Software ENgineering

From timed χ_t to μCRL: Combining performance and functional analysis

A.J. Wijs, W.J. Fokkink

REPORT SEN-R0420 DECEMBER 2004

ISSN 1386-369X

From timed Chi to mCRL: Combining performance and functional analysis

ABSTRACT
In this paper we first give short overviews of the modelling languages timed Chi and mCRL. Then we present a general translation scheme to translate timed Chi specifications to mCRL specifications. As timed Chi targets performance analysis and mCRL targets functional analysis of systems, this translation scheme provides a way to perform both kinds of analysis on a given timed Chi system model. Finally, we give an example of a timed Chi system and show how the translation works on a concrete case study.

2000 Mathematics Subject Classification: 68N30; 68Q85
1998 ACM Computing Classification System: I.6.5
Keywords and Phrases: modelling; process algebra; languages; formal methods; translation
Note: This work was carried out under project SEN2 - TIPSy

From χ_t to μCRL: Combining Performance and Functional Analysis

Anton Wijs[1] and Wan Fokkink[1,2]

[1] CWI,Department of Software Engineering,
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands,
{wijs,wan}@cwi.nl
[2] Vrije Universiteit Amsterdam, Department of Computer Science,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands,
wanf@cs.vu.nl

Abstract. In this paper we first give short overviews of the modelling languages timed χ (χ_t) and μCRL. Then we present a general translation scheme to translate χ_t specifications to μCRL specifications. As χ_t targets performance analysis and μCRL targets functional analysis of systems, this translation scheme provides a way to perform both kinds of analysis on a given χ_t system model. Finally, we give an example of a χ_t system and show how the translation works on a concrete case study.

1 Introduction

Performance analysis is traditionally based on techniques such as simulation, Markov chains and queueing networks. By contrast, main 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.

Hermanns and Katoen [25] verified performance properties of a LOTOS specification of a telephone system; LOTOS [11] is a process algebraic language with abstract data types, which is originally meant for functional analysis. Garavel and Hermanns [21] 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 [16], 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 paper we propose 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 [3] 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 [26], 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}. Currently there are no tools available for using the language χ (they are being developed), but predecessors of the language and their simulators have been successfully applied to a large number of industrial cases, such as an integrated circuit manufacturing plant, a brewery and process industry plants [2].

μCRL [19] is a modelling language for the specification of discrete-event systems. It is based on the process algebra ACP [5], extended with abstract data types [29]. It targets functional analysis of distributed systems and communication protocols, by means of simulation, model checking and theorem proving. The verification environment of μCRL together with the model checker CADP, which can serve as a back-end to μCRL, have been used to analyse for instance an in-flight data acquisition unit [20] and a distributed system for lifting trucks [24]. Moreover, a homegrown theorem prover has been developed for μCRL [18].

Recently, in [12] the χ_{t} specification of a turntable [13] was translated to three different specification formalisms: UPPAAL, SPIN and μCRL. While translating to μCRL, it was concluded, that χ_{t} and μCRL are quite closely related, and the development started of a general translation scheme from χ_{t} to μCRL. A general translation 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.

In this paper we present a general translation from χ_{t} specifications to *linear process equations* [8], which are basically μCRL specifications without parallelism and communication. The LPE format is important for the μCRL toolset, because it is used for the internal representation of processes. The translation is inspired by the translation of the turntable in [12]. Note that we have to limit ourselves to translating χ_{t} instead of the complete hybrid χ, because μCRL cannot cope with continuous events. The verification of a turntable system in [12] illustrates how our translation scheme can be used to combine performance and functional analysis.

Our work is closest in spirit to TwoTowers [7], which is a tool that combines performance and functional analysis. It has a single input language, based on the stochastic process algebra EMPA [6]. Performance analysis is based on simulation and reward Markov chains, while functional analysis is performed by the symbolic model checker nuSMV [15].

Our paper is set up as follows. The next two sections provide a short introduction to χ_{t} and μCRL: The basics of the languages are listed and a brief explanation is given. Section 4 provides a way to translate χ_{t} processes to linear process equations. Finally section 5 provides an example of translating a χ_{t} model to a μCRL model.

As future work, we plan to first manually apply the translation to larger examples, to gain further confidence in its applicability and improve it where necessary. Then we will work out a correctness proof of the transformation, to guarantee that it preserves a large class of interesting properties. We will also implement the translation and use it to automatically translate χ_{t} specifications of real-life systems to μCRL. Moreover we will apply the verification environments of χ_{t} (simulation) and μCRL (model checking and theorem proving) to such examples, thus obtaining the desired combination of performance and functional analysis.

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 models 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}. For a complete reference of χ, see [3].

Data types. 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 and real number. The language provides a mechanism to build sets, lists, array tuples, record tuples, dictionaries, functions, and distributions (for stochastic models). 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 weak time determinism principle, or sometimes called the time factorization property (time doesn't 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 implicitly delay.

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 and they cannot be split into smaller processes. They are:

1. The multi-assignment process ($x_n := e_n$). It assigns the values (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. The values of $e_1, ..., e_n$ must be defined and of the right type. For $n = 0$, $h\,!!\,e_n$ denotes $h\,!!$ 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.
5. The receive process ($h\,??\,x_n$). It receives values via the channel h and assigns them to the variables $x_1, ..., x_n$ which must be of the right type. For $n = 0$, $h\,??\,x_n$ denotes $h\,??$ 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.
7. The delay process ($\varDelta t$). 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 ($\varDelta_t$). The process term $\varDelta_t(p)$ 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 for the behaviour of p.
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.
4. The sequential composition operator (;). A process $p;\ q$ behaves as p followed by the process q.
5. The alternative composition operator ($[\!]$). A process $p \ [\!]\ q$ represents a non-deterministic choice between p and q if they can proceed.
6. The repetition operator ($*$). A process $*p$ behaves as p infinitely many times.
7. The guarded repetition operator ($*$:). A process $*b : p$ can be interpreted as "while b do p".
8. The parallel composition operator ($\|$). A process $p \parallel q$ executes p and q concurrently in an interleaved fashion, i.e. the actions of p and q are executed

in arbitrary order. 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.

9. The scope operator ($[\![\ |\ ]\!]$). 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 recursively defined as the empty state or as dcl, s' where s' is a state and dcl is a variable declaration ($x : type[= val]$) or a channel declaration ($h : ?type$ for receiving, $h : !type$ for sending, and $h : -type$ for both).
10. The encapsulation operator ($\partial_{\mathcal{A}}$). A process $\partial_{\mathcal{A}}(p)$ disables all actions of p that occur in the set $\mathcal{A}$. Typically this operator is used to enforce that send and receive actions synchronise.
11. The urgent communication operator ($\upsilon_{\mathcal{H}}$). Send and receive actions in a process $\upsilon_{\mathcal{H}}(p)$ via channels from set $\mathcal{H}$ can only delay when no communication with a corresponding receive or send action on the same channel is possible.

