$ and $p_0{\cdot}p_1\ C_w\ \hat{q}_{i+1}{\cdot}q_1$ for $u_i \leq w \leq u_{i+1}$.

Thirdly, $s = p_0{\cdot}p_1$ with $p_0\ C_u\ \surd$ and $p_1 \underline{\leftrightarrow}_{rtb} t$. Since $\mathscr{U}(s, v)$, also $\mathscr{U}(p_0, v)$. Since $v > u$, by case C, this contradicts $p_0\ C_u\ \surd$, so this case is vacuous.

Fourthly, $t = q_0{\cdot}q_1$ with $\surd\ C_u\ q_0$ and $s \underline{\leftrightarrow}_{rtb} q_1$. Since $\surd\ C_u\ q_0$, by case C, $q_0 \Rightarrow_u \surd$ and $\neg\mathscr{U}(q_0, v)$ for $v \geq u$. So $q_0{\cdot}q_1 \Rightarrow_u q_1$. Since $s \underline{\leftrightarrow}_{rtb} q_1$ and $\mathscr{U}(s, v)$, also $\mathscr{U}(q_1, v)$. Clearly, the proof obligation holds with $n = 1$.

□

Theorem 3. *If $p_0 \underline{\leftrightarrow}_{rtb} q_0$ and $p_1 \underline{\leftrightarrow}_{rtb} q_1$, then $p_0{\cdot}p_1 \underline{\leftrightarrow}_{rtb} q_0{\cdot}q_1$.*

Proof. By Definition 34, we distinguish four cases:

1. Let $p_0{\cdot}p_1 \xrightarrow{\ell}_u s$. By the transition rules, we can distinguish two cases:
   a) $p_0 \xrightarrow{\ell}_u p_0'$ and $s = p_0'{\cdot}p_1$. Since $p_0 \underline{\leftrightarrow}_{rtb} q_0$, $q_0 \xrightarrow{\ell}_u t$ with $p_0' \underline{\leftrightarrow}^u_{tb} t$. By the transition rules, we can distinguish two cases:
      i. Either $t \neq \surd$ and $q_0{\cdot}q_1 \xrightarrow{\ell}_u t{\cdot}q_1$. By proposition 3, $p_0'{\cdot}p_1 \underline{\leftrightarrow}^u_{tb} t{\cdot}q_1$.
      ii. Or $t = \surd$ and $q_0{\cdot}q_1 \xrightarrow{\ell}_u q_1$. By proposition 3, $p_0'{\cdot}p_1 \underline{\leftrightarrow}^u_{tb} q_1$.
   b) $p_0 \xrightarrow{\ell}_u \surd$ and $s = p_1$. Since $p_0 \underline{\leftrightarrow}_{rtb} q_0$, $q_0 \xrightarrow{\ell}_u t$ with $\surd \underline{\leftrightarrow}^u_{tb} t$. By the transition rules, we can distinguish two cases:
      i. Either $t \neq \surd$ and $q_0{\cdot}q_1 \xrightarrow{\ell}_u t{\cdot}q_1$. By proposition 3, $p_1 \underline{\leftrightarrow}^u_{tb} t{\cdot}q_1$.
      ii. Or $t = \surd$ and $q_0{\cdot}q_1 \xrightarrow{\ell}_u q_1$. Since $p_1 \underline{\leftrightarrow}_{rtb} q_1$, $p_1 \underline{\leftrightarrow}^u_{tb} q_1$.
2. Let $q_0{\cdot}q_1 \xrightarrow{\ell}_u t$. Similar to the previous case.
3. Let $\mathscr{U}(p_0{\cdot}p_1, u)$. Then $\mathscr{U}(p_0, u)$. Since $p_0 \underline{\leftrightarrow}_{rtb} q_0$, $\mathscr{U}(q_0, u)$. This means that $\mathscr{U}(q_0{\cdot}q_1, u)$.
4. Let $\mathscr{U}(q_0{\cdot}q_1, u)$. Similar to the previous case.

□

11.6.4 Congruence Proof for Alternative Composition

Next, we prove that rooted timed branching bisimilarity is a congruence for the alternative composition operator.

Theorem 4. *If $p_0 \leftrightarrows_{rtb} q_0$ and $p_1 \leftrightarrows_{rtb} q_1$, then $p_0 + p_1 \leftrightarrows_{rtb} q_0 + q_1$.*

Proof. By Definition 34, we distinguish four cases:

1. Let $p_0 + p_1 \xrightarrow{\ell}_u s$. By the transition rules, we can distinguish two cases:
   a) $p_0 \xrightarrow{\ell}_u s$. Since $p_0 \leftrightarrows_{rtb} q_0$, $q_0 \xrightarrow{\ell}_u t$ with $s \leftrightarrows^u_{tb} t$. Then $q_0 + q_1 \xrightarrow{\ell}_u t$.

   b) $p_1 \xrightarrow{\ell}_u s$. Similar to the previous case.
2. Let $q_0 + q_1 \xrightarrow{\ell}_u t$. Similar to the previous case.
3. Let $\mathcal{U}(p_0 + p_1, u)$. Since $\mathcal{U}(p_0 + p_1, u)$, either $\mathcal{U}(p_0, u)$ or $\mathcal{U}(p_1, u)$. Since $p_0 \leftrightarrows_{rtb} q_0$ and $p_1 \leftrightarrows_{rtb} q_1$, either $\mathcal{U}(q_0, u)$, or $\mathcal{U}(q_1, u)$, respectively. Hence, $\mathcal{U}(q_0 + q_1, u)$.
4. Let $\mathcal{U}(q_0 + q_1, u)$. Similar to the previous case.

□

11.6.5 Congruence Proof for Parallel Composition

Finally, we indicate how to prove that rooted timed branching bisimilarity is a congruence for the parallel composition operator. This proof largely follows the one for sequential composition.

Definition 37 (Relation D_u). *Let $D_u \subseteq \mathcal{S} \times \mathcal{S}$ for $u \in \mathbb{T}$ denote the smallest relation such that:*

1. *$\leftrightarrows^u_{tb} \subseteq D_u$;*
2. *if $p_0\, D_u\, q_0$ and $p_1\, D_u\, q_1$, then $p_0 \parallel p_1\, D_u\, q_0 \parallel q_1$;*
3. *if $p_0\, D_u \surd$ and $p_1\, D_u\, q_1$, then $p_0 \parallel p_1\, D_u\, q_1$;*
4. *if $p_0\, D_u\, q_0$ and $p_1\, D_u \surd$, then $p_0 \parallel p_1\, D_u\, q_0$;*
5. *if $\surd\, D_u\, q_0$ and $p_1\, D_u\, q_1$, then $p_1\, D_u\, q_0 \parallel q_1$;*
6. *if $p_0\, D_u\, q_0$ and $\surd\, D_u\, q_1$, then $p_0\, D_u\, q_0 \parallel q_1$;*
7. *if $p_0\, D_u \surd$ and $p_1\, D_u \surd$, then $p_0 \parallel p_1\, D_u \surd$;*
8. *if $\surd\, D_u\, q_0$ and $\surd\, D_u\, q_1$, then $\surd\, D_u\, q_0 \parallel q_1$.*

Lemma 9. *If $p\ D_u\ q$ and $\mathscr{U}(p,u)$, then $\mathscr{U}(q,u)$.*

Proof. By Definition 37, we can distinguish six cases (the last two of Definition 37 are not applicable):

1. $p \mathrel{\underline{\leftrightarrow}}^u_{tb} q$. Then it follows immediately from Definition 31, case 6, that $\mathscr{U}(q,u)$.[5]
2. $p = p_0 \parallel p_1$ and $q = q_0 \parallel q_1$, with $p_0\ D_u\ q_0$ and $p_1\ D_u\ q_1$. Since $\mathscr{U}(p_0 \parallel p_1, u)$, we have $\mathscr{U}(p_0,u)$ and $\mathscr{U}(p_1,u)$. Therefore, by induction, $\mathscr{U}(q_0,u)$ and $\mathscr{U}(q_1,u)$. From this, it follows that $\mathscr{U}(q_0 \parallel q_1, u)$.
3. $p = p_0 \parallel p_1$, with $p_0\ D_u \surd$ and $p_1\ D_u\ q$. Since $\mathscr{U}(p_0 \parallel p_1, u)$, we have $\mathscr{U}(p_1,u)$. Therefore, by induction, $\mathscr{U}(q,u)$.
4. $p = p_0 \parallel p_1$, with $p_0\ D_u\ q$ and $p_1\ D_u \surd$. Similar to the previous case.
5. $q = q_0 \parallel q_1$, with $\surd\ D_u\ q_0$ and $p\ D_u\ q_1$. By Definition 37, $\surd \mathrel{\underline{\leftrightarrow}}^u_{tb} q_0$. Since $q_0 \neq \surd$, it follows from Definition 31, case 3, that $\mathscr{U}(q_0,u)$. Since $\mathscr{U}(p,u)$, by induction, $\mathscr{U}(q_1,u)$. From $\mathscr{U}(q_0,u)$ and $\mathscr{U}(q_1,u)$, it follows that $\mathscr{U}(q_0 \parallel q_1, u)$.
6. $q = q_0 \parallel q_1$, with $p\ D_u\ q_0$ and $\surd\ D_u\ q_1$. Similar to the previous case.

□

In the forthcoming proofs of Proposition 4 and Theorem 5, all cases that involve successful termination have been discarded. We point out that the full proof contains at least 528 different cases.

Proposition 4. *The relations D_u constitute a timed branching bisimulation.*

Proof. The proof consists of two parts (plus two symmetric parts); The proof consists of three parts (plus three symmetric parts) if we would take into account successful termination.

A *If $s\ D_u\ t$ and $s \xrightarrow{\ell}_u s'$, then we must prove, that*

i *either $\ell = \tau$ and $s'\ D_u\ t$,*

ii *or $t \Rightarrow_u \hat{t} \xrightarrow{\ell}_u t'$ with $s\ D_u\ \hat{t}$ and $s'\ D_u\ t'$.*

We apply induction on the structure of s and t. Since $s\ D_u\ t$, by Definition 37, we can distinguish two cases (eight if we consider successful termination).

Firstly, $s \mathrel{\underline{\leftrightarrow}}^u_{tb} t$. The proof obligation follows directly from the definition of timed branching bisimilarity.

[5] Note that this holds due to '$u \leq v$' and '$n \geq 0$' in case 6, because in the original definition of Van der Zwaag, we would have $\delta(1) \mathrel{\underline{\leftrightarrow}}^{Z,2}_{tb} \delta(2)$; see footnote 3.

Secondly, $s = p_0 \parallel p_1$ and $t = q_0 \parallel q_1$, with $p_0 \; D_u \; q_0$ and $p_1 \; D_u \; q_1$. Since $p_0 \parallel p_1 \xrightarrow{\ell}_u s'$, by the transition rules, we can distinguish three cases (eight if we consider successful termination):

1. Let $p_0 \xrightarrow{\ell}_u p'_0$, $\mathscr{U}(p_1, u)$ and $s' = p'_0 \parallel p_1$. By Lemma 9, since $\mathscr{U}(p_1, u)$, also $\mathscr{U}(q_1, u)$. Since $p_0 \; D_u \; q_0$, by Definition 37, we can distinguish two cases (eight if we consider successful termination):

   (a) $p_0 \underline{\leftrightarrow}^u_{tb} q_0$. Since $p_0 \xrightarrow{\ell}_u p'_0$, by Definition 31, we can again distinguish two cases:

      i $\ell = \tau$ and $p'_0 \underline{\leftrightarrow}^u_{tb} q_0$. Then $p'_0 \parallel p_1 \; D_u \; q_0 \parallel q_1$.

      ii $q_0 \Rightarrow_u \hat{q} \xrightarrow{\ell}_u t_0$, with $p_0 \underline{\leftrightarrow}^u_{tb} \hat{q}$ and $p'_0 \underline{\leftrightarrow}^u_{tb} t_0$. Then, since $\mathscr{U}(q_1, u)$ (and we do not consider successful termination, hence $t_0 \neq \surd$), it follows that $q_0 \parallel q_1 \Rightarrow_u \hat{q} \parallel q_1 \xrightarrow{\ell}_u t_0 \parallel q_1$, $p_0 \parallel p_1 \; D_u \; \hat{q} \parallel q_1$ and $p'_0 \parallel p_1 \; D_u \; t_0 \parallel q_1$.

   (b) $p_0 = p_{00} \parallel p_{01}$ and $q_0 = q_{00} \parallel q_{01}$ with $p_{00} \; D_u \; q_{00}$ and $p_{01} \; D_u \; q_{01}$. Since $p_{00} \parallel p_{01} \xrightarrow{\ell}_u p'_0$, by the transition rules (not considering successful termination), either $p_{00} \xrightarrow{\ell}_u p'_{00}$ and $\mathscr{U}(p_{01}, u)$ with $p'_0 = p'_{00} \parallel p_{01}$, or $p_{01} \xrightarrow{\ell}_u p'_{01}$ and $\mathscr{U}(p_{00}, u)$ with $p'_0 = p_{00} \parallel p'_{01}$, or $p_{00} \parallel p_{01} \xrightarrow{\ell}_u p'_{00} \parallel p'_{01}$ with $p'_0 = p'_{00} \parallel p'_{01}$ if there exist $\ell_0, \ell_1 \in \mathscr{A}$ such that $p_{00} \xrightarrow{\ell_0}_u p'_{00}$, $p_{01} \xrightarrow{\ell_1}_u p'_{01}$, and $\ell_0 \mid \ell_1 = \ell$.

   In the first case, by Lemma 9, since $\mathscr{U}(p_{01}, u)$, also $\mathscr{U}(q_{01}, u)$. Since $p_{00} \; D_u \; q_{00}$ and $p_{00} \xrightarrow{\ell}_u p'_{00}$, by induction,

      i either $\ell = \tau$ and $p'_{00} \; D_u \; q_{00}$. Then $p'_{00} \parallel p_{01} \parallel p_1 \; D_u \; q_{00} \parallel q_{01} \parallel q_1$.

      ii or $q_{00} \Rightarrow_u \hat{q}_{00} \xrightarrow{\ell}_u t_{00}$, with $p_{00} \; D_u \; \hat{q}_{00}$ and $p'_{00} \; D_u \; t_{00}$. Then (we do not consider successful termination here, hence $t_{00} \neq \surd$), $q_{00} \parallel q_{01} \Rightarrow_u \hat{q}_{00} \parallel q_{01} \xrightarrow{\ell}_u t_{00} \parallel q_{01}$, so, since $\mathscr{U}(q_{01} \parallel q_1, u)$, $q_{00} \parallel q_{01} \parallel q_1 \Rightarrow_u \hat{q}_{00} \parallel q_{01} \parallel q_1 \xrightarrow{\ell}_u t_{00} \parallel q_{01} \parallel q_1$. Furthermore $p_{00} \parallel p_{01} \parallel p_1 \; D_u \; \hat{q}_{00} \parallel q_{01} \parallel q_1$ and $p'_{00} \parallel p_{01} \parallel p_1 \; D_u \; t_{00} \parallel q_{01} \parallel q_1$.

   The second case is similar to the first case.