Process definitions. The language χ_{t} provides the possibility to define processes. We do not give a syntax definition here but rather an example:

$$P(c : ?\,\mathsf{nat}\ ,\ b : \mathsf{bool}\ ) = [\![\ x : \mathsf{nat}\ \mid\ b \to c\,?\,x\ ]\!]$$

The process P has two arguments, a channel c that can transport natural numbers and a boolean variable b. It has only one local variable, x. The process can now be instantiated at the initialisation line as for example $P(m, y > 7)$ (using channel m and natural number y, both declared at the initialisation line).

3 The language μCRL

Basically, μCRL is based on the process algebra ACP [5, 1, 17], extended with equational abstract data types [29, 4]. In order to intertwine processes with data, actions and recursion variables can be parametrised with data types. Moreover, a conditional construct (if-then-else) can be used to have data elements influence the course of a process, and *alternative quantification* (also called *choice quantification*) is added to sum over possibly infinite data domains.

The language comes with a toolset [9] that can build a state space from the specification and store it in the `.aut` format, one of the input formats of the model checker CADP [16]. Next to that, in order to strive for precision in proofs, an important research area is to use theorem provers such as PVS [32] 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 [23, 33].

We will give a short overview of the language necessary for understanding this paper. For a complete reference, see [19].

Data types. Initially there are no data types known in a μCRL specification. Therefore each specification should start by defining the necessary data types and the functions that work on them. In fact, it is mandatory to define the boolean type in each specification, since the conditional construct works with boolean expressions. One can virtually define any data type. In an example at the end of this paper we use a data type for the natural numbers, to name one.

Actions. In μCRL one can declare actions in the `act` section of a specification. These actions may have zero, one or several data parameters. When parameters are used the data types of these parameters need to be given. One can also allow processes `P` and `Q` to communicate in the parallel process `P` $\|$ `Q`. To do this it is possible to define which actions are able to synchronise with each other in the `comm` section of a specification.

Finally the process deadlock (δ), which cannot terminate successfully, and the internal action τ are predefined.

Operators. There are eight operators in μCRL. We present each one of them with an informal semantics.

1. The alternative composition operator (`+`). A process `p+q` proceeds (non-deterministically) as `p` or `q` (if they can proceed).
2. The sum operator ($\sum_{\mathtt{d:D}}$ `X(d)`), with `X(d)` a mapping from the data type `D` to processes, behaves as $\mathtt{X(d_1)} + \mathtt{X(d_2)} + \ldots$, i.e., as the possibly infinite choice between `X(d)` for any data term `d` taken from `D`. This operator is used to describe a process that is reading some input over a data type [30].
3. The sequential composition operator (`.`). A process `p.q` proceeds as `p` followed by `q`.
4. The process expression $\mathtt{p} \lhd \mathtt{b} \rhd \mathtt{q}$ where `p` and `q` are processes, and `b` is a data term of data type `Bool`, 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, and operates as a then_if_else construct.
5. The parallel operator ($\|$). A process `p` $\|$ `q` executes `p` and `q` concurrently in an interleaved fashion, i.e. the actions of `p` and `q` are executed in arbitrary order. For all actions `a` and `b` which can communicate with each other: If one process can execute `a` and the other one can execute `b` then `p` and `q` can communicate (`p` $\|$ `q` executes the communication action).
6. The encapsulation operator ($\partial_{\mathtt{H}}$). A process $\partial_{\mathtt{H}}$`(p)` disables all actions of `p` that occur in the set $\mathtt{H} \subseteq \mathtt{Act}$. Typically this operator is used to enforce that certain actions synchronise.
7. The renaming operator ($\rho_{\mathtt{f}}$), with `f`: $\mathtt{Act} \rightarrow \mathtt{Act}$, is suited for reusing a given specification with different action names. The subscript `f` signifies that the action `a` must be renamed to `f(a)`. The process $\rho_{\mathtt{f}}$`(p)` behaves as `p` with its action names renamed according to `f`.
8. The abstraction operator ($\tau_{\mathtt{I}}$). A process $\tau_{\mathtt{I}}$`(p)` 'hides' (renames to τ) all actions of `p` that occur in the set $\mathtt{I} \subseteq \mathtt{Act}$.

Process definitions. The heart of a μCRL specification is the `proc` section, where the behaviour of the system is declared. This section consists of recursion equations of the following form, for $n \geq 0$:

$$\texttt{proc X(}x_1\texttt{:}s_1\texttt{, ..., }x_n\texttt{:}s_n\texttt{) = t}$$

Here `X` is the process name, the x_i are variables, not clashing with the name of a function symbol of arity zero nor with a parameterless process or action name, and the s_i are data type names, expressing that the data parameters x_i are of type s_i. Moreover, `t` is a process term possibly containing occurrences of expressions $\texttt{Y(}d_1\texttt{, ..., }d_m\texttt{)}$, where `Y` is a process name and the d_i are data terms that may contain occurrences of the variables x_1, ..., x_n. In this rule, $\texttt{X(}x_1\texttt{, ..., }x_n\texttt{)}$ is declared to have the same (potential) behaviour as the process expression `t` [19].

The initial state of the specification is declared in a separate initial declaration `init` section, which is of the form

$$\texttt{init X(}d_1\texttt{, ..., }d_n\texttt{)}$$

Here d_1, ..., d_n represent the initial values of the parameters x_1, ..., x_n. In general, in μCRL specifications the `init` section is used to instantiate the data parameters of a process declaration, meaning that the d_i are data terms that do not contain variables. The `init` section may be omitted, in which case the initial behaviour of the system is left unspecified.

The time model. 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 [22] 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 paper we use a method based on the one from [10]. In short it works like this: first we define two actions: `tick` and `tick2`. The `tick` action represents the end of a time slice and the beginning of a new one. 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. This synchronisation aspect is essential if one wants to use global timing. Note that, using this technique, we get discrete time in μCRL, since we represent a time period as a number of time units.

In most cases when using time in a model the modeller would like to give normal actions priority over `tick` actions. In order to realise this χ has implicit urgent communication, but in μCRL an operator for this does not exist. We can however get similar results by using the `tick2` action and post-processing the system after linearisation (more on the latter in section 4.12).

The differences between `tick2` and `tick` are the following:

- The action `tick` is used for translating delays, while `tick2` is used to make an action delayable (which means adding a `tick2` self-loop as an alternative to this action);

– A tick action can synchronise with any number of tick or tick2 actions, but a tick2 action cannot synchronise with only tick2 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 we define a parallel composition operator X| {tick2} |Y in the following way:

```
act  tick tick' tick2 tick2'
comm tick  | tick2 = tick'
     tick  | tick  = tick'
     tick2 | tick2 = tick2'
```

$$\mathtt{X} \mid \{\mathtt{tick2}\} \mid \mathtt{Y} = \rho_{\{\mathtt{tick'}\to\mathtt{tick},\mathtt{tick2'}\to\mathtt{tick2}\}}(\partial_{\{\mathtt{tick},\mathtt{tick2}\}}(\mathtt{X} \parallel \mathtt{Y}))$$

If we then put tick2 in the set H of the encapsulation operator used at the initialisation line, we remove all tick2 actions remaining in the final system.

Using this method we can avoid problems in many cases. Say we have a system consisting of two processes A and B. If process A is waiting for process B to send a message, process B may not be able to send it yet (in other words, cannot send it within the current time unit). Process B may have to delay for any number of time units. This is possible using the new method, because process A can delay (can perform a tick2 self-loop). Now, the moment process B is able to send the message, communication will take place immediately, even though both the send and the receive action are delayable, because two tick2 actions are not able to synchronise.

Of course, problems may arise when introducing a third process C, which wants to delay at the moment A and B 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. These problems can be solved however by post-processing the linearised system. More on that in subsection 4.12.

4 The translation scheme

4.1 Linear process equations

In this paper we use a slightly extended version of the linear process equation (LPE) definition as stated in [8]. An LPE is a one-line process declaration that consists of atomic actions, summations, sequential compositions and conditionals. In particular, an LPE does not contain any parallel operators, encapsulations or hidings. 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 is its effect on the vector of data parameters.

This format resembles I/O automata [31], extended finite state machines [27], Unity processes [14] and STGA [28]. An LPE is of the following form:

$$\begin{aligned}\mathtt{X(d{:}D)} = &\sum_{\mathtt{i\in I}}\sum_{\mathtt{e_i\in D_i}} \mathtt{a_i(f_i(d,e_i)).X(g_i(d,e_i))} \lhd \mathtt{h_i(d,e_i)} \rhd \delta \ + \\ &\sum_{\mathtt{i\in I'}}\sum_{\mathtt{e_i\in D_i'}} \mathtt{a_i'(f_i'(d,e_i))}.\checkmark\mathtt{(g_i'(d,e_i))} \lhd \mathtt{h_i'(d,e_i)} \rhd \delta\end{aligned}$$

where $\mathtt{I}$, $\mathtt{I'}$ are finite index sets, $\mathtt{D}$, $\mathtt{D_i}$, $\mathtt{D_i'}$, $\mathtt{D_{a_i}}$ and $\mathtt{D_{a_i'}}$ are data types, $\mathtt{a_i}, \mathtt{a_i'} \in$ $\mathbf{Act} \cup \{\tau\}$, $\mathtt{a_i} : \mathtt{D_{a_i}}$, $\mathtt{a_i'} : \mathtt{D_{a_i'}}$, $\mathtt{f_i} : \mathtt{D} \times \mathtt{D_i} \rightarrow \mathtt{D_{a_i}}$, $\mathtt{f_i'} : \mathtt{D} \times \mathtt{D_i'} \rightarrow \mathtt{D_{a_i'}}$, $\mathtt{g_i} : \mathtt{D} \times \mathtt{D_i} \rightarrow \mathtt{D}$, $\mathtt{g_i'} : \mathtt{D} \times \mathtt{D_i'} \rightarrow \mathtt{D}$, $\mathtt{h_i} : \mathtt{D} \times \mathtt{D_i} \rightarrow \mathtt{Bool}$ and $\mathtt{h_i'} : \mathtt{D} \times \mathtt{D_i'} \rightarrow \mathtt{Bool}$.

Here the different states of the process are represented by the data parameter `d:D`. Type `D` may be a Cartesian product of n data types. Besides that the data parameter $\mathtt{e_i}$ (either of type $\mathtt{D_i}$ or $\mathtt{D_i'}$) can influence the parameter of action $\mathtt{a_i}$ (or $\mathtt{a_i'}$), the condition $\mathtt{h_i}$ (or $\mathtt{h_i'}$) and the resulting state $\mathtt{g_i}$ (or $\mathtt{g_i'}$), thereby giving LPEs a more general form. The data parameter $\mathtt{e_i}$ is typically used to let a read action range over a data domain.

The extension to the definition from [8] is the usage of $\checkmark$. Using the original definition, process `X` would terminate after executing an action $\mathtt{a_i'}$ after which it would be impossible to state anything about the end state. Here however $\checkmark\mathtt{(g_i'(d,e_i))}$ should be read as "the process enters state $\mathtt{g_i'(d,e_i)}$ after which it successfully terminates" (the process has reached an end state).

In general, when translating a χ_{t} process to an LPE the variables s_i in the scope operator of χ_{t} should be translated to parameters of the LPE (in other words, should become part of the data parameter `d:D`). Channels in χ_{t} that a process works with are mentioned as parameters in the χ_{t} specification of that process, but these should not be included in the LPE.

In both χ_{t} and μCRL one can use data types. We could go into detail concerning how an element of a data type in χ_{t} can be translated to an element of a data type in μCRL, but we will not do so here. It suffices to say that this kind of translation is rather trivial; for virtually any data type in χ_{t} one can define a corresponding abstract data type in μCRL.

4.2 Initialisation

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 specific, 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.

In general the initialisation line of a χ_{t} specification looks like this:

$$\langle x_k : ty_k, h_m : -th_m \mid p_1() \parallel \ \ldots \ \parallel p_n()\rangle$$

Here $x_k : ty_k, h_m : -th_m$ is an abbreviation for $x_1 : ty_1, ..., x_k : ty_k, h_1 : -th_1, ..., h_m :$ $-th_m$. This declares discrete variables $x_1, ..., x_k$ of types $ty_1, ..., ty_k$, respectively, and channels $h_1, ..., h_m$ that communicate information of types $th_1, ..., th_m$, respectively. Furthermore $p_1(), ..., p_n()$ are processes.

The variables declared at the initialisation line are global variables. These cannot be translated in a straightforward fashion, since μCRL does not have them. It is possible however to achieve the same results by having all 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 4.13.

Each channel which is declared at the initialisation line should be translated to a send action, a corresponding receive action, and a corresponding communication action, defined in the `act` section of the μCRL specification. Then a rule should be added to the `comm` section, allowing the send and receive action to communicate.