   In the third case, since $p_{00} \; D_u \; q_{00}$, $p_{00} \xrightarrow{\ell_0}_u p'_{00}$, and $\ell_0 \neq \tau$ (since $\ell_0 \mid \ell_1 = \ell$), by induction, $q_{00} \Rightarrow_u \hat{q}_{00} \xrightarrow{\ell_0}_u t_{00}$, with $p_{00} \; D_u \; \hat{q}_{00}$ and $p'_{00} \; D_u \; t_{00}$. Similarly, since $p_{01} \; D_u \; q_{01}$, $p_{01} \xrightarrow{\ell_1}_u p'_{01}$, and $\ell_1 \neq \tau$ (since $\ell_0 \mid \ell_1 = \ell$), by induction, $q_{01} \Rightarrow_u \hat{q}_{01} \xrightarrow{\ell_0}_u t_{01}$. Then (we do not consider successful termination here, hence $t_{00} \neq \surd$ and $t_{01} \neq \surd$), it follows that $q_{00} \parallel q_{01} \Rightarrow_u \hat{q}_{00} \parallel \hat{q}_{01} \xrightarrow{\ell}_u t_{00} \parallel t_{01}$. Since $\mathcal{U}(q_1, u)$, $q_{00} \parallel q_{01} \parallel q_1 \Rightarrow_u \hat{q}_{00} \parallel \hat{q}_{01} \parallel q_1 \xrightarrow{\ell}_u t_{00} \parallel t_{01} \parallel q_1$. Furthermore $p_{00} \parallel p_{01} \parallel p_1 \, D_u \, \hat{q}_{00} \parallel \hat{q}_{01} \parallel q_1$ and $p'_{00} \parallel p'_{01} \parallel p_1 \, D_u \, t_{00} \parallel t_{01} \parallel q_1$.

2. Let $p_1 \xrightarrow{\ell}_u p'_1$, $\mathcal{U}(p_0, u)$ and $s' = p_0 \parallel p'_1$. This case is similar to the previous case.
3. Let $p_0 \parallel p_1 \xrightarrow{\ell}_u p'_0 \parallel p_1$ with $\ell_0, \ell_1 \in \mathcal{A}$ such that $p_0 \xrightarrow{\ell_0}_u p'_0$, $p_1 \xrightarrow{\ell_1}_u p'_1$, and $\ell_0 \mid \ell_1 = \ell$. Since $p_0 \, D_u \, q_0$, $p_0 \xrightarrow{\ell_0}_u p'_0$, and $\ell_0 \neq \tau$ (since $\ell_0 \mid \ell_1 = \ell$), by induction, $q_0 \Rightarrow_u \hat{q}_0 \xrightarrow{\ell_0}_u t_0$, with $p_0 \, D_u \, \hat{q}_0$ and $p'_0 \, D_u \, t_0$. Similarly, since $p_1 \, D_u \, q_1$, $p_1 \xrightarrow{\ell_1}_u p'_1$, and $\ell_1 \neq \tau$ (since $\ell_0 \mid \ell_1 = \ell$), by induction, $q_1 \Rightarrow_u \hat{q}_1 \xrightarrow{\ell_1}_u t_1$. Then (we do not consider successful termination here, hence $t_0 \neq \surd$ and $t_1 \neq \surd$), it follows that $q_0 \parallel q_1 \Rightarrow_u \hat{q}_0 \parallel \hat{q}_1 \xrightarrow{\ell}_u t_0 \parallel t_1$. Furthermore $p_0 \parallel p_1 \, D_u \, \hat{q}_0 \parallel \hat{q}_1$ and $p'_0 \parallel p'_1 \, D_u \, t_0 \parallel t_1$.

B *If $s \, D_u \, t$ and $u \leq v$ and $\mathcal{U}(s, v)$, then we must prove, that for some $n \geq 0$ there are $t_0,\ldots,t_n \in \mathcal{S}$ with $t = t_0$ and $\mathcal{U}(t_n, v)$, and $u_0,\ldots,u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $t_i \Rightarrow_{u_i} t_{i+1}$ and $s \, D_w \, t_{i+1}$ for $u_i \leq w \leq u_{i+1}$.*

We apply induction on the structure of s and t. Since $s \, D_u \, t$, by Definition 37, we can distinguish two cases (eight if we consider successful termination).

Firstly, $s \leftrightarrow^{u}_{tb} t$. The proof obligation follows directly from the definition of timed branching bisimilarity.

Secondly, $s = p_0 \parallel p_1$ and $t = q_0 \parallel q_1$, with $p_0 \, D_u \, q_0$ and $p_1 \, D_u \, q_1$. Since $\mathcal{U}(p_0 \parallel p_1, v)$, also $\mathcal{U}(p_0, v)$ and $\mathcal{U}(p_1, v)$. Since $p_0 \, D_u \, q_0$ and $u \leq v$, by induction, for some $n \geq 0$ there are $\hat{q}_0, \ldots, \hat{q}_n \in \mathcal{S}$ with $q_0 = \hat{q}_0$ and $\mathcal{U}(\hat{q}_n, v)$, and $u_0, \ldots, u_n \in \mathbb{T}$ with $u = u_0$ and $v = u_n$, such that for $i < n$, $\hat{q}_i \Rightarrow_{u_i} \hat{q}_{i+1}$ and $p_0 \, D_w \, \hat{q}_{i+1}$ for $u_i \leq w \leq u_{i+1}$. Similarly, since $p_1 \, D_u \, q_1$ and $u \leq v$, by induction, for some $m \geq 0$ there are $\hat{q}'_0, \ldots, \hat{q}'_m \in \mathcal{S}$ with $q_1 = \hat{q}'_0$ and $\mathcal{U}(\hat{q}'_m, v)$, and $u'_0, \ldots, u'_m \in \mathbb{T}$ with $u = u'_0$ and $v = u'_m$, such that for $i < m$, $\hat{q}'_i \Rightarrow_{u'_i} \hat{q}'_{i+1}$ and $p_1 \, D_w \, \hat{q}'_{i+1}$ for $u'_i \leq w \leq u'_{i+1}$. Clearly, these two sequences for $\hat{q}_0, \ldots, \hat{q}_n$ and for $\hat{q}'_0, \ldots, \hat{q}'_m$ can in a straightforward fashion be transformed into a sequence, such that for $k = n + m$ there are $\bar{q}_0, \ldots, \bar{q}_k \in \mathcal{S}$ and $\bar{q}'_0, \ldots, \bar{q}'_k \in \mathcal{S}$ with $q_0 = \bar{q}_0$, $\mathcal{U}(\bar{q}_k, v)$, $q_1 = \bar{q}'_0$, $\mathcal{U}(\bar{q}'_k, v)$, and $u_0, \ldots, u_k \in \mathbb{T}$ with $u = u_0$ and $v = u_k$, such that for $i < k$, $\bar{q}_i \parallel \bar{q}'_i \Rightarrow_{u_i} \bar{q}_{i+1} \parallel \bar{q}'_{i+1}$ and $p_0 \parallel p_1 \, D_w \, \bar{q}_{i+1} \parallel \bar{q}'_{i+1}$ for $u_i \leq w \leq u_{i+1}$.

□

Theorem 5. *If $p_0 \leftrightarrow_{rtb} q_0$ and $p_1 \leftrightarrow_{rtb} q_1$, then $p_0 \parallel p_1 \leftrightarrow_{rtb} q_0 \parallel q_1$.*

Proof. By Definition 34, we distinguish four cases:

1. Let $p_0 \parallel p_1 \xrightarrow{\ell}_u s$. By the transition rules, we can distinguish three cases (eight if we consider successful termination):
   a) $p_0 \xrightarrow{\ell}_u p'_0$ and $s = p'_0 \parallel p_1$. Since $p_0 \underline{\leftrightarrow}_{rtb}\, q_0$, $q_0 \xrightarrow{\ell}_u t$ with $p'_0 \underline{\leftrightarrow}^u_{tb}\, t$. Since we do not consider successful termination ($t \neq \surd$), by the transition rules, $q_0 \parallel q_1 \xrightarrow{\ell}_u t \parallel q_1$. By proposition 4, $p'_0 \parallel p_1 \underline{\leftrightarrow}^u_{tb}\, t \parallel q_1$.

   b) $p_1 \xrightarrow{\ell}_u p'_1$ and $s = p_0 \parallel p'_1$. Similar to the previous case.

   c) $p_0 \parallel p_1 \xrightarrow{\ell}_u p'_0 \parallel p'_1$ with $\ell_0, \ell_1 \in \mathscr{A}$ such that $p_0 \xrightarrow{\ell_0}_u p'_0$, $p_1 \xrightarrow{\ell_1}_u p'_1$, and $\ell_0 \mid \ell_1 = \ell$. Since $p_0 \underline{\leftrightarrow}_{rtb}\, q_0$, $q_0 \xrightarrow{\ell_0}_u t_0$ with $p'_0 \underline{\leftrightarrow}^u_{rtb}\, t_0$. Since $p_1 \underline{\leftrightarrow}_{rtb}\, q_1$, $q_1 \xrightarrow{\ell_0}_u t_1$ with $p'_1 \underline{\leftrightarrow}^u_{rtb}\, t_1$. Since we do not consider successful termination ($t_0 \neq \surd$ and $t_1 \neq \surd$), by the transition rules, $q_0 \parallel q_1 \xrightarrow{\ell}_u t_0 \parallel t_1$. By proposition 4, $p'_0 \parallel p'_1 \underline{\leftrightarrow}^u_{tb}\, t_0 \parallel t_1$.

2. Let $q_0 \parallel q_1 \xrightarrow{\ell}_u t$. Similar to the previous case.

3. Let $\mathcal{U}(p_0 \parallel p_1, u)$. Then $\mathcal{U}(p_0, u)$ and $\mathcal{U}(p_1, u)$. Since $p_0 \underline{\leftrightarrow}_{rtb}\, q_0$ and $p_1 \underline{\leftrightarrow}_{rtb}\, q_1$, $\mathcal{U}(q_0, u)$ and $\mathcal{U}(q_1, u)$. This means that $\mathcal{U}(q_0 \parallel q_1, u)$.

4. Let $\mathcal{U}(q_0 \parallel q_1, u)$. Similar to the previous case.

□

11.7 Future Work

We conclude the chapter by pointing out some possible research directions for the future.

1. Van der Zwaag (2001) extended the cones and foci verification method of Groote and Springintveld (2001) to TLTSs. Fokkink and Pang (2005) proposed an adaptation of this timed cones and foci method. Both papers take $\underline{\leftrightarrow}^Z_{tb}$ as a starting point. It should be investigated whether a timed cones and foci method can be formulated for $\underline{\leftrightarrow}_{tb}$ as defined in the current chapter.

2. Van Glabbeek (1993) presented a wide range of concurrency semantics for untimed processes with the silent step τ. It would be a challenge to try and formulate timed versions of these semantics, and prove equivalence and congruence properties for the resulting timed semantics.

Appendix A

Preserving Behaviour when Transforming μCRL^{tick} to μCRL

IN THIS APPENDIX, WE PROVE that the behaviour of a μCRL^{tick} specification $\mathcal{M}_{\top}$ is preserved in a μCRL specification $\mathcal{M}$ resulting from a transformation of $\mathcal{M}_{\top}$. Since the sets of axioms and transition rules of μCRL are subsets of the sets of axioms and transition rules of μCRL^{tick}, the majority of the work here involves proving that the axioms and transition rules concerning the clock actions, as stated in Tables 5.1 and 5.2, can be 'observed' in μCRL specifications stemming from the transformation of a μCRL^{tick} specification, i.e. even though these axioms and transition rules do not explicitly exist for μCRL, in this particular subclass of μCRL specifications they do hold. We denote the transition rules as TR1,...,TR26. If $X = a(i)$, $Y(m) = a(m)$ and $m = i$, we say that $X = Y(m)$. Furthermore, since in practice, μCRL (and therefore μCRL^{tick}) processes do not (syntactically) contain successful termination, we restrict the proof to those cases in Table 5.2 where the process terms x and y do not lead to $\surd$ after firing a transition. First focussing on the axioms and transition rules shared by μCRL and μCRL^{tick}, we begin by proving that an untimed system remains unchanged after transformation. This relates to *semantics conservation* as defined by Nicollin and Sifakis (1991). This notion is defined in Definition 38. Following, Proposition 5 describes what needs to be shown to prove that semantics conservation holds for μCRL^{tick}.

Definition 38 (Semantics conservation (Nicollin and Sifakis, 1991)). *Considering an untimed process algebra UPA and a timed version of this algebra TPA,* semantics conservation *holds for these algebras iff an untimed UPA process and its time-equivalent TPA process have the same behaviour as long as we observe execution of actions only. This imposes that the rules of UPA remain valid in TPA, as far as they are applied on terms of UPA.*

Proposition 5 (Semantics conservation of μCRL^{tick}). *Given a μCRL^{tick} specification $\mathcal{M}_{\top}$ without delays, then $\mathcal{M}_{\top}$ behaves as its transformation $\mathcal{M}$.*

Proof. Follows from the form of the $TX \in \mathcal{P}$ and $\mathcal{I}$. Say that $\mathcal{M}_{\top}$ consists of a number of processes without delays. For each process P, in the transformed process TP, all actions, with their corresponding parameters, conditions and recursive calls are copied from P. Since $I_C^P = \emptyset$, furthermore, all other lines in TP are disabled, except for the *tock* alternatives. Since this holds for all processes in $\mathcal{M}_\mathrm{T}$, communications of clock actions can only result in *tock* actions, which are in the end encapsulated in $\mathcal{I}$. □

Proposition 6 (DRT1). *Given a μCRL^{tick} process $X = tick(n) \cdot Y$, with $n < 0$. Then TX behaves as $T\delta$.*

Proof. If we transform X to TX, we get $TX = ring \cdot T\hat{X} \lhd n = 0 \rhd \delta + tick(\theta(\mathrm{T}, n)) \cdot TX'(\mathrm{T} \to n - t, n) \lhd \theta(\mathrm{T}, n) \neq \mathsf{c} \rhd \delta + \sum_{t \in \mathbb{T}} tock(t) \cdot TX'(\mathrm{T} \to n - t, n) \lhd 0 < t < \theta(\mathrm{T}, n) \rhd \delta$. Since $n \neq 0$, the *ring* option is disabled. Furthermore, since $\theta(\mathrm{T}, n) = \mathsf{c}$, the *tick* option is also disabled. So, practically, $TX = \sum_{t \in \mathbb{T}} tock(t) \cdot TX'(n - t) \lhd 0 < t < \mathsf{c} \rhd \delta$. Similarly, the *ring* and *tick* options in TX' are disabled, and since $I_D^X = \emptyset$, we practically have $TX' = \sum_{t \in \mathbb{T}} tock(t) \cdot TX'(n - t) \lhd 0 < t < \mathsf{c} \rhd \delta$. Clearly, we have $T\delta = \delta + \sum_{t \in \mathbb{T}} tock(t) \cdot T\delta' \lhd 0 < t < \mathsf{c} \rhd \delta$, and $T\delta' = \delta + \sum_{t \in \mathbb{T}} tock(t) \cdot T\delta' \lhd 0 < t < \mathsf{c} \rhd \delta$. Since the parameter of TX' does not have any behavioural effect and δ is never an option to fire, these two systems behave in the same way. □

Proposition 7 (DRT2). *Given two μCRL^{tick} processes $Z_0 = tick(n) \cdot X + tick(n) \cdot Y$ and $Z_1 = tick(n) \cdot (X + Y)$. Then, TZ_0 behaves as TZ_1.*