The usage of the parallel composition operator at initialisation can be translated in a straightforward fashion. In the initialisation line we see the parallel composition operator being used to specify which processes make up the system (in other words, which processes run in parallel from the start). The initialisation line of a μCRL specification looks like this:

$$\texttt{init}\ \partial_{\texttt{H}}(\texttt{p}_1\texttt{()} \parallel \dots \parallel \texttt{p}_n\texttt{()})$$

Here we can see that the parallel composition operator is used in the same way.

In χ_t the parallel composition operator can also be used inside processes. In such a case the result is really a process consisting of subprocesses running in parallel. Since this usage of the parallel composition operator is not allowed in μCRL, we want to avoid these constructions in χ_t. Soon, however, there will be a new μCRL lineariser available, which does allow nested parallelism. Then this will no longer be a restriction.

The encapsulation operator of χ_t ($\partial_{\mathcal{A}}$) is implicitly used at initialisation. It can be translated by using the encapsulation operator of μCRL ($\partial_{\texttt{H}}$), making the set `H` equal to the set $\mathcal{A}$.

For the remainder of this chapter we assume that a χ_t specification ready for translation has the previously displayed initialisation line with processes $p_1(), \dots, p_n()$ not containing the parallel composition operator, the encapsulation operator and the urgent communication operator (more on the latter in section 4.12).

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 $x_n{:=}e_n$ can be translated to $\texttt{X(d:D)} = \tau.\surd\texttt{(d[}^{\texttt{e}_1}/_{\texttt{x}_1},\dots,^{\texttt{e}_n}/_{\texttt{x}_n}\texttt{])}$, in which $\surd\texttt{(d[}^{\texttt{e}_1}/_{\texttt{x}_1},\dots,^{\texttt{e}_n}/_{\texttt{x}_n}\texttt{])}$ means that you end up in a state where $\texttt{e}_1,\dots,\texttt{e}_n$ have been substituted for $\texttt{x}_1,\dots,\texttt{x}_n$ 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 then becomes $\texttt{X(d:D)} = \tau.\surd\texttt{(d)}$.

Table 1. Translation of send processes

χ_t	μCRL
$h\,!\,e_n$	`X(d:D) = sh(e1,...,en).√(d) + tick2.X(d)`
$h\,!!\,e_n$	`X(d:D) = sh(e1,...,en).√(d)`
$h\,!$	`X(d:D) = sh.√(d) + tick2.X(d)`
$h\,!!$	`X(d:D) = sh.√(d)`

The send process. In μCRL channels are not available as a type, like they are in χ_t. Instead one can define actions and synchronise these 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' 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, this has to be translated to $\mathtt{X(d:D)} = \mathtt{sh(e_1,...,e_n).\checkmark(d)\ +\ tick2.X(d)}$ with `sh` being the action of sending something over the channel h.

All variants of the χ_t send process can be found in table 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 delay-ability 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 4.12.

The receive process. As mentioned in the previous paragraph μCRL does not work with channels, but one can define send and receive actions and force them to communicate. Receive actions traditionally begin with the letter `r`, which would mean in this case that the receive action would be defined as `rh`.

The process $h\,?\,x_n$ translates to

$$\mathtt{X(d:D)} = \sum_{\mathtt{y_1:Ty_1}} \cdots \sum_{\mathtt{y_n:Ty_n}} \mathtt{(rh(y_1,\ldots,y_n).\checkmark(d[^{y_1}/_{x_1},\ldots,^{y_n}/_{x_n}]))\ +\ tick2.X(d)}$$

with $\mathtt{Ty_i}$ being the type of both $\mathtt{x_i}$ and $\mathtt{y_i}$, respectively, for all $\mathtt{i}$ $(\mathtt{1} \leq \mathtt{i} \leq \mathtt{n})$.

All variants of the χ_t receive process can be found in table 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 similar remark to the one in the previous paragraph holds for the receive process.

Table 2. Translation of receive processes

χ_t	μCRL
$h\,?\,e_n$	$\mathtt{X(d:D)} = \sum_{\mathtt{y_1:Ty_1}} \cdots \sum_{\mathtt{y_n:Ty_n}} \mathtt{(rh(y_1,\ldots,y_n).\checkmark(d[^{y_1}/_{x_1},\ldots,^{y_n}/_{x_n}]))}$ $\mathtt{+\ tick2.X(d)}$
$h\,??\,e_n$	$\mathtt{X(d:D)} = \sum_{\mathtt{y_1:Ty_1}} \cdots \sum_{\mathtt{y_n:Ty_n}} \mathtt{(rh(y_1,\ldots,y_n).\checkmark(d[^{y_1}/_{x_1},\ldots,^{y_n}/_{x_n}]))}$
$h\,?$	$\mathtt{X(d:D) = rh.\checkmark(d)\ +\ tick2.X(d)}$
$h\,??$	$\mathtt{X(d:D) = rh.\checkmark(d)}$

The delay process. The translation of the delay process Δt is highly dependent on the time model used in μCRL. Therefore the reader should be aware of this time model as described in the time model paragraph of section 3. Note that while χ_t uses continuous time, this time model 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 we can translate the delay to the following LPE, where `t` and $\mathtt{t_0}$ are both translations of t:

$$\mathtt{X}(\underline{\mathtt{t:Nat}}, \underline{\mathtt{t_0:Nat}}, \mathtt{d:D}) = \mathtt{tick.X(t-1, t_0, d)} \lhd \mathtt{t} > 0 \rhd \delta\ + \\ \tau.\checkmark(\mathtt{t_0, t_0, d}) \lhd \mathtt{t} = 0 \rhd \delta$$

We cannot set the initial value of `t` and $\mathtt{t_0}$ in process `X`. We have to do that in the initialisation line. We introduce a set V, which contains all initial values of process parameters. When writing the initialisation line, we obtain the parameter values from this set. For now we set the initial value of both `t` and $\mathtt{t_0}$ in V to the value of t. Counter $\mathtt{t_0}$ is used to be able to reset counter `t` to its initial value. It is important that this is done after executing process `X` to allow the possibility to repeat execution of `X` by means of the repetition operator (see section 4.9). Finally note that both `t` and $\mathtt{t_0}$ are underlined in the parameter list. This has been done to indicate that these parameters are specially marked. For the use of this see section 4.9.

4.4 Delay operator

When discussing the translation of χ_t operators in the upcoming sections, two LPEs `P` and `Q` will be used. These processes have the following form:

$$\begin{aligned}
\mathtt{P(d:D)} &= \sum_{\mathtt{i\in I}} \sum_{\mathtt{e_i\in D_i}} \mathtt{a_i(f_{P_i}(d,e_i)).P(g_{P_i}(d,e_i))} \lhd \mathtt{h_{P_i}(d,e_i)} \rhd \delta\ + \\
&\quad \sum_{\mathtt{i\in I'}} \sum_{\mathtt{e_i\in D_i'}} \mathtt{a_i'(f_{P_i}'(d,e_i)).\checkmark(g_{P_i}'(d,e_i))} \lhd \mathtt{h_{P_i}'(d,e_i)} \rhd \delta \\
\mathtt{Q(d':D')} &= \sum_{\mathtt{j\in J}} \sum_{\mathtt{e_j\in D_j}} \mathtt{a_j(f_{Q_j}(d',e_j)).Q(g_{Q_j}(d',e_j))} \lhd \mathtt{h_{Q_j}(d',e_j)} \rhd \delta\ + \\
&\quad \sum_{\mathtt{j\in J'}} \sum_{\mathtt{e_j\in D_j'}} \mathtt{a_j'(f_{Q_j}'(d',e_j)).\checkmark(g_{Q_j}'(d',e_j))} \lhd \mathtt{h_{Q_j}'(d',e_j)} \rhd \delta
\end{aligned}$$

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

In the upcoming sections we use a function $\mathcal{T}$: $\chi \rightarrow \mu$CRL which gets a χ_t process as input and returns an LPE as output. This function provides a translation by induction on the structure of a χ_t process.

Consider a process $\Delta_t(p)$ with the LPE P $= \mathcal{T}(p)$. Then $\mathcal{T}(\Delta_t(p))$ is defined as follows, where t and t_0 are both translations of t:

$$\begin{aligned}
&\mathtt{X}(\underline{\mathtt{t:Nat}},\ \underline{\mathtt{t_0:Nat}},\ \mathtt{d:D}) = \\
&\quad \mathtt{tick}.\mathtt{X}(\mathtt{t}-1,\mathtt{t_0},\mathtt{d}) \triangleleft \mathtt{t} > 0 \triangleright \delta\ + \\
&\quad \sum_{\mathtt{i}\in\mathtt{I}}\sum_{\mathtt{e_i}\in\mathtt{D_i}} \mathtt{a_i}(\mathtt{f_{P_i}}(\mathtt{d},\mathtt{e_i})).\mathtt{X}(\mathtt{t},\mathtt{t_0},\mathtt{g_{P_i}}(\mathtt{d},\mathtt{e_i})) \triangleleft \mathtt{h_{P_i}}(\mathtt{d},\mathtt{e_i}) \wedge \mathtt{t} = 0 \triangleright \delta\ + \\
&\quad \sum_{\mathtt{i}\in\mathtt{I'}}\sum_{\mathtt{e_i}\in\mathtt{D_i'}} \mathtt{a_i}(\mathtt{f_{P_i}}'(\mathtt{d},\mathtt{e_i})).\checkmark(\mathtt{t_0},\mathtt{t_0},\mathtt{g_{P_i}}'(\mathtt{d},\mathtt{e_i})) \triangleleft \mathtt{h_{P_i}}'(\mathtt{d},\mathtt{e_i}) \wedge \mathtt{t} = 0 \triangleright \delta
\end{aligned}$$

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

1. The counters t and t_0 have been introduced. They are used in the same way as they are in the delay process. The parameters t and t_0 have been underlined to indicate that they are specially marked. More on this in section 4.9.
2. The guards of the lines that originate from LPE P have been extended with the boolean expression t=0.

4.5 Delay enabling operator

Assume we have a process $[p]$ with the LPE P $= \mathcal{T}(p)$. Then $\mathcal{T}([p])$ is defined as follows, where t and t_0 are both translations of t:

$$\begin{aligned}
&\mathtt{X}(\mathtt{n:Nat},\ \mathtt{d:D}) = \\
&\quad \sum_{\mathtt{i}\in\mathtt{I}}\sum_{\mathtt{e_i}\in\mathtt{D_i}} \mathtt{a_i}(\mathtt{f_{P_i}}(\mathtt{d},\mathtt{e_i})).\mathtt{X}(1,\mathtt{g_{P_i}}(\mathtt{d},\mathtt{e_i})) \triangleleft \mathtt{h_{P_i}}(\mathtt{d},\mathtt{e_i}) \triangleright \delta\ + \\
&\quad \sum_{\mathtt{i}\in\mathtt{I'}}\sum_{\mathtt{e_i}\in\mathtt{D_i'}} \mathtt{a_i}(\mathtt{f_{P_i}}'(\mathtt{d},\mathtt{e_i})).\checkmark(0,\mathtt{g_{P_i}}'(\mathtt{d},\mathtt{e_i})) \triangleleft \mathtt{h_{P_i}}'(\mathtt{d},\mathtt{e_i}) \triangleright \delta\ + \\
&\quad \mathtt{tick2}.\mathtt{X}(\mathtt{d}) \triangleleft \mathtt{n} = 0 \triangleright \delta
\end{aligned}$$

A counter n has been introduced. It has type Nat but in practise n $\in$ {0, 1}. This counter is set to 0 before executing X (this is set in the 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 after this execution an end state has not been reached, counter n is set to 1, resulting in the added tick2 action being disabled.

4.6 Guard operator

Consider a process $b \rightarrow p$ with b being a boolean expression. Say LPE P $= \mathcal{T}(p)$ and b is a translation of the guard b. Finally we say that the finite index set

$\mathtt{I} = \mathtt{I_n} \cup \mathtt{I_t}$, with $\mathtt{I_n} \cap \mathtt{I_t} = \emptyset$, $\mathtt{I_n}$ being a set of indexes of all actions in P which are not tick or tick2 actions (i.e. 'normal' actions) and $\mathtt{I_t}$ being a set of indexes of all actions in P which are either tick or tick2 actions. For I' we do not have to do a similar thing since $\mathtt{I_t}$' will always be empty. A process never terminates after executing a tick or tick2 action; after a tick action there is always eventually a normal action (see the translations of the delay process and the delay operator) and tick2 actions only occur in self-loops.

Now $\mathcal{T}(b \to p)$ is defined as follows:

$$
\begin{array}{l}
\mathtt{X(n{:}Nat,\ d{:}D)} = \\
\displaystyle\sum_{\mathtt{i}\in\mathtt{I_n}}\sum_{\mathtt{e_i}\in\mathtt{D_i}} \mathtt{a_i(f_{P_i}(d,e_i)).X(1,g_{P_i}(d,e_i))} \lhd \mathtt{h_{P_i}(d,e_i)} \wedge (\mathtt{n}=1 \vee \mathtt{b}) \rhd \delta\ + \\
\displaystyle\sum_{\mathtt{i}\in\mathtt{I_t}}\sum_{\mathtt{e_i}\in\mathtt{D_i}} \mathtt{a_i(f_{P_i}(d,e_i)).X(n,g_{P_i}(d,e_i))} \lhd \mathtt{h_{P_i}(d,e_i)} \wedge (\mathtt{n}=1 \vee \mathtt{b}) \rhd \delta\ + \\
\displaystyle\sum_{\mathtt{i}\in\mathtt{I'}}\sum_{\mathtt{e_i}\in\mathtt{D_i'}} \mathtt{a_i(f_{P_i}'(d,e_i)).\checkmark(0,g_{P_i}'(d,e_i))} \lhd \mathtt{h_{P_i}'(d,e_i)} \wedge (\mathtt{n}=1 \vee \mathtt{b}) \rhd \delta\ + \\
\mathtt{tick2.X(n,d)} \lhd \mathtt{n}=0 \wedge \neg\mathtt{b} \rhd \delta
\end{array}
$$

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 Nat but in practise n $\in$ {0, 1}. This counter is initially 0 (we set this in the set 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 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 (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.