Proof. The transformation leads to the following:

$$
\begin{aligned}
TZ_0 = {} & ring \cdot T\hat{Z}_0 \lhd n = 0 \rhd \delta + \\
& tick(\theta(\mathrm{T}, n)) \cdot TZ_0'(n - \theta(\mathrm{T}, n), n - \theta(\mathrm{T}, n)) \lhd \theta(\mathrm{T}, n) \neq \mathsf{c} \rhd \delta + \\
& \sum_{t \in \mathbb{T}} tock(t) \cdot TZ_0'(n - t, n - t) \lhd 0 < t < \theta(\mathrm{T}, n) \rhd \delta \\
T\hat{Z}_0 = {} & T(X[n = 0] + Y[n = 0]) \\
TZ_0'(ct_0 : \mathbb{T}, ct_1 : \mathbb{T}) = {} & ring \cdot T\hat{Z}_0'(ct_0, ct_1) \lhd ct_0 = 0 \vee ct_1 = 0 \rhd \delta + \\
& tick(\theta(\mathrm{T}, ct_0, \mathrm{T}, ct_1)) \cdot TZ_0'(ct_0 - \theta(\mathrm{T}, ct_0, \mathrm{T}, ct_1), ct_1 - \theta(\mathrm{T}, ct_0, \mathrm{T}, ct_1)) \lhd \theta(\mathrm{T}, ct_0, \mathrm{T}, ct_1) \neq \mathsf{c} \rhd \delta + \\
& \sum_{t \in \mathbb{T}} tock(t) \cdot TZ_0'(ct_0 - t, ct_1 - t) \lhd 0 < t < \theta(\mathrm{T}, ct_0, \mathrm{T}, ct_1) \rhd \delta \\
T\hat{Z}_0'(ct_0 : \mathbb{T}, ct_1 : \mathbb{T}) = {} & T(X[ct_0 = 0] + Y[ct_1 = 0]) \\
TZ_1 = {} & ring \cdot T\hat{Z}_1 \lhd n = 0 \rhd \delta + \\
& tick(\theta(\mathrm{T}, n)) \cdot TZ_1'(n - \theta(\mathrm{T}, n)) \lhd \theta(\mathrm{T}, n) \neq \mathsf{c} \rhd \delta + \\
& \sum_{t \in \mathbb{T}} tock(t) \cdot TZ_1'(n - t) \lhd 0 < t < \theta(\mathrm{T}, n) \rhd \delta \\
T\hat{Z}_1 = {} & T(X[n = 0] + Y[n = 0]) \\
TZ_1'(ct_0 : \mathbb{T}) = {} & ring \cdot T\hat{Z}_1'(ct_0) \lhd ct_0 = 0 \rhd \delta + \\
& tick(\theta(\mathrm{T}, ct_0)) \cdot TZ_1'(ct_0 - \theta(\mathrm{T}, ct_0)) \lhd ct_0 \neq \mathsf{c} \rhd \delta + \\
& \sum_{t \in \mathbb{T}} tock(t) \cdot TZ_1'(ct_0 - t) \lhd 0 < t < ct_0 \rhd \delta \\
T\hat{Z}_1'(ct_0 : \mathbb{T}) = {} & T(X[ct_0 = 0] + Y[ct_0 = 0])
\end{aligned}
$$

Here, $P[b]$, with $b : \mathbb{B}$, refers to the process P, with all the conditions of its alternatives extended with '$\wedge b$'. First of all, clearly $T\hat{Z}_0 = T\hat{Z}_1$. Furthermore, $TZ_0 = TZ_1, TZ_0' = TZ_1'$, and $T\hat{Z}_0' = T\hat{Z}_1'$, considering that we can observe, that the two delays are initially of equal length, and furthermore, at all times $ct_0 = ct_1$. □

Proposition 8 (TR1, TR2). *For $X = tick(m){\cdot}Y$ with $m > 0$, $\exists TX_i.TX \overset{tick(m)}{\longrightarrow} TX_i$ with $TX_i = TZ_1$, $Z_1 = tick(0){\cdot}Y$ and $\exists TX_j.TX \overset{tock(n)}{\longrightarrow} TX_j$ with $0 < n < m$ and $TX_j = TZ_2$, $Z_2 = tick(m-n){\cdot}Y$.*

Proof. Consider a process $X = tick(m){\cdot}Y$ with $m > 0$. This transforms to $TX = ring{\cdot}T\hat{X} \lhd m = 0 \rhd \delta + tick(\theta(\mathrm{T},m)){\cdot}TX'(m - \theta(\mathrm{T},m)) \lhd \theta(\mathrm{T},m) \neq \mathrm{c} \rhd \delta + \sum_{t\in\mathbb{T}} tock(t){\cdot}TX'(m-t) \lhd 0 < t < \theta(\mathrm{T},m) \rhd \delta$ with $TX'(ct_0 : \mathbb{T}) = ring{\cdot}T\hat{X}'(ct_0) \lhd ct_0 = 0 \rhd \delta + tick(\theta(\mathrm{T},ct_0)){\cdot}T\hat{X}'(ct_0 - \theta(\mathrm{T},ct_0)) \lhd \theta(\mathrm{T},ct_0) \neq \mathrm{c} \rhd \delta + \sum_{t\in\mathbb{T}} tock(t){\cdot}TX'(ct_0 - t) \lhd 0 < t < \theta(\mathrm{T},ct_0) \rhd \delta$ and $T\hat{X}' = TY$. Since $\theta(\mathrm{T},m) = m$, TX can fire $tick(m)$, leading to $TX'(m-m)$, which is clearly equal to TZ_1. Another possibility for TX is to fire $tock(t)$, with $0 < t < m$, leading to process $TX'(m-t)$, which is clearly equal to TZ_2. □

Proposition 9 (TR3). *Given a μCRL^{tick} process $X = a{\cdot}Y$, with $a \in \mathcal{A}_D$. Then $\exists TX_i.TX \overset{tock(m)}{\longrightarrow} TX_i$, with $0 < m$ and $TX_i = TX$.*

Proof. We have $TX = a{\cdot}TY + tick(\theta){\cdot}TX' \lhd \theta \neq \mathrm{c} \rhd \delta + \sum_{t\in\mathbb{T}} tock(t){\cdot}TX' \lhd 0 < t < \theta \rhd \delta$. Since $\theta = \mathrm{c}$, $tick \notin en_{\mathcal{A}}(TX)$, but $tock \in en_{\mathcal{A}}(TX)$. We have $(TX, tock(t), TX') \in en(TX)$, with $0 < t < \mathrm{c}$, c being the reasonable upper-limit to the size of time steps. Furthermore, $TX' = a{\cdot}TY + tick(\theta){\cdot}TX' \lhd \theta \neq \mathrm{c} \rhd \delta + \sum_{t\in\mathbb{T}} tock(t){\cdot}TX' \lhd 0 < t < \theta \rhd \delta$, so clearly $TX = TX'$. □

Proposition 10 (TR4). *Given a μCRL^{tick} process $X = a{\cdot}Y$, with $a \in \mathcal{A}_U$. Then $\exists TX_i.TX \overset{tock(m)}{\longrightarrow} TX_i$, with $0 < m$ and $TX_i = T\delta$.*

Proof. We have $TX = a{\cdot}TY + tick(\theta){\cdot}TX' \lhd \theta \neq \mathrm{c} \rhd \delta + \sum_{t\in\mathbb{T}} tock(t){\cdot}TX' \lhd 0 < t < \theta \rhd \delta$. Since $\theta = \mathrm{c}$, $tick \notin en_{\mathcal{A}}(TX)$, but $tock \in en_{\mathcal{A}}(TX)$. We have $(TX, tock(t), TX') \in en(TX)$, with $0 < t < \mathrm{c}$, c being the reasonable upper-limit to the size of time steps. Furthermore, $TX' = tick(\theta){\cdot}TX' \lhd \theta \neq \mathrm{c} \rhd \delta + \sum_{t\in\mathbb{T}} tock(t){\cdot}TX' \lhd 0 < t < \theta \rhd \delta$, which can only fire *tock* steps. Clearly TX' behaves as $T\delta$ with $T\delta'$. □

Proposition 11 (TR5). *Given a μCRL^{tick} process $X = tick(0){\cdot}TY$. Then $TX \overset{ring}{\longrightarrow} TY$.*

Proof. We have $TX = ring{\cdot}T\hat{X} \lhd 0 = 0 \rhd \delta + tick(\theta(\mathrm{T},0)){\cdot}TX'(0 - \theta(\mathrm{T},0)) \lhd \theta(\mathrm{T},0) \neq \mathrm{c} \rhd \delta + \sum_{t\in\mathbb{T}} tock(t){\cdot}TX'(0 - \theta(\mathrm{T},0)) \lhd 0 < t < \theta(\mathrm{T},0) \rhd \delta$. From this, it clearly follows that TX can fire *ring*. Furthermore, since *tick* is a clock action, and there is only one *tick* action in X, it follows that $T\hat{X} = TY$. □

Proposition 12 (TR6, TR7). *Given two μCRL^{tick} processes P and Q, such that $\exists TP_i.TP \overset{tick(m)}{\longrightarrow} TP_i$ and $\exists TQ_j.TQ \overset{tick(m)\vee tock(m)}{\longrightarrow} TQ_j$, then (1) $T(P+Q) \overset{tick(m)}{\longrightarrow} T(P_i+Q_j)$ and (2) $T(Q+P) \overset{tick(m)}{\longrightarrow} T(Q_j+P_i)$.*

Proof. We only prove (1) here. The proof for (2) is similar, due to the commutativity of $+$. First of all, let $R(d:\mathbb{D}) = P + Q$ be of the following form:

$$\begin{aligned} R(d:\mathbb{D}) = & \sum_{i\in I^P} \sum_{e_i^P \in \mathbb{D}_i^P} a_i^P(\boldsymbol{f_i^P}) \cdot X_i^P(\boldsymbol{g_i^P}) \lhd \boldsymbol{h_i^P} \rhd \delta + \\ & \sum_{i\in I^Q} \sum_{e_i^Q \in \mathbb{D}_i^Q} a_i^Q(\boldsymbol{f_i^Q}) \cdot X_i^Q(\boldsymbol{g_i^Q}) \lhd \boldsymbol{h_i^Q} \rhd \delta \end{aligned}$$

Now we have $TR(d:\mathbb{D})$ as follows:

$$\begin{aligned} & TR(d:\mathbb{D}) = \\ & \quad \sum_{i\in I^P\setminus I_C^P} \sum_{e_i^P \in \mathbb{D}_i^P} a_i^P(\boldsymbol{f_i^P}) \cdot TX_i^P(\boldsymbol{g_i^P}) \lhd \boldsymbol{h_i^P} \rhd \delta + \\ & \quad \sum_{i\in I^Q\setminus I_C^Q} \sum_{e_i^Q \in \mathbb{D}_i^Q} a_i^Q(\boldsymbol{f_i^Q}) \cdot TX_i^Q(\boldsymbol{g_i^Q}) \lhd \boldsymbol{h_i^Q} \rhd \delta + \\ & \quad ring \cdot T\hat{R}(d) \lhd F \vee \bigvee_{i\in I_C^P} (\boldsymbol{f_i^P} = 0 \wedge \boldsymbol{h_i^P}) \vee \bigvee_{i\in I_C^Q} (\boldsymbol{f_i^Q} = 0 \wedge \boldsymbol{h_i^Q}) \rhd \delta + \\ & \quad tick(\theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}})) \cdot TR'(d, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}} \to \boldsymbol{f}_{\boldsymbol{i_c^P}} - \theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}), \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \\ & \qquad\qquad \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}} \to \boldsymbol{f}_{\boldsymbol{i_c^Q}} - \theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}), \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) \lhd \theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) \neq \mathsf{c} \rhd \delta + \\ & \quad \sum_{t\in\mathbb{T}} tock(t) \cdot TR'(d, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}} \to \boldsymbol{f}_{\boldsymbol{i_c^P}} - t, \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}} \to \boldsymbol{f}_{\boldsymbol{i_c^Q}} - t, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) \lhd 0 < t < \theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) \rhd \delta \end{aligned}$$

Since $\exists TP_i.TP \overset{tick(m)}{\longrightarrow} TP_i$, by the form of TP, $\theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}) = m$ and $TP_i = TP'$. We can distinguish two cases:

1. $\exists TQ_j.TQ \overset{tick(m)}{\longrightarrow} TQ_j$. Then, by the form of TQ, $\theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) = m$ and $TQ_j = TQ'$. By the definition of θ, $\theta(\overrightarrow{b_0, n_0}, \overrightarrow{b_1, n_1}) = min\{\theta(\overrightarrow{b_0, n_0}), \theta(\overrightarrow{b_1, n_1})\}$, therefore $\theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) = m$. So $TR(d) \overset{tick(m)}{\longrightarrow} TR'(d, \ldots)$.

2. $\exists TQ_j.TQ \overset{tock(m)}{\longrightarrow} TQ_j$. Then, by the form of TQ, $\theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) > m$ and $TQ_j = TQ'$. By the definition of θ, $\theta(\overrightarrow{b_0, n_0}, \overrightarrow{b_1, n_1}) = min\{\theta(\overrightarrow{b_0, n_0}), \theta(\overrightarrow{b_1, n_1})\}$, therefore $\theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) = m$. So $TR(d) \overset{tick(m)}{\longrightarrow} TR'(d, \ldots)$.

Finally, we show that $TR'(d, \overrightarrow{ct_{i_c^P} : \mathbb{T}}, \overrightarrow{ct_{i_c^Q} : \mathbb{T}}) = T(P' + Q')$.

We have $TR'(d, \overrightarrow{ct_{i_c^P} : \mathbb{T}}, \overrightarrow{ct_{i_c^Q} : \mathbb{T}}) = T(P+Q)'$. It follows directly that $T(P+Q)' = T(P' + Q')$. A similar argument holds for $T\hat{R}$ and $T\hat{R}'$. □

Proposition 13 (TR8). *Given two μCRL^{tick} processes P and Q, such that $\exists TP_i . TP \xrightarrow{tock(m)} TP_i$ and $\exists TQ_j . TQ \xrightarrow{tock(m)} TQ_j$, then $T(P+Q) \xrightarrow{tock(m)} T(P_i + Q_j)$.*

Proof. First of all, let $R(d:\mathbb{D}) = P + Q$ and $TR(d:\mathbb{D})$ be of the forms as presented in Proposition 12.

Since $\exists TP_i . TP \xrightarrow{tock(m)} TP_i$, by the form of TP, $\theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}) > m$ and $TP_i = TP'$. Since $\exists TQ_j . TQ \xrightarrow{tock(m)} TQ_j$, by the form of TQ, $\theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) > m$ and $TQ_j = TQ'$. By the definition of θ, $\theta(\overrightarrow{b_0, n_0}, \overrightarrow{b_1, n_1}) = min\{\theta(\overrightarrow{b_0, n_0}), \theta(\overrightarrow{b_1, n_1})\}$, therefore $\theta(\overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^P}}, \boldsymbol{f}_{\boldsymbol{i_c^P}}}, \overrightarrow{\boldsymbol{h}_{\boldsymbol{i_c^Q}}, \boldsymbol{f}_{\boldsymbol{i_c^Q}}}) > m$. So $TR(d) \xrightarrow{tock(m)} TR'(d, \ldots)$.

Finally, we show that $TR'(d, \overrightarrow{ct_{i_c^P} : \mathbb{T}}, \overrightarrow{ct_{i_c^Q} : \mathbb{T}}) = T(P' + Q')$.
We have $TR'(d, \overrightarrow{ct_{i_c^P} : \mathbb{T}}, \overrightarrow{ct_{i_c^Q} : \mathbb{T}}) = T(P+Q)'$. It follows directly that $T(P+Q)' = T(P' + Q')$. A similar argument holds for $T\hat{R}$ and $T\hat{R}'$. □

Proposition 14 (TR9, TR10, TR11, TR12). *Given two μCRL^{tick} processes P and Q, such that $\exists TP_i . TP \xrightarrow{ring} TP_i$ and $\exists TQ_j . TQ \xrightarrow{ring} TQ_j$, then $T(P+Q) \xrightarrow{ring} T(P_i + Q_j)$.*

Proof. First of all, let $R(d:\mathbb{D}) = P + Q$ and $TR(d:\mathbb{D})$ be of the forms as presented in Proposition 12.