4.7 Sequential composition operator

Assume we have the χ_t process $p;\ q$ with the LPE $\mathtt{P} = \mathcal{T}(p)$ and the LPE $\mathtt{Q} = \mathcal{T}(q)$. Now we define $\mathcal{T}(p;\ q)$ as follows:

$$
\begin{aligned}
&\mathtt{X(n{:}Nat,\ d{:}D,\ d'{:}D')} = \\
&\sum_{\mathtt{i}\in\mathtt{I}}\sum_{\mathtt{e_i}\in\mathtt{D_i}} \mathtt{a_i(f_{P_i}(d,e_i)).X(0,g_{P_i}(d,e_i),d')} \lhd \mathtt{h_{P_i}(d,e_i)} \wedge \mathtt{n} = 0 \rhd \delta\ + \\
&\sum_{\mathtt{i}\in\mathtt{I'}}\sum_{\mathtt{e_i}\in\mathtt{D_i'}} \mathtt{a_i(f_{P_i}'(d,e_i)).X(1,g_{P_i}'(d,e_i),d')} \lhd \mathtt{h_{P_i}'(d,e_i)} \wedge \mathtt{n} = 0 \rhd \delta\ + \\
&\sum_{\mathtt{j}\in\mathtt{J}}\sum_{\mathtt{e_j}\in\mathtt{D_j}} \mathtt{a_j(f_{Q_j}(d',e_j)).X(1,d,g_{Q_j}(d',e_j))} \lhd \mathtt{h_{Q_j}(d',e_j)} \wedge \mathtt{n} = 1 \rhd \delta\ + \\
&\sum_{\mathtt{j}\in\mathtt{J'}}\sum_{\mathtt{e_j}\in\mathtt{D_j'}} \mathtt{a_j(f_{Q_j}'(d',e_j))}.\checkmark\mathtt{(0,d,g_{Q_j}'(d',e_j))} \lhd \mathtt{h_{Q_j}'(d',e_j)} \wedge \mathtt{n} = 1 \rhd \delta
\end{aligned}
$$

A counter `n` has been introduced to regulate the order of execution. Initially this counter has value `0`, 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`.

4.8 Alternative composition operator

In this section we give a translation of the χ_t process $p\ [\!]\ q$, which chooses non-deterministically 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 time mechanism of χ_t; if both alternatives p and q can delay, then they delay *together*. If only one alternative can delay and furthermore no actions can be executed at all then there is a deadlock. Finally, time does not make a choice, meaning that if the process delays before a choice for one of the alternatives has been made the process still has to make this choice after that delay.

Say we have the χ_t process $p\ [\!]\ q$ with the LPE $\mathtt{P} = \mathcal{T}(p)$ and the LPE $\mathtt{Q} = \mathcal{T}(q)$. Furthermore we say that the finite index set $\mathtt{I} = \mathtt{I_n} \cup \mathtt{I_t}$ (similar to section 4.6). Finally we say that $\mathtt{I_t} = \mathtt{I_{t1}} \cup \mathtt{I_{t2}}$, with $\mathtt{I_{t1}} \cap \mathtt{I_{t2}} = \emptyset$, $\mathtt{I_{t1}}$ being a set of indexes of all occurrences of the `tick` action in `P` and $\mathtt{I_{t2}}$ being a set of indexes of all occurrences of the `tick2` action in `P`. In a similar way we define

$J = J_n \cup J_t$ and $J_t = J_{t1} \cup J_{t2}$. Now we define $\mathcal{T}(p \,[\!]\, q)$ as follows:

$$
\begin{aligned}
&\mathtt{X(n:Nat,\ d:D,\ d':D')} = \\
&\sum_{i\in I_n}\sum_{e_i\in D_i} a_i(f_{P_i}(d,e_i)).X(1,g_{P_i}(d,e_i),d') \lhd h_{P_i}(d,e_i) \wedge (n=0 \vee n=1) \rhd \delta\ + \\
&\sum_{i\in I_t}\sum_{e_i\in D_i} a_i(f_{P_i}(d,e_i)).X(n,g_{P_i}(d,e_i),d') \lhd h_{P_i}(d,e_i) \wedge n=1 \rhd \delta\ + \\
&\sum_{i\in I'}\sum_{e_i\in D_i'} a_i(f_{P_i}'(d,e_i)).\checkmark(0,g_{P_i}'(d,e_i),d') \lhd h_{P_i}'(d,e_i) \wedge (n=0 \vee n=1) \rhd \delta\ + \\
&\sum_{j\in J_n}\sum_{e_j\in D_j} a_j(f_{Q_j}(d',e_j)).X(2,d,g_{Q_j}(d',e_j)) \lhd h_{Q_j}(d',e_j) \wedge (n=0 \vee n=2) \rhd \delta\ + \\
&\sum_{j\in J_t}\sum_{e_j\in D_j} a_j(f_{Q_j}(d',e_j)).X(n,d,g_{Q_j}(d',e_j)) \lhd h_{Q_j}(d',e_j) \wedge n=2 \rhd \delta\ + \\
&\sum_{j\in J'}\sum_{e_j\in D_j'} a_j(f_{Q_j}'(d',e_j)).\checkmark(0,d,g_{Q_j}'(d',e_j)) \\
&\qquad\qquad\qquad \lhd h_{Q_j}'(d',e_j) \wedge (n=0 \vee n=2) \rhd \delta\ + \\
&\sum_{i\in I_{t1}}\sum_{j\in J_{t2}}\sum_{e_i\in D_i}\sum_{e_j\in D_j} \mathtt{tick}.X(n,g_{P_i}(d,e_i),g_{Q_j}(d',e_j)) \\
&\qquad\qquad\qquad \lhd h_{P_i}(d,e_i) \wedge h_{Q_j}(d',e_j) \wedge n=0 \rhd \delta\ + \\
&\sum_{i\in I_{t2}}\sum_{j\in J_{t1}}\sum_{e_i\in D_i}\sum_{e_j\in D_j} \mathtt{tick}.X(n,g_{P_i}(d,e_i),g_{Q_j}(d',e_j)) \\
&\qquad\qquad\qquad \lhd h_{P_i}(d,e_i) \wedge h_{Q_j}(d',e_j) \wedge n=0 \rhd \delta\ + \\
&\sum_{i\in I_{t1}}\sum_{j\in J_{t1}}\sum_{e_i\in D_i}\sum_{e_j\in D_j} \mathtt{tick}.X(n,g_{P_i}(d,e_i),g_{Q_j}(d',e_j)) \\
&\qquad\qquad\qquad \lhd h_{P_i}(d,e) \wedge h_{Q_j}(d',e) \wedge n=0 \rhd \delta\ + \\
&\sum_{i\in I_{t2}}\sum_{j\in J_{t2}}\sum_{e_i\in D_i}\sum_{e_j\in D_j} \mathtt{tick2}.X(n,d,d') \lhd h_{P_i}(d,e) \wedge h_{Q_j}(d',e) \wedge n=0 \rhd \delta
\end{aligned}
$$

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

1. A counter n has been introduced. It has type Nat but in practise $n \in \{0,\ 1,\ 2\}$.
2. In the first line we find the 'normal' actions that originally are not at the end of process P (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 `tick2` 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 `tick2` 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 `tick2` 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 `tick2` 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 `tick2` 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 `tick2` occurrences in P with `tick` occurrences in Q, `tick` occurrences in P and Q and `tick2` occurrences in P and Q respectively.

So in line two and five the `tick` and `tick2` 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 $p \,[\!]\, q$, where p and q delay together if they can both delay and no delay will happen if one of them cannot.

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 $\mathtt{P} = \mathcal{T}(p)$. Now we define $\mathcal{T}(*p)$ as follows:

$$
\begin{aligned}
&\mathtt{X(d{:}D)} = \\
&\sum_{i\in I}\sum_{e_i\in D_i} a_i(f_{P_i}(d,e_i)).X(g_{P_i}(d,e_i)) \lhd h_{P_i}(d,e_i) \rhd \delta \; + \\
&\sum_{i\in I'}\sum_{e_i\in D_i'} a_i(f_{P_i}'(d,e_i)).X(\underline{g_{P_i}'(d,e_i)}) \lhd h_{P_i}'(d,e_i) \rhd \delta
\end{aligned}
$$

In the LPE the check-mark ($\checkmark$) has been replaced by X, resulting in executing X from the beginning again every time X has executed the final action in the LPE. When repeating the execution the LPE automatically begins with the first action, which is assured 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 underlined. This is done to indicate that all marked parameters t in d (see sections 4.3 and 4.4) are assigned the values of their accompanying parameters t_0. The reason for this is that should process P contain an alternative composition, it is not always the case that all timers are reset to their initial values upon termination.

4.10 Guarded repetition operator

Say we have the χ_t process $*b : p$ with the LPE $\mathtt{P} = \mathcal{T}(p)$ and $\mathtt{b}$ is a translation of the guard b. Now we define $\mathcal{T}(*b : p)$ as follows:

$$
\begin{aligned}
&\mathtt{X(n{:}Nat,\ d{:}D)} = \\
&\quad \tau.\mathtt{X}(1,\mathtt{d}) \lhd \mathtt{n} = 0 \land \mathtt{b} \rhd \delta\ + \\
&\quad \sum_{\mathtt{i}\in\mathtt{I}} \sum_{\mathtt{e_i}\in\mathtt{D_i}} \mathtt{a_i}(\mathtt{f}_{\mathtt{P_i}}(\mathtt{d},\mathtt{e_i})).\mathtt{X}(\mathtt{n},\mathtt{g}_{\mathtt{P_i}}(\mathtt{d},\mathtt{e_i})) \lhd \mathtt{h}_{\mathtt{P_i}}(\mathtt{d},\mathtt{e_i}) \land \mathtt{n} = 1 \rhd \delta\ + \\
&\quad \sum_{\mathtt{i}\in\mathtt{I'}} \sum_{\mathtt{e_i}\in\mathtt{D_i'}} \mathtt{a_i}(\mathtt{f}_{\mathtt{P_i}}'(\mathtt{d},\mathtt{e_i})).\mathtt{X}(0,\mathtt{g}_{\mathtt{P_i}}'(\mathtt{d},\mathtt{e_i})) \lhd \mathtt{h}_{\mathtt{P_i}}'(\mathtt{d},\mathtt{e_i}) \land \mathtt{n} = 1 \rhd \delta\ + \\
&\quad \tau.\checkmark(\mathtt{n},\mathtt{d}) \lhd \mathtt{n} = 0\ \land \neg\mathtt{b} \rhd \delta\ + \\
&\quad \mathtt{tick2}.\mathtt{X}(\mathtt{n},\mathtt{d}) \lhd \mathtt{n} = 0 \land \neg\mathtt{b} \rhd \delta
\end{aligned}
$$

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

1. A counter `n` has been introduced.
2. In the first line the process can do a τ action if the boolean expression `b` equals `T`. After this the LPE `P` can be executed.
3. In the second line $\mathtt{P}(\mathtt{g}_{\mathtt{P_i}}(\mathtt{d}))$ is replaced by $\mathtt{X}(1,\ \mathtt{g}_{\mathtt{P_i}}(\mathtt{d}))$.
4. In the third line $\checkmark(\mathtt{g}_{\mathtt{P_i}}'(\mathtt{d}))$ is replaced by $\mathtt{X}(0,\ \mathtt{g}_{\mathtt{P_i}}'(\mathtt{d}))$.
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.
6. The fifth line makes sure that the process is delayable when trying to start executing the actions of `P` and the boolean expression `b` equals `F`.

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 χ_t the state s is used to define local programming variables or local channels. So in a way, a state is a tuple consisting of variables and channels. In μCRL the programming variables of a process can be found in its parameter `d` (initial values can be found in the initialisation line) while the channels are defined as send and receive actions globally. In other words, s is captured in μCRL by recursion parameters and global action and communication definitions. Therefore, there is no direct translation of the scope operator needed.

As the χ_t local channels are translated to μCRL global actions, some extra work is needed to make these actions seem local. In χ_t send and receive actions on local channels are automatically hidden from the outside world. This is the only abstraction which is done in χ_t. To get the same result in μCRL we have to add all send and receive actions, which are translations of send and receive actions over local channels in χ_t, to the set `I` of the abstraction operator used at the initialisation line.