Since $\exists TP_i . TP \xrightarrow{ring} TP_i$, by the form of TP, $\exists k \in I_C^P . \boldsymbol{f_k^P} = 0 \wedge \boldsymbol{h_k^P}$ and $TP_k = T\hat{P}$. Since $\exists TQ_j . TQ \xrightarrow{ring} TQ_j$, by the form of TQ, $\exists l \in I_C^Q . \boldsymbol{f_l^Q} = 0 \wedge \boldsymbol{h_l^Q}$ and $TQ_l = T\hat{Q}$. By the form of $TR(d)$, it follows that $TR(d) \xrightarrow{ring} T\hat{R}(d)$.

Finally, we show that $T\hat{R}(d) = T(\hat{P} + \hat{Q})$. We do that by comparing $\hat{R}(d)$ with $\hat{P} + \hat{Q}$. For $\hat{R}(d)$, we have:

$$\begin{aligned}\hat{R}(d:\mathbb{D}) = & \sum_{i \in I_C^P} \sum_{j \in I^{X_i^P}} \sum_{e_j^{i,P} \in \mathbb{D}_j^{i,P}} a_j^{i,P}(f_j^{i,P}(\boldsymbol{g_i^P}, e_j^{i,P})) \cdot \\ & X_j^{i,P}(g_j^{i,P}(\boldsymbol{g_i^P}, e_j^{i,P})) \lhd h_j^{i,P}(\boldsymbol{g_i^P}, e_j^{i,P}) \wedge \boldsymbol{f_i^P} = 0 \wedge \boldsymbol{h_i^P} \rhd \delta + \\ & \sum_{i \in I_C^Q} \sum_{j \in I^{X_i^Q}} \sum_{e_j^{i,Q} \in \mathbb{D}_j^{i,Q}} a_j^{i,Q}(f_j^{i,Q}(\boldsymbol{g_i^Q}, e_j^{i,Q})) \cdot \\ & X_j^{i,Q}(g_j^{i,Q}(\boldsymbol{g_i^Q}, e_j^{i,Q})) \lhd h_j^{i,Q}(\boldsymbol{g_i^Q}, e_j^{i,Q}) \wedge \boldsymbol{f_i^Q} = 0 \wedge \boldsymbol{h_i^Q} \rhd \delta\end{aligned}$$

Clearly, this is equal to $\hat{P} + \hat{Q}$.

□

Proposition 15 (TR13, TR14, TR15, TR16). *Given two μCRL^{tick} processes P and Q, such that*
$\exists TP_i . TP \xrightarrow{ring} TP_i$ and $TQ \not\xrightarrow{ring}$, then (1) $T(P+Q) \xrightarrow{ring} TP_i$ and (2) $T(Q+P) \xrightarrow{ring} TP_i$.

Proof. We only prove (1) here. The proof for (2) is similar, due to the commutativity of +. First of all, let $R(d : \mathbb{D}) = P + Q$ and $TR(d : \mathbb{D})$ be of the forms as presented in Proposition 12.

Since $\exists TP_i.TP \xrightarrow{ring} TP_i$, by the form of TP, $\exists k \in I_C^P.\boldsymbol{f_k^P} = 0 \wedge \boldsymbol{h_k^P}$ and $TP_k = T\hat{P}$. Since $TQ \overset{ring}{\not\longrightarrow}$, by the form of TQ, $\neg\exists l \in I_C^Q.\boldsymbol{f_l^Q} = 0 \wedge \boldsymbol{h_l^Q}$. By the form of $TR(d)$, it follows that $TR(d) \xrightarrow{ring} T\hat{R}(d)$.

Finally, we show that $T\hat{R}(d) = T\hat{P}$. We do that by comparing $\hat{R}(d)$ with $T\hat{P}$. $\hat{R}(d)$ is of the form as presented in Proposition 14. Since we know that $\neg\exists l \in I_C^Q.\boldsymbol{f_l^Q} = 0 \wedge \boldsymbol{h_l^Q}$, effectively, $TR(d) = \hat{P}$. □

Proposition 16 (TR17, TR18, TR19, TR20, TR21, TR22). *Given a μCRL^{tick} process P, such that*
$\exists TP_i.TP \xrightarrow{ring} TP_i$, then, for any other μCRL^{tick} process Q, $TP|\!\!\top\!\!|TQ \xrightarrow{ring} TP_i|\!\!\top\!\!|TQ$,
$TQ|\!\!\top\!\!|TP \xrightarrow{ring} TQ|\!\!\top\!\!|TP_i$ and $TP{\cdot}TQ \xrightarrow{ring} TP_i{\cdot}TQ$.

Proof. Follows from the transition rules for $\|$ and $\cdot$ in μCRL, since in the transformed $\mathcal{I}_\top$, *ring* actions are not treated in a special way, i.e. are not encapsulated or renamed. □

Proposition 17 (TR23, TR24). *Given two μCRL^{tick} processes P and Q, such that $\exists TP_i.TP \xrightarrow{tick(m)} TP_i$ and $\exists TQ_j.TQ \xrightarrow{tick(m)\vee tock(m)} TQ_j$, then (1) $TP|\!\!\top\!\!|TQ \xrightarrow{tick(m)} TP_i|\!\!\top\!\!|TQ_j$ and (2) $TQ_j|\!\!\top\!\!|TP_i \xrightarrow{tick(m)} TQ_j|\!\!\top\!\!|TP_i$.*

Proof. We only prove (1) here. The proof for (2) is similar, due to the commutativity of $|\!\!\top\!\!|$. First of all, we have $TP|\!\!\top\!\!|TQ \triangleq \rho_{\{tick'\to tick, tock'\to tock\}}(\partial_{\{tick,tock\}}(TP \parallel TQ))$. We can distinguish two cases:

1. $\exists TQ_j.TQ \xrightarrow{tick(m)} TQ_j$. Since $(tick, tick, tick') \in \mathcal{C}$, $\partial_{\{tick,tock\}}(TP \parallel TQ) \xrightarrow{tick'(m)}$ $\partial_{\{tick,tock\}}(TP_i \parallel TQ_j)$, therefore $\rho_{\{tick'\to tick, tock'\to tock\}}(\partial_{\{tick,tock\}}(TP \parallel TQ)) \xrightarrow{tick(m)}$ $\rho_{\{tick'\to tick, tock'\to tock\}}(\partial_{\{tick,tock\}}(TP_i \parallel TQ_j))$.

2. $\exists TQ_j.TQ \xrightarrow{tock(m)} TQ_j$. Since $(tick, tock, tick') \in \mathcal{C}$, $\partial_{\{tick,tock\}}(TP \parallel TQ) \xrightarrow{tick'(m)}$ $\partial_{\{tick,tock\}}(TP_i \parallel TQ_j)$, therefore $\rho_{\{tick'\to tick, tock'\to tock\}}(\partial_{\{tick,tock\}}(TP \parallel TQ)) \xrightarrow{tick(m)}$ $\rho_{\{tick'\to tick, tock'\to tock\}}(\partial_{\{tick,tock\}}(TP_i \parallel TQ_j))$.

□

Proposition 18 (TR25). *Given two μCRL^{tick} processes P and Q, such that $\exists TP_i.TP \xrightarrow{tock(m)} TP_i$ and $\exists TQ_j.TQ \xrightarrow{tock(m)} TQ_j$, then $TP|\!\!\top\!\!|TQ \xrightarrow{tock(m)} TP_i|\!\!\top\!\!|TQ_j$.*

Proof. First of all, we have $TP\overset{\top}{|}|TQ \triangleq \rho_{\{tick'\to tick,tock'\to tock\}}(\partial_{\{tick,tock\}}(TP \parallel TQ))$. Since $(tock, tock, tock') \in \mathcal{C}$, $\partial_{\{tick,tock\}}(TP \parallel TQ) \xrightarrow{tock'(m)} \partial_{\{tick,tock\}}(TP_i \parallel TQ_j)$, therefore $\rho_{\{tick'\to tick,tock'\to tock\}}(\partial_{\{tick,tock\}}(TP \parallel TQ)) \xrightarrow{tock(m)} \rho_{\{tick'\to tick,tock'\to tock\}}(\partial_{\{tick,tock\}}(TP_i \parallel TQ_j))$. □

Proposition 19 (TR26). *Given a μCRL^{tick} process P, such that (1) $\exists TP_i.TP \xrightarrow{tick(m)} TP_i$, then for any μCRL^{tick} process Q, $TP\cdot TQ \xrightarrow{tick(m)} TP_i\cdot TQ$, and (2) $\exists TP_i.TP \xrightarrow{tock(m)} TP_i$, then for any μCRL^{tick} process Q, $TP\cdot TQ \xrightarrow{tock(m)} TP_i\cdot TQ$.*

Proof. Both cases follow directly from the form of TP. □

Appendix B

Cannibals and Missionaries Specifications

THIS APPENDIX PRESENTS all the technical details concerning the μCRL and the PROMELA specifications of the Cannibals and Missionaries problem, as described in Section 10.3. Besides the specifications themselves, we show the commands invoked at the command line to initiate the searches performed.

B.1 The Experiments with SPIN

First of all, Figure B.1 presents the PROMELA specification of the Cannibals and Missionaries problem we used. It incorporates C-code, in order to facilitate BnB search as described by Ruys (2003). The C-code near the end of the specification is invoked when a violation to the checked property is found (the property to check will be presented next). In a BnB search, the C-variable *min_cost* is used to keep track of the cumulated cost along a trace. Note that besides the check if the latest result is the best result found so far ('*now.cost* < *min.cost*'), the C-code contains an additional check, since we are only interested in violations of the property which represent successful termination. Whether a state represents successful termination or not is encoded by means of the flag *finished*, which is set to *true* in successful termination states.

Next, Figure B.2 displays the property files used to search for optimal answers in both the exhaustive and the BnB searches on the left and the right, respectively. The difference between the two files lies in the fact that for the BnB searches, the property contains C-code which accesses the variable *min_cost*; the actual result of this technique is that the property to check changes during a search.

Finally, in Figure B.3 the commands are shown which were used to run the exhaustive and the BnB searches, respectively, employing SPIN 4.2.7.

B.2 The Experiments with μCRL

In the μCRL specification of the Cannibals and Missionaries problem, besides the usual data types $\mathbb{N}$ and $\mathbb{B}$, we defined two special data types, to make the specification more intuitive. First of all, the data type *Person* distinguishes exactly two kinds of persons; a person is either a cannibal (C) or a missionary (M). Second of all, with the data type *Shore*, we differentiate between the *Left* shore and the *Right* shore. The μCRL specification $\mathcal{M} = (\mathcal{D}, \mathcal{F}, \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{I})$ for the Cannibals and Missionaries problem used in our experiments is as follows:

- $\mathcal{D} = \{\mathbb{N}, \mathbb{B}, \mathit{Person}, \mathit{Shore}\}$.

- $\mathcal{F}$ contains the usual operators for $\mathbb{N}$ and $\mathbb{B}$, plus equality, negation, and *if-then-else* for the data types *Person* and *Shore*.[1] Furthermore, there is the function *noviolations*, which checks for a given 'snapshot' of the problem, whether there is a violation concerning the cannibals-missionaries ratio on either shore or in the boat.

- $\mathcal{A} = \{\mathit{goright} : \mathbb{N} \times \mathbb{N}, \mathit{goleft} : \mathbb{N} \times \mathbb{N}, \mathit{getin} : \mathit{Person}, \mathit{getout} : \mathit{Person}, \mathit{finished}, \mathit{tick}\}$, where *tick* and *finished* are used to represent the progress of time and successful termination, respectively, and *goright*, *goleft*, *getin*, and *getout* are used to express the boat moving to the right and the left shore, and a person moving in and out of the boat, respectively. The parameters of *goright* and *goleft* respectively indicate the number of cannibals and missionaries occupying the boat.

- No communication is used, therefore $\mathcal{C} = \emptyset$.

- $\mathcal{P} = \{\mathit{Delay}, \mathit{CanMis}\}$, with *Delay* and *CanMis* as presented in Figure B.4. The process *Delay* is used to compactly describe delays in *CanMis*.

- $\mathcal{I} = \mathit{CanMis}(\mathit{misnumber}, \mathit{cannumber}, 0, 0, \mathit{Left}, 0, 0)$, with *misnumber* and *cannumber* set to the initial number of missionaries and cannibals in the problem, respectively, and the subsequent two 0 values indicating that there are no missionaries and cannibals currently in the boat. Furthermore, *Left* in the initial setting expresses that the boat starts at the left shore, and the final two arguments of *CanMis* are used for delay and search guiding purposes.

Figure B.5 displays the commands to invoke linearisation of the μCRL specification and exploration of the resulting state spaces, both using exhaustive (breadth-first) search and g-synchronised flexible detailed beam search. For the experiments, we used the μCRL toolset version 2.17.13.

[1] The *if-then-else* operator is written as $b \rightarrow\ x, y$, with $b : \mathbb{B}$ and $x, y : \mathit{Person}$ (or *Shore*), where $\mathrm{T} \rightarrow x, y \triangleq x$ and $\mathrm{F} \rightarrow x, y \triangleq y$.

```
#define CAPBOAT 2
#define PPNUMBER 3
#define MISNUMBER PPNUMBER
#define CANNUMBER PPNUMBER
// For BoatPos: true is left shore, false is right shore
int MisLeft = MISNUMBER;
int CanLeft = CANNUMBER;
int MisBoat, CanBoat, cost;
bool BoatPos =  true;
bool finished = false;
c_state "int min_cost" "Hidden" "200"

active proctype CANMIS()
{
new: if
:: MisBoat>0 && BoatPos && (MisBoat >= CanBoat || MisBoat == 0) && (MisLeft + MisBoat >= CanLeft
    + CanBoat || MisLeft + MisBoat == 0) && (MISNUMBER - MisLeft - MisBoat >= CANNUMBER - CanLeft -
  CanBoat || MISNUMBER - MisLeft - MisBoat == 0) -> d_step{MisLeft++ ; MisBoat--}; goto new
:: MisBoat>0 && !BoatPos && (MisBoat >= CanBoat || MisBoat == 0) && (MisLeft >= CanLeft || MisLeft
  == 0) && (MISNUMBER - MisLeft >= CANNUMBER - CanLeft || MISNUMBER - MisLeft == 0)
  -> d_step{MisBoat--}; goto new
:: CanBoat>0 && BoatPos && (MisBoat >= CanBoat || MisBoat == 0) && (MisLeft + MisBoat >= CanLeft +
  CanBoat || MisLeft + MisBoat == 0) && (MISNUMBER - MisLeft - MisBoat >= CANNUMBER - CanLeft -
  CanBoat || MISNUMBER - MisLeft - MisBoat == 0) -> d_step{CanLeft++ ; CanBoat--}; goto new
:: CanBoat>0 && !BoatPos && (MisBoat >= CanBoat || MisBoat == 0) && (MisLeft >= CanLeft || MisLeft
  == 0) && (MISNUMBER - MisLeft >= CANNUMBER - CanLeft || MISNUMBER - MisLeft == 0)
  -> d_step{CanBoat--}; goto new
:: MisLeft > 0 && MisBoat + CanBoat < CAPBOAT && BoatPos && (MisBoat >= CanBoat || MisBoat == 0)
  && (MisLeft + MisBoat >= CanLeft + CanBoat || MisLeft + MisBoat == 0) && (MISNUMBER - MisLeft -
  MisBoat >= CANNUMBER - CanLeft - CanBoat || MISNUMBER - MisLeft - MisBoat == 0)
  -> d_step{MisLeft-- ; MisBoat++}; goto new
:: MisLeft + MisBoat < MISNUMBER && MisBoat + CanBoat < CAPBOAT && !BoatPos && (MisBoat >= CanBoat
  || MisBoat == 0) && (MisLeft >= CanLeft || MisLeft == 0) && (MISNUMBER - MisLeft >= CANNUMBER -
  CanLeft || MISNUMBER - MisLeft == 0) -> d_step{MisBoat++}; goto new
:: CanLeft > 0 && MisBoat + CanBoat < CAPBOAT && BoatPos && (MisBoat >= CanBoat || MisBoat == 0)
  && (MisLeft + MisBoat >= CanLeft + CanBoat || MisLeft + MisBoat == 0) && (MISNUMBER - MisLeft -
  MisBoat >= CANNUMBER - CanLeft - CanBoat || MISNUMBER - MisLeft - MisBoat == 0)
  -> d_step{CanLeft-- ; CanBoat++}; goto new
:: CanLeft + CanBoat < CANNUMBER && MisBoat + CanBoat < CAPBOAT && !BoatPos && (MisBoat >= CanBoat
  || MisBoat == 0) && (MisLeft >= CanLeft || MisLeft == 0) && (MISNUMBER - MisLeft >= CANNUMBER -
  CanLeft || MISNUMBER - MisLeft == 0) -> d_step{CanBoat++}; goto new
:: BoatPos && MisBoat + CanBoat >= 1 && (MisBoat >= CanBoat || MisBoat == 0) && (MisLeft + MisBoat
  >= CanLeft + CanBoat || MisLeft + MisBoat == 0) && (MISNUMBER - MisLeft - MisBoat >= CANNUMBER -
  CanLeft - CanBoat || MISNUMBER - MisLeft - MisBoat == 0)
  -> d_step{cost = cost + MisBoat + CanBoat ; BoatPos = false}; goto new
:: !BoatPos && MisBoat + CanBoat >= 1 && (MisBoat >= CanBoat || MisBoat == 0) && (MisLeft >=
  CanLeft || MisLeft == 0) && (MISNUMBER - MisLeft >= CANNUMBER - CanLeft || MISNUMBER - MisLeft
  == 0) -> d_step{cost = cost + MisBoat + CanBoat ; BoatPos = true}; goto new
:: MisLeft == 0 && CanLeft == 0 && MisBoat == 0 && CanBoat == 0 -> finished = true; goto end

fi;

end: c_code {
                   if (now.cost < min_cost && now.finished) {
                         min_cost = now.cost;
                         printf(">min cost now: %d\n", min_cost);
                         putrail(); Nr_Trails--;
                   }
}}
```

Figure B.1: A PROMELA specification of the Cannibals and Missionaries problem

```
#define q (cost >= 200)
/*
 * Formula As Typed:  <>  q
 * The Never Claim Below Corresponds
 * To The Negated Formula !( <>  q)
 * (formalizing violations of the original)
 */
never {    /* !( <>  q) */
accept_init:
T0_init:
if
:: (! ((q))) -> goto T0_init
fi;
}
#ifdef NOTES
Use Load to open a file or a template.
#endif
#ifdef RESULT
#endif
```

```
#define q (c_expr {now.cost >= min_cost})
/*
 * Formula As Typed:  <>  q
 * The Never Claim Below Corresponds
 * To The Negated Formula !( <>  q)
 * (formalizing violations of the original)
 */
never {    /* !( <>  q) */
accept_init:
T0_init:
if
:: (! ((q))) -> goto T0_init
fi;
}
#ifdef NOTES
Use Load to open a file or a template.
#endif
#ifdef RESULT
#endif
```

Figure B.2: CM property files for SPIN exhaustive and BnB search

```
spin -a -N property.ltl canmis.pml && gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=900 -DNOREDUCE
  -DNOFAIR pan.c && time ./pan -v -m100000 -w20 -a -c0

spin -a -N bnb_property.ltl canmis.pml && gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=900
  -DNOREDUCE -DNOFAIR pan.c && time ./pan -v -w20 -a -m100000 -c0
```

Figure B.3: Commands to invoke CM SPIN exhaustive (top) and BnB (bottom) search

$$\mathit{Delay}(t:\mathbb{N}) = \mathit{tick}\cdot\mathit{Delay}(t-1) \lhd t>1 \rhd \delta + \mathit{tick} \lhd t=1 \rhd \delta$$

$$\begin{aligned}
&\mathit{CanMis}(\mathit{MisLeft}:\mathbb{N},\mathit{CanLeft}:\mathbb{N},\mathit{MisBoat}:\mathbb{N},\mathit{CanBoat}:\mathbb{N},\mathit{BoatPos}:\mathit{Shore},d:\mathbb{N},f:\mathbb{N}) = \\
&\quad \mathit{Delay}(d)\cdot\mathit{CanMis}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos},0,(\mathit{MisLeft}=\mathit{CanLeft}\rightarrow 0,1)) \lhd d=1 \rhd \delta + \\
&\quad \mathit{Delay}(d)\cdot\mathit{CanMis}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos},0,(\mathit{MisLeft}=\mathit{CanLeft}\rightarrow 0,1)) \lhd d>1 \wedge \\
&\qquad\quad \mathit{noviolations}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos}) \rhd \delta + \\
&\quad \mathit{getout}(M)\cdot\mathit{CanMis}((\mathit{BoatPos}=\mathit{Left}\rightarrow \mathit{MisLeft}+1,\mathit{MisLeft}),\mathit{CanLeft},\mathit{MisBoat}-1,\mathit{CanBoat},\mathit{BoatPos},0, \\
&\qquad\quad ((\mathit{BoatPos}=\mathit{Left}\rightarrow \mathit{MisLeft}+1,\mathit{MisLeft})=\mathit{CanLeft}\rightarrow 0,1)) \lhd \mathit{MisBoat}>0 \wedge \\
&\qquad\quad \mathit{noviolations}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos}) \wedge d=0 \rhd \delta + \\
&\quad \mathit{getout}(C)\cdot\mathit{CanMis}(\mathit{MisLeft},(\mathit{BoatPos}=\mathit{Left}\rightarrow \mathit{CanLeft}+1,\mathit{CanLeft}),\mathit{MisBoat},\mathit{CanBoat}-1,\mathit{BoatPos},0, \\
&\qquad\quad (\mathit{MisLeft}=(\mathit{BoatPos}=\mathit{Left}\rightarrow \mathit{CanLeft}+1,\mathit{CanLeft})\rightarrow 0,1)) \lhd \mathit{CanBoat}>0 \wedge \\
&\qquad\quad \mathit{noviolations}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos})) \wedge d=0 \rhd \delta + \\
&\quad \mathit{getin}(M)\cdot\mathit{CanMis}((\mathit{BoatPos}=\mathit{Left}\rightarrow \mathit{MisLeft}-1,\mathit{MisLeft}),\mathit{CanLeft},\mathit{MisBoat}+1,\mathit{CanBoat},\mathit{BoatPos},0, \\
&\qquad\quad ((\mathit{BoatPos}=\mathit{Left},\mathit{MisLeft}-1,\mathit{MisLeft})=\mathit{CanLeft}\rightarrow 0,1)) \lhd (\mathit{BoatPos}=\mathit{Left}\rightarrow \mathit{MisLeft}>0, \\
&\qquad\quad (\mathit{MisLeft}+\mathit{MisBoat})<\mathit{misnumber}) \wedge (\mathit{MisBoat}+\mathit{CanBoat})<\mathit{capboat} \wedge \\
&\qquad\quad \mathit{noviolations}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos})) \wedge d=0 \rhd \delta + \\
&\quad \mathit{getin}(C)\cdot\mathit{CanMis}(\mathit{MisLeft},(\mathit{BoatPos}=\mathit{Left}\rightarrow \mathit{CanLeft}-1,\mathit{CanLeft}),\mathit{MisBoat},\mathit{CanBoat}+1,\mathit{BoatPos},0, \\
&\qquad\quad (\mathit{MisLeft}=(\mathit{BoatPos}=\mathit{Left}\rightarrow \mathit{CanLeft}-1,\mathit{CanLeft}),0,1)) \lhd (\mathit{BoatPos}=\mathit{Left}\rightarrow \mathit{CanLeft}>0, \\
&\qquad\quad (\mathit{CanLeft}+\mathit{CanBoat})<\mathit{cannumber}) \wedge (\mathit{MisBoat}+\mathit{CanBoat})<\mathit{capboat} \wedge \\
&\qquad\quad \mathit{noviolations}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos}) \wedge d=0 \rhd \delta + \\
&\quad \mathit{goright}(\mathit{MisBoat},\mathit{CanBoat})\cdot\mathit{CanMis}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{Right},\mathit{MisBoat}+\mathit{CanBoat}, \\
&\qquad\quad (\mathit{MisLeft}=\mathit{CanLeft})\rightarrow 0,1)) \lhd \mathit{BoatPos}=\mathit{Left} \wedge (\mathit{MisBoat}+\mathit{CanBoat})\geq 1 \wedge \\
&\qquad\quad \mathit{noviolations}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos}) \wedge d=0 \rhd \delta + \\
&\quad \mathit{goleft}(\mathit{MisBoat},\mathit{CanBoat})\cdot\mathit{CanMis}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{Left},\mathit{MisBoat}+\mathit{CanBoat}, \\
&\qquad\quad (\mathit{MisLeft}=\mathit{CanLeft}\rightarrow 0,1)) \lhd \mathit{BoatPos}=\mathit{Right} \wedge \mathit{MisBoat}+\mathit{CanBoat}\geq 1 \wedge \\
&\qquad\quad \mathit{noviolations}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos}) \wedge d=0 \rhd \delta + \\
&\quad \mathit{finished}\cdot\delta\cdot\mathit{CanMis}(\mathit{MisLeft},\mathit{CanLeft},\mathit{MisBoat},\mathit{CanBoat},\mathit{BoatPos},d,(\mathit{MisLeft}=\mathit{CanLeft}\rightarrow 0,1)) \\
&\qquad\quad \lhd \mathit{MisLeft}=0 \wedge \mathit{CanLeft}=0 \wedge \mathit{MisBoat}=0 \wedge \mathit{CanBoat}=0 \wedge d=0 \rhd \delta
\end{aligned}$$

Figure B.4: The μCRL process *CanMis* specifying the Cannibals and Missionaries problem

```
mcrl -regular2 -binary canmis
time instantiators -tick -trace -action finished -local -stop canmis

mcrl -regular2 -binary canmis
time instantiators -tick -trace -action finished -stop -local -beam-width++ 3
  -detailed-expr canmis<<EOF
(MisLeft+CanLeft+(f*(6)))
EOF
```

Figure B.5: Commands to invoke CM μCRL exhaustive search (top) and g-synchronised flexible detailed beam search (bottom)

Appendix C

Branching Tail Bisimulation

THE NOTION OF *branching tail bisimulation* was defined by Baeten and Middelburg (2002), which is closely related to Van der Zwaag's definition of timed branching bisimulation. We show that in case of dense time, our counter-example (see Example 5) again shows that branching tail bisimilarity is not an equivalence relation.

In the absolute time setting of Baeten and Middelburg, states are of the form $<p,u>$ with p a process algebraic term and u a time stamp referring to the absolute time. They give an operational semantics to their process algebras such that if $<p,u> \xmapsto{v} <p,u{+}v>$ (where $\xmapsto{v}$ for $v > 0$ denotes a time step of v time units), then $<p,u> \xmapsto{w} <p,u{+}w>$ for $0 < w < v$; in our example this saturation with time steps will be mimicked. The reflexive transitive closure of $\xrightarrow{\tau}$ is denoted by $\twoheadrightarrow$. The relation $s \overset{u}{\mapstochar\twoheadrightarrow} s'$ is defined by: either $s \twoheadrightarrow \hat{s} \xmapsto{u} s'$, or $s \overset{v}{\mapstochar\twoheadrightarrow} \hat{s} \overset{w}{\mapstochar\twoheadrightarrow} s'$ with $v + w = u$.[1]

Branching tail bisimulation is defined as follows.[2]

Definition 39 (Branching tail bisimulation (Baeten and Middelburg, 2002)). *Assume a TLTS in the style of Baeten and Middelburg. A symmetric binary relation $B \subseteq \mathscr{S} \times \mathscr{S}$ is a* branching tail bisimulation *if sBt implies:*

1. *if $s \xrightarrow{\ell} s'$, then*

   i either $\ell = \tau$ and $t \twoheadrightarrow t'$ with sBt' and $s'Bt'$;

   ii or $t \twoheadrightarrow \hat{t} \xrightarrow{a} t'$ with $sB\hat{t}$ and $s'Bt'$;

2. *if $s \xrightarrow{\ell} <\surd, u>$, then $t \twoheadrightarrow t' \xrightarrow{\ell} <\surd, u>$ with sBt';*

3. *if $s \xmapsto{u} s'$, then*

   i either $t \twoheadrightarrow \hat{t} \xmapsto{v} \hat{t}' \overset{w}{\mapstochar\twoheadrightarrow} t'$ with $v + w = u$, $sB\hat{t}$ and $s'Bt'$;

[1] Baeten and Middelburg also have a deadlock predicate $\uparrow$, which we do not take into account here, as it does not play a role in our counter-example.

[2] Baeten and Middelburg define this notion in the setting with relative time, and remark that the adaptation of this definition to absolute time is straightforward. Here we present this straightforward adaptation.

ii or $t \twoheadrightarrow \hat{t} \overset{u}{\longmapsto} t'$ *with* $sB\hat{t}$ *and* $s'Bt'$.

Two states s and t are branching tail bisimilar, *written* $s \underline{\leftrightarrow}^{BM}_{tb} t$, *if there is a branching tail bisimulation B with sBt.*[3]

We proceed to transpose the TLTSs from Example 5 into the setting of Baeten and Middelburg. We now have the following transitions, for $i \geq 0$:

$<p,0> \xrightarrow{\tau} <p_0,0>$
$<p_i,0> \xrightarrow{\tau} <p_{i+1},0>$
$<p_{i+1},0> \xrightarrow{\tau} <p_i,0>$
$<p_i,u> \overset{v-u}{\longmapsto} <p_i,v>,\ 0 \leq u < v \leq \frac{1}{i+2}$
$<p_i,\frac{1}{i+2}> \xrightarrow{\tau} <p'_i,\frac{1}{i+2}>$
$<p'_i,u> \overset{v-u}{\longmapsto} <p'_i,v>,\ \frac{1}{i+2} \leq u < v \leq 1$
$<p'_i,\frac{1}{n}> \xrightarrow{a} <\surd,\frac{1}{n}>,\ n = 1,\ldots,i+1$

$<q,0> \xrightarrow{\tau} <q_0,0>$
$<q_i,0> \xrightarrow{\tau} <q_{i+1},0>$
$<q_{i+1},0> \xrightarrow{\tau} <q_i,0>$
$<q_i,u> \overset{v-u}{\longmapsto} <q_i,v>,\ 0 \leq u < v \leq 1$
$<q_i,\frac{1}{n}> \xrightarrow{a} <\surd,\frac{1}{n}>,\ n = 1,\ldots,i+1$

$<r,0> \xrightarrow{\tau} <r_0,0>$
$<r_i,0> \xrightarrow{\tau} <r_{i+1},0>$
$<r_{i+1},0> \xrightarrow{\tau} <r_i,0>$
$<r_i,u> \overset{v-u}{\longmapsto} <r_i,v>,\ \frac{1}{i+2} \leq u < v \leq 1$
$<r_i,\frac{1}{n}> \xrightarrow{a} <\surd,\frac{1}{n}>,\ n = 1,\ldots,i+1$
$<r_0,0> \xrightarrow{\tau} <r_\infty,0>$
$<r_\infty,0> \xrightarrow{\tau} <r_0,0>$
$<r_\infty,u> \overset{v-u}{\longmapsto} <r_\infty,v>,\ 0 \leq u < v \leq 1$
$<r_\infty,\frac{1}{n}> \xrightarrow{a} <\surd,\frac{1}{n}>,\ n \in \mathbb{N}$

$<p,0> \underline{\leftrightarrow}^{BM}_{tb} <q,0>$, since $<p,w>B<q,w>$ for $w \geq 0$, $<p_i,w>B<q_i,w>$ for $w \leq \frac{1}{i+2}$, and $<p'_i,w>B<q_i,w>$ for $w > 0$ (for $i \geq 0$) is a branching tail bisimulation.

Moreover, $<q,0> \underline{\leftrightarrow}^{BM}_{tb} <r,0>$, since $<q,w>B<r,w>$ for $w \geq 0$, $<q_i,w>B<r_i,w>$ for $w \geq 0$, $<q_i,0>B<r_j,0>$, and $<q_i,w>B<r_\infty,w>$ for $w = 0 \vee w > \frac{1}{i+2}$ (for $i,j \geq 0$) is a branching tail bisimulation.

However, $<p,0> \not\underline{\leftrightarrow}^{BM}_{tb} <r,0>$, since p cannot simulate r. This is due to the fact that none of the p_i can simulate r_∞. Namely, r_∞ can idle until time 1. p_i can only simulate this by executing a τ at time $\frac{1}{i+2}$, but the resulting process $<p'_i,\frac{1}{i+2}>$ is not timed branching bisimilar to $<r_\infty,\frac{1}{i+2}>$, since only the latter can execute action a at time $\frac{1}{i+2}$.

[3] The superscript *BM* refers to Baeten and Middelburg, to distinguish it from the notion of timed branching bisimulation as defined in Definition 31.

Bibliography

H. Aljazzar, D. Bošnački, S. Edelkamp, A. Fehnker, V. Schuppan, and A.J. Wijs. Algorithmic Foundations of Directed Model Checking. Unpublished manuscript, 2007.

B. Badban, W.J. Fokkink, J.F. Groote, J. Pang, and J.C. van de Pol. Verifying a Sliding Window Protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.

J.C.M. Baeten. Embedding untimed into timed process algebra: the case of explicit termination. *Mathematical Structures in Computer Science*, 13(4):589–618, 2003.

J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing*. EATCS Monograph. Springer, 2002.

J.C.M. Baeten and M.A. Reniers. Termination in Timed Process Algebra. Technical Report CSR 00-13, Technical University of Eindhoven, 2000.

J.C.M. Baeten, C.A. Middelburg, and M.A. Reniers. A New Equivalence for Processes with Timing: With an Application to Protocol Verification. Technical Report CSR 02-10, Technical University of Eindhoven, 2002.

H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. Elsevier, 1984.

T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.

D.A. van Beek, A. van der Ham, and J.E. Rooda. Modelling and Control of Process Industry Batch Production Systems. In *Proceedings of IFAC 2002*, 2002.

D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and Semantics of Timed Chi. CS-Report 05-09, Technical University of Eindhoven, 2005.

D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and Consistent Equation Semantics of Hybrid Chi. *Journal of Logic and Algebraic Programming*, 68(1-2):129–210, 2006.

G. Behrmann, T. Hune, and F. Vaandrager. Distributing Timed Model Checking - How the Search Order Matters. In *Proceedings of CAV 2000*, volume 1855 of *LNCS*, pages 216–231. Springer, 2000.

G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersson, and J.M.T. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In *Proceedings of TACAS 2001*, volume 2031 of *LNCS*, pages 174–188. Springer, 2001a.

G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersson, J.M.T. Romijn, and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In *Proceedings of HSCC 2001*, volume 2034 of *LNCS*, pages 147–161. Springer, 2001b.

G. Behrmann, A. David, and K.G. Larsen. A Tutorial on UPPAAL. In *Proceedings of SFM-RT 2004*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.

G. Behrmann, K.G. Larsen, and J.I. Rasmussen. Optimal Scheduling Using Priced Timed Automata. *SIGMETRICS Performance Evaluation Review*, 32(4):34–40, 2005.

J.A. Bergstra and J.W. Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

J.A. Bergstra and J.W. Klop. The Algebra of Recursively Defined Processes and the Algebra of Regular Processes. In *Proceedings of ICALP 1984*, volume 172 of *LNCS*, pages 82–95. Springer, 1984a.

J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984b.

M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202(1-2):1–54, 1998.

M. Bernardo, W.R. Cleaveland, S.T. Sims, and W.J. Stewart. TwoTowers: A Tool Integrating Functional and Performance Analysis of Concurrent Systems. In *Proceedings of FORTE/PSTV 1998*, pages 457–467. Kluwer Academic Publishers, 1998.

M. Bezem and J.F. Groote. Invariants in Process Algebra with Data. In *Proceedings of CONCUR 1994*, volume 836 of *LNCS*, pages 401–416. Springer, 1994.

R. Bisiani. Beam Search. In *Encyclopedia of Artificial Intelligence*, pages 1467–1568. Wiley, 1992.

S.C.C. Blom and S. Orzan. Distributed State Space Minimization. *International Journal on Software Tools for Technology Transfer*, 7(3):280–291, 2005.

S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A Toolset for Analysing Algebraic Specifications. In *Proceedings of CAV 2001*, volume 2102 of *LNCS*, pages 250–254. Springer, 2001.

S.C.C. Blom, N. Ioustinova, and N. Sidorova. Timed verification with μCRL. In *Proceedings of PSI 2003*, volume 2890 of *LNCS*, pages 178–192. Springer, 2003.

S.C.C. Blom, J.R. Calamé, B. Lisser, S. Orzan, J. Pang, J.C. van de Pol, M. Torabi Dashti, and A.J. Wijs. Distributed Analysis with μCRL: A Compendium of Case Studies. In *Proceedings of TACAS 2007*, volume 4424 of *LNCS*, pages 683–689. Springer, 2007.

G. Bolch, S. Greiner, H. de Meer, and K. Shridharbhai Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications, 2nd Edition*. Wiley, 2006.

T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14(1):25–59, 1987.

E.M. Bortnik, N. Trčka, A.J. Wijs, S.P. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkink, and J.E. Rooda. Analyzing a χ Model of a Turntable System using SPIN, CADP and UPPAAL. *Journal of Logic and Algebraic Programming*, 65 (2):51–104, 2005a.

E.M. Bortnik, D.A. van Beek, and J.M. van de Mortel-Fronczak. Verification of Timed χ Models Using UPPAAL. In *Proceedings of ICINCO 2005*. INSTICC Press, 2005b.

V. Bos and J.J.T. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer Integrated Manufacturing*, 17(3):185–198, 2001.

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. PhD thesis, Technical University of Eindhoven, 2002.

E. Brinksma and A. Mader. Verification and Optimization of a PLC Control Schedule. In *Proceedings of SPIN 2000*, volume 1885 of *LNCS*, pages 73–92. Springer, 2000.

E. Brinksma, H. Hermanns, and J.P. Katoen, editors. *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *LNCS*, 2001. Springer.

P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49(6):107–127, 1994.

P.P. Chakrabarti, S. Ghose, A. Acharya, and S.C. de Sarkas. Heuristic Search in Restricted Memory. *Artificial Intelligence*, 41(2):197–222, 1989.

S. Christensen, L.M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *Proceedings of TACAS 2001*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001. ISBN 3-540-41865-2.

G. Ciardo, J. Gluckman, and D. Nicol. Distributed State Space Generation of Discrete-State Stochastic Models. *INFORMS Journal on Computing*, 10(1):82–93, 1998.

A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of CAV 2002*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

E.M. Clarke, S. Jha, and W. Marrero. Partial Order Reductions for Security Protocol Verification. In *Proceedings of TACAS 2000*, volume 1785 of *LNCS*, pages 503–518. Springer, 2000.

R. De Nicola and F. Vaandrager. Three Logics for Branching Bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.

R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A^*. *Journal of the ACM*, 32(3):505–536, 1985.

F. Della Croce and V. T'kindt. A recovering beam search algorithm for the one-machine dynamic total completion time scheduling problem. *Journal of the Operational Research Society*, 53(11):1275–1280, 2002.

E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

D. Dill. The MURϕ Verification System. In *Proceedings of CAV 1996*, volume 1102 of *LNCS*, pages 390–393. Springer, 1996.

H.E. Dudeney. *Amusements in Mathematics*, chapter 9, pages 112–114. Dover Publications, Inc., 1958.

S. Edelkamp. Taming Numbers and Duration in the Model Checking Integrated Planning System. *Journal of Artificial Intelligence Research*, 20:195–238, 2003.

S. Edelkamp and S. Jabbar. Real-Time Model Checking on Secondary Storage. In *Proceedings of MoChArt 2006*, volume 4428 of *LNAI*, pages 68–84. Springer, 2007.

S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI Symposium on Model-Based Validation of Intelligence*, pages 84–92. AAAI, 2001a.

S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of SPIN 2001*, volume 2057 of *LNCS*, pages 57–79. Springer, 2001b.

S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2):247–267, 2004.

E.A. Emerson and C.-L. Lei. Modalities for Model Checking: Branching Time Logic Strikes Back. *Science of Computer Programming*, 8(3):275–306, 1987.

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking*. PhD thesis, Radboud University of Nijmegen, 2002.

A. Fehnker. Scheduling a Steel Plant with Timed Automata. In *Proceedings of RTCSA 1999*, pages 280–286. IEEE Computer Society Press, 1999.

A. Felner, S. Kraus, and R.E. Korf. KBFS: K-Best-First Search. *Annals of Mathematics and Artificial Intelligence*, 39(1-2):19–39, 2003.

W.J. Fokkink. Rooted Branching Bisimulation as a Congruence. *Journal of Computer and System Sciences*, 60(1):13–37, 2000a.

W.J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2000b.

W.J. Fokkink and J. Pang. Formal verification of timed systems using cones and foci. In *Proceedings of ARTS 2004*, volume 139 of *ENTCS*, pages 105–122. Elsevier, 2005.

W.J. Fokkink, N.Y. Ioustinova, E. Kesseler, J.C. van de Pol, Y.S. Usenko, and Y.A. Yushtein. Refinement and verification applied to an in-flight data acquisition unit. In *Proceedings of CONCUR 2002*, volume 2421 of *LNCS*, pages 1–23. Springer, 2002.

W.J. Fokkink, J. Pang, and A.J. Wijs. Is Timed Branching Bisimilarity an Equivalence Indeed? In *Proceedings of FORMATS 2005*, volume 3829 of *LNCS*, pages 258–272. Springer, 2005.

W.J. Fokkink, M. Torabi Dashti, and A.J. Wijs. Partial Order Reduction for Branching Security Protocols. In *Proceedings of WITS 2007*, pages 178–193, 2007.

M.S. Fox. *Constraint-directed search: A case study of job-shop scheduling*. PhD thesis, Carnegie-Mellon University, 1983.

H. Garavel and H. Hermanns. On Combining Functional Verification and Performance Evaluation Using CADP. In *Proceedings of FME 2002*, volume 2391 of *LNCS*, pages 410–429. Springer, 2002.

H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In *Proceedings of SPIN 2001*, volume 2057 of *LNCS*, pages 217–234. Springer, 2001.

H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. In *European Association for Software Science and Technology (EASST) Newsletter*, volume 4, pages 13–24, 2002.

H. Garavel, R. Mateescu, D. Bergamini, A. Curic, N. Descoubes, C. Joubert, I. Smarandache-Sturm, and G. Stragier. DISTRIBUTOR and BCG_MERGE: Tools for Distributed Explicit State Space Generation. In *Proceedings of TACAS 2006*, volume 3920 of *LNCS*, pages 445–449. Springer, 2006.

M.L. Ginsberg and W.D. Harvey. Iterative broadening. *Artificial Intelligence*, 55(2): 367–383, 1992.

R.J. van Glabbeek. The linear time – branching time spectrum II: The semantics of sequential systems with silent moves. In *Proceedings of CONCUR 1993*, volume 715 of *LNCS*, pages 66–81. Springer, 1993.

R.J. van Glabbeek. What is branching time and why to use it? *The Concurrency Column, Bulletin of the EATCS*, 53:190–198, 1994.

R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. In *Proceedings of Information Processing 1989*, pages 613–618, 1989.

R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996a.

R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996b.

P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Proceedings of TACAS 2002*, volume 2280 of *LNCS*, pages 266–280. Springer, 2002.

P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *Proceedings of CAV 1991*, volume 575 of *LNCS*, pages 410–429. Springer, 1991.

A. Groce and W. Visser. Heuristics for model checking Java programs. *International Journal on Software Tools for Technology Transfer*, 6(4):260–276, 2004.

J.F. Groote. The Syntax and Semantics of Timed μCRL. Technical Report SEN-R9709, CWI, 1997.

J.F. Groote and A. Ponse. The Syntax and Semantics of μCRL. In *Proceedings of ACP 1994*, Workshops in Computing Series, pages 26–62. Springer, 1995.

J.F. Groote and M.A. Reniers. Algebraic Process Verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier, 2001.

J.F. Groote and J.G. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49(1,2):31–60, 2001.

J.F. Groote, F. Monin, and J.C. van de Pol. Checking Verifications of Protocols and Distributed Systems by Computer. In *Proceedings of CONCUR 1998*, volume 1466 of *LNCS*, pages 629–655. Springer, 1998.

J.F. Groote, M.A. Reniers, J.J. van Wamel, and M.B. van der Zwaag. Completeness of timed μCRL. *Fundamenta Informaticae*, 50(3,4):361–402, 2002.

J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1-2):21–56, 2003.

M. Hammer and M. Weber. “to Store or Not to Store” Reloaded: Reclaiming Memory on Demand. In *Proceedings of FMICS 2006*, volume 4346 of *LNCS*, pages 51–66. Springer, 2006.

P.E. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems, Science and Cybernetics*, 2:100–107, 1968.

P. Haslum and H. Geffner. Admissible Heuristics for Optimal Planning. In *Proceedings of ICAPS 2000*, pages 140–149. AAAI Press, 2000.

H. Hermanns and J.-P. Katoen. Performance Evaluation := (Process Algebra + Model Checking) × Markov Chains. In *Proceedings of CONCUR 2001*, volume 2154 of *LNCS*, pages 59–81. Springer, 2001.

P.M.C. Hesen. Design of the Clinical Chemical Analyzer. Technical report, Stan Ackermans Institute, 2000.

C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21 (8):666–677, 1978.

J. Hoffmann and H. Geffner. Branching Matters: Alternative Branching in Graphplan. In *Proceedings of ICAPS'03*, pages 22–31. AAAI Press, 2003.

A.T. Hofkamp and H.W.A.M. van Rooy. *Embedded Systems Laboratory Exercises Manual*. Technical University of Eindhoven, 2003.

G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

G.J. Holzmann. *The* SPIN *Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods for System Design*, 13(3):289–307, 1998.

J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computations*. Addison-Wesley, 2001.

M. Huth and M. Kwiatkowska. Quantitative Analysis and Model Checking. In *Proceedings of LICS 1997*, pages 111–127. IEEE Computer Society Press, 1997.

N. Ioustinova. *Abstractions and Static Analysis for Verifying Reactive Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

S. Jabbar and S. Edelkamp. Parallel External Directed Model Checking With Linear I/O. In *Proceedings of VMCAI 2006*, volume 3855 of *LNCS*, pages 237–251. Springer, 2006.

H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL. In *Proceedings of SPIN 1996*, volume 32 of *Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

A.S. Klusener. Abstraction in real time process algebra. In *Proceedings of Real-Time: Theory in Practice REX Workshop*, volume 600 of *LNCS*, pages 325–352. Springer, 1991.

A.S. Klusener. The silent step in time. In *Proceedings of CONCUR 1992*, volume 630 of *LNCS*, pages 421–435. Springer, 1992.

A.S. Klusener. *Models and Axioms for a Fragment of Real Time Process Algebra*. PhD thesis, Technical University of Eindhoven, 1993.

R.E. Korf. Uniform-cost search. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 1461–1462. Wiley, 1992.

R.E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.

R.E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27 (3):333–354, 1983.

V. Kumar. Branch-and-bound search. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 1468–1472. Wiley, 1992.

S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In *Proceedings of SPIN 2006*, volume 3925 of *LNCS*, pages 35–52. Springer, 2006.

K. Larsen and J. Rasmussen. Optimal Conditional Reachability for Multi-Priced Timed Automata. In *Proceedings of FOSSACS 2005*, volume 3441 of *LNCS*, pages 234–249. Springer, 2005.

K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

E. Lawler, J.K. Lenstra, A. Rinnooy Kan, and D. Shmoys, editors. *The Traveling Salesman Problem*. Wiley, 1985.

F. Lerda and R. Sista. Distributed-Memory model checking with SPIN. In *Proceedings of SPIN 1999*, volume 1680 of *LNCS*, pages 22–39. Springer, 1999.

R. Lim. Cannibals and missionaries. In *Proceedings of APL 1992*, pages 135–142. ACM Press, 1992.

J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley-Teubner, 1996.

B.T. Lowerre. *The HARPY speech recognition system*. PhD thesis, Carnegie-Mellon University, 1976.

LTSVIEW. 3D Interactive Visualisation of a State Space. `http://www.mcrl2.org`, 2007. Last visited on 10 May 2007.

S.P. Luttik. *Choice Quantification in Process Algebra*. PhD thesis, University of Amsterdam, 2002.

R. Mateescu. Modélisation et analyse de systèmes asynchrones avec CADP. In *Systèmes temps réel 1 - techniques de description et de vérification*, pages 151–180. Lavoisier, 2006.

R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

D.S. Nau, V. Kumar, and L.N. Kanal. General Branch and Bound, and Its Relation to A^* and AO^*. *Artificial Intelligence*, 23(1):29–58, 1984.

X. Nicollin and J. Sifakis. An Overview and Synthesis on Timed Process Algebras. In *Proceedings of CAV 1991*, volume 575 of *LNCS*, pages 376–398. Springer, 1991.

P. Niebert, S. Tripakis, and S. Yovine. Minimum-time reachability for timed automata. In *Proceedings of MED 2000*. IEEE Computer Society Press, 2000.

S. Oechsner and O. Rose. Scheduling Cluster Tools Using Filtered Beam Search and Recipe Comparison. In *Proceedings of 2005 Winter Simulation Conference*, pages 2203–2210. IEEE Computer Society Press, 2005.

S. Orzan and J.C. van de Pol. Detecting strongly connected components in large distributed state spaces. Technical Report SEN-E0501, CWI, 2005.

S. Owre, J.M. Rushby, and N. Shankar. PVS: a Prototype Verification System. In *Proceedings of CADE 1992*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

J. Pang, W. J. Fokkink, R. F. Hofman, and R. Veldema. Model Checking a Cache Coherence Protocol for a Java DSM Implementation. *Journal of Logic and Algebraic Programming*, 71(1):1–43, 2007.

J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

E.P.D. Pednault. Formulating Multiagent, Dynamic-world Problems in the Classical Framework. In *Reasoning about Actions and Plans*, pages 47–82. Morgan Kaufmann, 1986.

D. Peled, V. Pratt, and G. Holzmann, editors. *Partial Order Methods in Verification*, volume 29 of *Series in Discrete Mathematics and Theoretical Computer Science*, 1996. American Mathematical Society.

M. Pinedo. *Scheduling: Theory, algorithms, and systems*. Prentice-Hall, 1995.

A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.

I. Pohl. Heuristic Search Viewed as Path Finding in a Graph. *Artificial Intelligence*, 1 (3):193–204, 1970.

G. Polya. *How to solve it*. Princeton University Press, 2nd edition, 1945.

D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998.

P. Regnier and B. Fade. Détermination du parallélisme maximal et optimisation temporelle dans les plans d'actions linéaires. *Revue d'intelligence artificielle*, 5(2):67–88, 1991.

M.A. Reniers and M. van Weerdenburg. Action Abstraction in Timed Process Algebra: The Case for an Untimed Silent Step. In *Proceedings of FSEN 2007*, 2007.

S. Rubin. *The ARGOS image understanding system*. PhD thesis, Carnegie-Mellon University, 1978.

S. Russell and P. Norvig. *Artificial intelligence: A modern approach*. Prentice-Hall, 1995.

T.C. Ruys. Optimal scheduling using Branch-and-Bound with SPIN 4.0. In *Proceedings of SPIN 2003*, volume 2648 of *LNCS*, pages 1–17. Springer, 2003.

T.C. Ruys and E. Brinksma. Experience with Literate Programming in the Modelling and Validation of Systems. In *Proceedings of TACAS 1998*, volume 1384 of *LNCS*, pages 393–408. Springer, 1998.

I. Sabuncuoglu and M. Bayiz. Job Shop Scheduling with Beam Search. *European Journal of Operational Research*, 118(2):390–412, 1999.

P. Si Ow and E.T. Morton. Filtered Beam Search in Scheduling. *International Journal of Production Research*, 26(1):35–62, 1988.

P. Si Ow and E.T. Morton. The single machine early/tardy problem. *Management Science*, 35(2):177–191, 1989.

P. Si Ow and S.F. Smith. Viewing scheduling as an opportunistic problem-solving process. *Annals of Operations Research*, 12(1-4):85–108, 1988.

W.P.C. Spronk. Throughput Analysis of a Clinical Chemical Analyzer. Master's thesis, Technical University of Eindhoven, 1999.

A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 2nd edition, 1989.

R.J.M. Theunissen. Process Algebraic Linearization of Hybrid χ. SE Report 420479, Technical University of Eindhoven, 2006.

Toolset of χ. Tools developed for the χ language. `http://se.wtb.tue.nl/sewiki/chi`, 2007. Last visited on 10 May 2007.

M. Torabi Dashti and A.J. Wijs. Pruning State Spaces with Extended Beam Search. In *Proceedings of ATVA 2007*, LNCS. Springer, 2007. To appear. Extended version as CWI Technical Report SEN-R0610.

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. PhD thesis, Technical University of Eindhoven, 2007.

N. Trčka. Verifying χ Models of Industrial Systems with SPIN. In *Proceedings of ICFEM 2006*, volume 4260 of *LNCS*, pages 132–148. Springer, 2006.

I. Ulidowski and S. Yuen. Extending Process Languages with Time. In *Proceedings of AMAST 1997*, volume 1349 of *LNCS*, pages 524–538. Springer, 1997.

Y.S. Usenko. *Linearization in μCRL*. PhD thesis, Technical University of Eindhoven, 2002a.

Y.S. Usenko. State space generation for the HAVi leader election protocol. *Science of Computer Programming*, 43(1):1–33, 2002b.

J.M.S. Valente and R.A.F.S. Alves. Beam Search Algorithms for the early/tardy scheduling problem with release dates. Working Paper 143, Faculdade de Economia do Porto, 2004.

J.M.S. Valente and R.A.F.S. Alves. Beam search algorithms for the single machine total weighted tardiness scheduling problem with sequence-dependent setups. Working Paper 186, Faculdade de Economia do Porto, 2005a.

J.M.S. Valente and R.A.F.S. Alves. Improved heuristics for the early/tardy scheduling problem with no idle time. *Computers and Operations Research*, 32(3):557–569, 2005b.

J.M.S. Valente and R.A.F.S. Alves. Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time. *Computers & Industrial Engineering*, 48(2):363–375, 2005c.

M.M. Veloso, M. A. Pérez, and J.G. Carbonell. Nonlinear Planning with Parallel Resource Allocation. In *Proceedings Innovative Approaches to Planning, Scheduling and Control*, pages 207–212, 1991.

J. Vervoort. Model of a Chemical Analyzer. WPA Report 420216, Technical University of Eindhoven, 1999.

L.J.P. Vieillot. Taeniopygia guttata. In *Nouveau Dictionnaire d'Histoire Naturelle Applique Aux Arts*, volume 12, page 233. Abel Lanoe, 1817.

B.W. Wah. *MIDA**, An *IDA** Search With Dynamic Control. Technical Report UILU-ENG-91-2216 CRHC-91-9, Center for Reliable and High-Performance Computing Coordinated Science Lab, University of Illinois, 1991.

S. Weber. *Design of Real-Time Supervisory Control Systems*. PhD thesis, Technical University of Eindhoven, 2003.

A.J. Wijs. Achieving Discrete Relative Timing with Untimed Process Algebra. In *Proceedings of ICECCS 2007*, pages 35–44. IEEE Computer Society Press, 2007.

A.J. Wijs and W.J. Fokkink. From χ_{t} to μCRL: Combining Performance and Functional Analysis. In *Proceedings of ICECCS 2005*, pages 184–193. IEEE Computer Society Press, 2005.

A.J. Wijs and B. Lisser. Distributed Extended Beam Search for Quantitative Model Checking. In *Proceedings of MoChArt 2006*, volume 4428 of *LNAI*, pages 165–182. Springer, 2007.

A.J. Wijs, J.C. van de Pol, and E. Bortnik. Solving Scheduling Problems by Untimed Model Checking - The Clinical Chemical Analyser Case Study. In *Proceedings of FMICS 2005*, pages 54–61. ACM Press, 2005. Extended version as CWI Technical Report SEN-R0608.

A.G. Wouters. Manual for the μCRL tool set. Technical Report SEN-R0130, CWI, 2001.

R.A. Zann. *The Zebra Finch - A Synthesis of Field and Laboratory Studies*. Oxford University Press Inc., 1996.

W. Zhang. *State-Space Search - Algorithms, Complexity, Extensions, and Applications*. Springer, 1999.

R. Zhou and E.A. Hansen. Beam-Stack Search: Integrating Backtracking with Beam Search. In *Proceedings of ICAPS 2005*, pages 90–98. AAAI, 2005.

M.B. van der Zwaag. The cones and foci proof technique for timed transition systems. *Information Processing Letters*, 80(1):33–40, 2001.

M.B. van der Zwaag. *Models and Logics for Process Algebra*. PhD thesis, University of Amsterdam, 2002.

Index of Symbols

$en_{\mathcal{A}}(P)$	Set of enabled actions of process term P	18
$en_{\mathcal{M}}(P)$	Set of transitions of process term P	18
I^X	Finite index set of LPE X	20
I_H	Finite index set of actions in an LPE to prioritise	45
I'	Finite index set of successful terminations in an extended LPE	48
$f_{X_i}(d,e_i)$	Parameter of action a_i in LPE X	20
$g_{X_i}(d,e_i)$	New state after executing action a_i in LPE X	20
$h_{X_i}(d,e_i)$	Condition of action a_i in LPE X	20
$\boldsymbol{f_i^X}$	Short for $f_{X_i}(d,e_i)$	105
$\boldsymbol{g_i^X}$	Short for $g_{X_i}(d,e_i)$	105
$\boldsymbol{h_i^X}$	Short for $h_{X_i}(d,e_i)$	105
$\mathcal{M}_{\top}$	μCRL^{tick} specification	103
$\mathcal{D}_{\top}$	Set of data domains in a μCRL^{tick} specification	103
$\mathcal{F}_{\top}$	Set of functions over data domains used in a μCRL^{tick} specification	103
$\mathcal{A}_{\top}$	Set of actions used in a μCRL^{tick} specification	103
$\mathcal{C}_{\top}$	Set of communication rules in a μCRL^{tick} specification	103
$\mathcal{P}_{\top}$	Set of recursive equations (processes) in a μCRL^{tick} specification	103
$\mathcal{I}_{\top}$	The initialisation line of a μCRL^{tick} specification	103
tick	Action representing progress of time	41
tick(t)	Delay of t time units in a μCRL^{tick} process	104
tick$'$	Action used for the synchronisation of *tick* (and *tock*) actions	41
tock	Action representing delayability	43
tock$'$	Action used for the synchronisation of *tick* and *tock* actions	43
$\mathcal{A}_U$	Set of urgent or undelayable actions	101
$\mathcal{A}_D$	Set of delayable actions	101
$\mathcal{A}_C$	Set of clock actions (i.e. $tick, tock, ring$)	101
I_N^X	Finite index set of ‘normal’ actions in LPE X	54
I_C^X	Finite set of ‘clock’ actions in LPE X	54
I_{C1}^X	Finite index set of *tick* actions in LPE X	56
I_{C2}^X	Finite index set of *tock* actions in LPE X	56
I_U	Finite index set of urgent or undelayable actions in an LPE	105
I_D	Finite index set of delayable actions in an LPE	105
$\overrightarrow{x_{i_c}}$	Vector of x_i’s of an LPE for which $i \in I_C$	105
$\overrightarrow{f_{i_c}(d,e_{i_c})}$	Vector of $f_i(d,e_i)$’s of an LPE for which $i \in I_C$	105
$\overrightarrow{g_{i_c}(d,e_{i_c})}$	Vector of $g_i(d,e_i)$’s of an LPE for which $i \in I_C$	105
$\overrightarrow{h_{i_c}(d,e_{i_c})}$	Vector of $h_i(d,e_i)$’s of an LPE for which $i \in I_C$	105
$P \mid\{tick\}\mid Q$	Parallel composition with synchronisation of *tick* actions	41
$P \mid^{\top}\mid Q$	Parallel composition with synchronisation of *tick* and *tock* actions	44
$P \mid^{\top}_{S}\mid Q$	As $P \mid^{\top}\mid Q$, plus supporting a set of shared variables S	60

Index of Symbols

$\mathscr{I}$	Set of initial states in a state space	22
$\mathscr{G}$	Set of goal states	23
$\mathscr{B}$	Set of deadlock or unsuccessful termination states	23
$\surd$	Termination state	15
$s\downarrow$	Successful termination predicate	15
ℓ	Action label in $\mathscr{A}$, action formula	22
$s \xrightarrow{\ell} s'$	Transition from s to s' with action label ℓ	22
$s \to^{*} s'$	State s' is reachable from state s	23
$s \to^{*}_{T} s'$	State s' is reachable from state s through the set of transitions T	46
$s \Rightarrow s'$	Reaching state s' from s through zero or more τ-steps	244
$en_{\mathscr{M}}(s)$	Set of enabled transitions in state s	23
$nxt_{\mathscr{M}}(s,T)$	Set of successor states of s through transitions in T	23

$\mathscr{L}_i$	Level i in a state space search	23
$\mathscr{H}$	Search horizon of a best-first search	121
$\mathscr{W}$	Set of states to be expanded in the next cost-region	167
f	Guiding function for a best-first search	121
β	Selection width of a best-first search, beam width of a beam search	121
α	Widening factor of a priority beam search	181
l	Stabilisation level of a priority beam search	181
$g(s)$	Cumulated cost of state s	123
$g_T(s)$	Cumulated cost of state s relative to scope T	124
$G(s)$	Guiding value of s in G-synchronised beam search	185
$S_{\downarrow,<U}$	Set of termination states for which the cumulated cost $< U$	165
$S_{\geq U}$	Set of states for which the cumulated cost $\geq U$	165
$h(s)$	Estimated remaining cost of a state s	134
$prio(\ell)$	Priority in $\mathbb{Z}$ of action ℓ	142
$prio_{min}(T)$	Lowest priority of actions in the set of transitions T	179
$getprio_{min}(T)$	A transition labelled with an action ℓ for which $prio(\ell) = prio_{min}(T)$	181
$getf_{max}(S)$	One of the states in S with the highest f-value	182
$\langle s, s.g\rangle$	A state s together with its cumulated cost	125
$\text{đ}(S,S')$	Minimal weighted distance between sets of states S and S'	123
U	Upper-bound to the guiding function f	128
D	Depth upper-bound in bounded depth-first search	166
β'	Selection width for the next iteration in an iterative search	132
U'	Cost upper-bound for the next iteration in an iterative search	132
f_i	Guiding function for phase i in a multi-phase best-first search	140
β_i	Selection width for phase i in a multi-phase best-first search	140
U_i	Cost upper-bound for phase i in a multi-phase best-first search	140
$\mathscr{H}_i$	Intermediate search horizon i in multi-phase best-first search	140
(ξ_A, ν_A)	Guiding signature of a best-first search A	190

$\xi_A(S)$	State space level created by search A given search horizon S	190
$\nu_A^n(S)$	Search horizon after n rounds in A starting with search horizon S	190
$\mathit{ample}(s)$	Selection of outgoing transitions of s to be expanded by POR	195

Ta	Set of tasks in a scheduling problem	152
t_i	Task in a scheduling problem	152
C_i	Constraint in a scheduling problem	152
$d(t_i)$	Execution time of task t_i	152
c_0	Initial progress in a scheduling problem	153
c_{end}	Final progress in a scheduling problem when the goal is reached	153
$s.\mathit{cost}$	Cumulated cost of state s stored in state variable *cost*	155
$w_i(d, e_i)$	Execution time of action a_i	157
$\mathit{finished}$	Action indicating successful termination	157
ω_s	A sequential schedule	158
ω_c	A parallel schedule	158

$\mathtt{CID}s$	Set of client `ID`s	25
$\#(s)$	Owner of state s	27
$\mathtt{ID}$	Unique client number in distributed state space generation	27
$\mathscr{L}_i^j$	The states in level $\mathscr{L}_i$ to be expanded by client j	27
$\mathscr{W}^{\mathtt{ID}}$	States to be expanded in the next cost-region by client `ID`	200
$\hat{\mathscr{W}}^{\mathtt{ID}}$	States to be expanded in the next cost-region found by client `ID`	200
$\mathscr{E}^{\mathtt{ID}}$	Set of successful termination states found by client `ID`	200
$[\sigma_u^j]$	Equivalence class of states for which $f(s) = u$, created by client j	204
$p_f(S)$	Distribution of states in S over equivalence classes according to f	204
$sel_\gamma^f(\mathscr{L}_i^j)$	Selection of γ states from $\mathscr{L}_i^j$ using f	204
$\gamma_{i,j}$	Initial selection width in round i of client j	204
E_i^j	Evaluation info in round i of client j	205
E_i	Evaluation info in round i for the manager	205
$p_e(S)$	Distribution of evaluation info over equivalence classes	205
$T_j(E_i)$	Number of states of client j represented in E_i	205
$T(E_i)$	Total number of states represented in E_i	205
$evsel_\beta(E_i)$	Selection of info representing up to β states from E_i	205

ϕ	Temporal logic state formula	28
$\neg\phi$	Negation	28
$\phi_1 \vee \phi_2$	Disjunction	28
$\phi_1 \wedge \phi_2$	Conjunction	28
$\langle\ell\rangle\phi, \Diamond\phi$	Possibility, eventuality	28
$[\ell]\phi$	Necessity	28
$\mu Y.\phi$	Minimal fixed point	28

Summary

The thesis consists of three parts:

Part I We provide an explanation of the discrete event subset of the hybrid modelling language χ_t. Following that, we present an approach to model time in μCRL. Such an approach allows to model timed systems using an untimed modelling language, thereby enabling the reuse of existing model checking tools. Next, we extend this approach, in order to subsequently provide a general scheme to translate χ_t specifications into μCRL specifications. This translation scheme can be seen as a bridge between the areas of performance analysis and system verification, as χ_t is mainly targeted to the former, while μCRL is mostly used for the latter. Next, the applicability of the translation scheme is demonstrated by translating and verifying a number of χ_t specifications, most notably the specification of a turntable system. Finally, the insights gained by designing the translation scheme lead to an extension of μCRL with a notion of discrete relative timing, called $\mu\text{CRL}^{\textit{tick}}$. It is shown that $\mu\text{CRL}^{\textit{tick}}$ specifications can be translated to μCRL specifications, thereby (again) allowing the use of the μCRL toolset for the verification of timed systems. The $\mu\text{CRL}^{\textit{tick}}$ approach builds on earlier proposals to model time with an untimed process algebra by emphasising ease of use for the modeller, meaning that the modeller does not need to be concerned about the correctness of the time mechanism in a specification, and incorporating time jumps of arbitrary size.

Part II This part starts by providing a uniform presentation of the most prominent state space search algorithms in the field of Directed Model Checking. Many of these algorithms stem from the field of Artificial Intelligence, often incorporating additional information about the problem at hand, such that the search can be directed to interesting areas of the state space. In the presentation, connections between the searches considered are accentuated, highlighting a framework in which many searches can be placed. The overview is concluded by considering action-based guiding, and proposing a further generalisation of best-first search, in which the search may consist of a number of sequential phases, giving rise to the compositionality of state space searches. Finally, a glossary of Directed Model Checking terms is proposed.

After that, the focus is narrowed to the modelling and solving of scheduling problems by means of existing model checkers and their input languages. The techniques available in the model checkers SPIN (with PROMELA) and UPPAAL CORA (with priced

timed automata) for scheduling are discussed, and at times extended to improve efficiency or quality of the solution. Besides that, new techniques are proposed, which have been implemented in the μCRL toolset.

The beam search algorithm is the subject of the next chapter. Traditionally, beam search is applied on highly structured search trees. By extending the basic algorithm in a number of ways, it can be efficiently applied on arbitrary state spaces. Both state-based (detailed) beam searches and action-based (priority) beam searches are developed and discussed. This leads to a spectrum of beam searches. By applying a proposed mechanism to compare search algorithms, we establish that several prominent searches can be seen as specific instances of beam search in this spectrum. Finally, it is shown how these beam search variants, and another, exhaustive, search useful for scheduling, can be adapted to work in a distributed setting, in which a number of computers work together in a cluster to generate, or search, a state space.

The part is concluded by presenting some case studies in which most of the proposed search algorithms have been applied in practice. The most notable case study is a Clinical Chemical Analyser. In all cases the efficiency and effectiveness of the techniques in the μCRL toolset are analysed, while in one case, some of the techniques in the μCRL toolset are compared with techniques available in the model checker SPIN.

Part III Timed behaviour of systems can be compared by means of timed versions of bisimilarity relations. We look at the properties of timed branching bisimilarity, and point out that the existing definition by Van der Zwaag is not transitive in an absolute, continuous time setting. Therefore, the definition needs to be extended in order to ensure that it is an equivalence in an absolute, continuous time setting. A stronger notion is proposed (stronger in the sense that it relates fewer processes), and it is proven that this timed branching bisimilarity is indeed an equivalence, even if the time domain is continuous. Furthermore, we show that in case of a discrete time domain, the notion by Van der Zwaag and our stronger notion coincide. As Appendix C shows, the presented counter-example for transitivity also applies to the notion of timed branching bisimilarity by Baeten & Middelburg in case of a continuous time domain. So that notion does not constitute an equivalence relation as well.

After that, a rooted version of the extended timed branching bisimilarity is defined, and proven to be a congruence over a process algebra with parallelism, successful termination, and deadlock. In a number of ways, the proof differs from the usual congruence proof for untimed branching bisimilarity. For example, due to the presence of successful termination, there is a large number of cases. In fact, the congruence proof for the parallel composition operator is restricted to a setting without successful termination, since the number of cases in a proof considering successful termination is just too large. Furthermore, it is demonstrated that the standard approach for untimed branching bisimilarity, i.e. take the smallest congruence closure and prove that this yields a branching bisimulation, falls short in a timed setting.

Samenvatting

Wat nu? Het analyseren en optimaliseren van systeemgedrag over tijd

Het proefschrift bestaat uit drie gedeelten:

Deel I We geven een uitleg van het discrete deel van de hybride modelleertaal χ_t. Daarna presenteren we een aanpak om tijd te modelleren in μCRL. Een dergelijke aanpak maakt het mogelijk om getimede systemen te modelleren met een ongetimede modelleertaal, waardoor de bestaande model checking tools kunnen worden hergebruikt. Vervolgens breiden we de aanpak uit, om daaropvolgend een algemeen schema te leveren om χ_t specificaties te vertalen naar μCRL specificaties. Dit vertalingsschema kan worden beschouwd als een brug tussen de gebieden van performance analyse en systeemverificatie, aangezien χ_t hoofdzakelijk gebruikt wordt voor het eerstgenoemde gebied en μCRL over het algemeen voor het laatstgenoemde. Daarna wordt de toepasbaarheid van het vertalingsschema gedemonstreerd door een aantal χ_t specificaties te vertalen en te verifiëren, waaronder met name een draaitafelsysteem. Tenslotte leiden de inzichten verkregen door het ontwerpen van het vertalingsschema tot een uitbreiding van μCRL met een notie van discrete, relatieve tijd, genaamd μCRL^{tick}. Er wordt aangetoond dat μCRL^{tick} specificaties kunnen worden vertaald naar μCRL specificaties, waardoor het (opnieuw) mogelijk is om de μCRL toolset te hergebruiken voor de verificatie van getimede systemen. De μCRL^{tick} aanpak bouwt voort op eerdere voorstellen om tijd te modelleren met een ongetimede procesalgebra door gebruiksgemak voor de modelleerder te benadrukken, wat wil zeggen dat de modelleerder zich niet hoeft te bekommeren om de correctheid van het tijdmechanisme in een specificatie, en door tijdstappen van variabele grootte te verwezenlijken.

Deel II Dit deel begint met het bieden van een uniforme presentatie van de meest prominente algoritmen voor het doorzoeken van toestandsruimten in het gebied van Directed Model Checking. Veel van deze algoritmen komen voort uit het gebied van de Kunstmatige Intelligentie, en maken vaak gebruik van extra informatie over het probleem in kwestie, zodat de zoektocht kan worden geleid naar interessante gedeelten van de toestandsruimte. In de presentatie worden de overeenkomsten tussen de

zoekmethoden benadrukt, waardoor een kader wordt geschetst waarbinnen veel zoekmethoden kunnen worden geplaatst. Het overzicht wordt afgerond met het overwegen van het sturen van een zoektocht op basis van acties, en het voorstellen van een verdere generalisatie van best-first search, waarbij de zoekmethode kan bestaan uit een aantal sequentiële fasen, wat een principe van compositionaliteit van zoekmethoden voor toestandsruimten introduceert. Tenslotte wordt er een vakwoordenlijst voorgesteld voor de termen in Directed Model Checking.

Vervolgens wordt de focus verlegd naar het modelleren en oplossen van schedulingproblemen met behulp van bestaande model checkers en hun invoertalen. De technieken die beschikbaar zijn in de model checkers SPIN (met PROMELA) en UPPAAL CORA (met geprijsde getimede automaten) voor scheduling worden besproken, en op sommige punten uitgebreid om de efficiëntie van de technieken of de kwaliteit van de resultaten te verbeteren. Daarnaast worden er nieuwe technieken voorgesteld, die geïmplementeerd zijn in de μCRL toolset.

Het beam search algoritme is het onderwerp van het volgende hoofdstuk. Traditioneel wordt beam search toegepast op zeer gestructureerde zoekbomen. Door het basisalgoritme op meerdere manieren uit te breiden kan het efficient worden toegepast op willekeurige toestandsruimten. Zowel toestandsgerelateerde (detailed) beam searches als actiegerelateerde (priority) beam searches worden ontwikkeld en besproken. Dit leidt tot een spectrum van beam searches. Door een voorgesteld mechanisme om zoekalgoritmen te vergelijken toe te passen, kunnen we vaststellen dat meerdere prominente zoekmethoden kunnen worden beschouwd als specifieke instanties van beam search in dit spectrum. Tenslotte wordt er aangetoond hoe deze beam search varianten, en een andere, uitputtende, zoekmethode voor scheduling, kunnen worden aangepast om te werken in een gedistribueerde omgeving, waarin een aantal computers samenwerken in een cluster om een toestandsruimte te genereren, of te doorzoeken.

Het deel wordt afgesloten door een aantal casestudies te presenteren waarbij de meeste voorgestelde zoekalgoritmen in de praktijk zijn toegepast. De meest opmerkelijke casestudie is een analysator van klinische chemicaliën. In alle gevallen is de efficiëntie en effectiviteit van de technieken in de μCRL toolset geanalyseerd, terwijl in één geval de technieken in de μCRL toolset zijn vergeleken met technieken beschikbaar in de model checker SPIN.

Deel III Getimed gedrag van systemen kan worden vergeleken door middel van getimede versies van bisimilariteitsrelaties. We kijken naar de eigenschappen van getimede vertakkende bisimilariteit, en wijzen erop dat de bestaande definitie van Van der Zwaag niet transitief is in een absolute, continue tijdomgeving. Dientengevolge moet de definitie worden uitgebreid om te verzekeren dat het een equivalentie is in een absolute, continue tijdomgeving. Een sterkere notie wordt voorgesteld (sterker in de zin dat het minder processen met elkaar relateert), en er wordt bewezen dat deze getimede vertakkende bisimilariteit inderdaad een equivalentie is, zelfs als het tijdsdomein continu is. Voorts laten we zien dat in het geval van een discreet tijdsdomein de notie van Van der Zwaag en onze sterkere notie overeenkomen. Zoals Appendix C laat zien is het gepresenteerde tegenvoorbeeld voor transitiviteit ook van toepassing op de notie van getimede vertakkende bisimilariteit van Baeten & Middelburg in het geval van een continu tijdsdomein. Dus die notie vestigt ook niet een equivalentierelatie.

Daarna wordt er een gewortelde versie van de uitgebreide getimede vertakkende bisimilariteit gedefinieerd, en wordt er bewezen dat het een congruentie is over een procesalgebra met parallellisme, succesvolle terminatie, en deadlock. Op een aantal punten wijkt het bewijs af van het gebruikelijke congruentiebewijs voor ongetimede vertakkende bisimilariteit. Vanwege de aanwezigheid van succesvolle terminatie is er bijvoorbeeld een groot aantal gevallen. Feitelijk is het congruentiebewijs voor de parallelle-compositieoperator beperkt tot een situatie zonder succesvolle terminatie, aangezien het aantal gevallen in een bewijs met succesvolle terminatie gewoonweg te groot is. Bovendien wordt er gedemonstreerd dat de standaardaanpak voor ongetimede vertakkende bisimilariteit, namelijk het nemen van de kleinste congruentiesluiting en bewijzen dat deze een vertakkende bisimulatie levert, tekort schiet in een getimede omgeving.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition* *and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to Do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13