4.12 Urgent communication operator

In μCRL there is no urgent communication operator. This means that it is not possible to give action transitions priority over delay transitions, which is what the urgent communication operator of χ_t does. We can however still get similar results using μCRL, which will be explained next.

In order to translate the urgent communication operator we first need to linearise the translation. This means that translating urgent communication can only be done once all other translations are ready. This reflects nicely the fact that in χ_t, urgent communication is added to a system once it is completed.

Say we have a χ_t process $\upsilon_{\mathcal{H}}(p)$ where the LPE $\mathtt{P} = \mathcal{T}(p)$ and $\mathcal{H}$ contains all channels used by p. Now we define $\mathcal{T}(\upsilon_{\mathcal{H}}(p))$ as follows:

$$\begin{aligned}
&\mathtt{X(d{:}D)} = \\
&\sum_{\mathtt{i}\in\mathtt{I_n}}\sum_{\mathtt{e_i}\in\mathtt{D_i}} \mathtt{a_i(f_i(d,e_i)).X(g_i(d,e_i))} \lhd \mathtt{h_i(d,e_i)} \rhd \delta \;+ \\
&\sum_{\mathtt{i}\in\mathtt{I_t}}\sum_{\mathtt{e_i}\in\mathtt{D_i}} \mathtt{a_i(f_i(d,e_i)).X(g_i(d,e_i))} \lhd \mathtt{h_i(d,e_i)} \wedge \neg \bigvee_{\mathtt{j}\in\mathtt{I_n}} \mathtt{h_j(d,e_i)} \rhd \delta
\end{aligned}$$

Note that the finite index set $\mathtt{I} = \mathtt{I_n} \cup \mathtt{I_t}$, as used before. In `X` there are no `tick2` actions, since these are never observable from outside the system. Variable $\mathtt{e_i}$ does not really play a role in the second line, since the `tick` action does not use it. Finally, `I'` is empty, since the linearised version of a system does not contain check-marks.

4.13 Global variables

In χ_t processes can use global variables; if one process changes the value of such a variable, the other processes may be affected by this. In μCRL there are no global variables, but it is possible to get similar results.

Say two χ_t processes A and B share a variable named x. We translate process A to `A` and process B to `B`. Both processes maintain a local copy of variable `x` (the translation of 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` 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`, 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 more processes sharing the same variable, once an `assignx` action has communicated with an `updatex` action this `updatex` communicates immediately with the `updatex` action of another process, such that in the end all processes are aware of the assignment. More specific this is the definition of the actions (in this case x is a natural number):

```
act  assignx, assignx', updatex, updatex' : Nat
comm assignx | updatex = assignx'
     updatex | updatex = updatex'
```

In order to achieve the desired behaviour described above we introduce a new parallel operator `X| SVTPC |Y` (`SVTPC` stands for *Shared Variables supporting Timed Parallel Composition*). This operator handles both the synchronising between `tick` and `tick2` actions (like `X|` {`tick2`} `|Y`) and the synchronising necessary for using shared variables. The definition is as follows: `X` | `SVTPC` | `Y` = $\rho_{\mathtt{f}}(\partial_{\mathtt{H}}(\mathtt{X} \parallel \mathtt{Y}))$, where `f` = {`tick'`→`tick`, `tick2'`→`tick2`, `assignx'`→`assignx`, `updatex'`→`updatex`} and `H` = {`tick`, `assignx`, `updatex`}.

5 An example: the system *PQ*

Concluding we look at an example χ_t specification in order to illustrate how translation works on a concrete case study. The example we use is a system consisting of the processes P and Q. The χ_t specification of the system is the following:

$P(a : !\,\mathsf{bool}\,) =$
$[\![\; i : \mathsf{nat} \; = 2$
$\mid \ast(\; i \geq 1 \rightarrow \Delta 3.0;\; a\,!\,\mathrm{false};\; i := i + 1$
$\quad [\!]\; i \geq 2 \rightarrow a\,!\,\mathrm{true};\; i := i - 1 \;)]\!]$

$Q(a : ?\,\mathsf{bool}\,) =$
$[\![\; b : \mathsf{bool}$
$\mid \ast(\; \Delta 2.0;\; a\,?\,b \;)]\!]$

$\langle a : -\mathit{bool} \mid P(a) \parallel Q(a) \rangle$

Next we translate these two processes to LPEs. After that we will linearise the translation using the μCRL toolset and then 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 we will use the data types `Nat` and `Bool`. 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` and `ca`, all three having a boolean value as parameter. 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. We define that `sa` can communicate with `ra`, forming action `ca`.

Concerning process P, 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 the set of natural numbers). Furthermore we see two guards placed in an alternative composition. Finally this 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:

$$\begin{array}{l}
\mathtt{proc\ P(n{:}Nat,\ t{:}Nat,\ t_0{:}Nat,\ i{:}Nat)} =\\
\quad \mathtt{tick.P(n,t\text{-}1,t_0,i)} \triangleleft \mathtt{t} > 0 \wedge \mathtt{n} = 0 \wedge \mathtt{i} \geq 1 \triangleright \delta\ +\\
\quad \tau\mathtt{.P(1,t_0,t_0,i)} \triangleleft \mathtt{t} = 0 \wedge \mathtt{n} = 0 \wedge \mathtt{i} \geq 1 \triangleright \delta\ +\\
\quad \mathtt{sa(F).P(2,t,t_0,i)} \triangleleft \mathtt{n} = 1 \triangleright \delta\ +\\
\quad \mathtt{tick2.P(n,t,t_0,i)} \triangleleft \mathtt{n} = 1 \triangleright \delta\ +\\
\quad \tau\mathtt{.P(0,t_0,t_0,i{+}1)} \triangleleft \mathtt{n} = 2 \triangleright \delta\ +\\
\quad \mathtt{sa(T).P(3,t,t_0,i)} \triangleleft \mathtt{n} = 0 \wedge \mathtt{i} \geq 2 \triangleright \delta\ +\\
\quad \tau\mathtt{.P(0,t_0,t_0,i\text{-}1)} \triangleleft \mathtt{n} = 3 \triangleright \delta\ +\\
\quad \mathtt{tick2.P(n,t,t_0,i)} \triangleleft \mathtt{i} < 1 \wedge \mathtt{n} = 0 \triangleright \delta
\end{array}$$

Finally we translate process Q. This is a very small process which is translated to the following LPE:

$$\begin{array}{l}
\mathtt{proc\ Q(n{:}Nat,t{:}Nat,t_0{:}Nat,b{:}Bool)} =\\
\quad \mathtt{tick.Q(n,t\text{-}1,t_0,b)} \triangleleft \mathtt{t} > 0 \wedge \mathtt{n} = 0 \triangleright \delta\ +\\
\quad \tau\mathtt{.P(1,t_0,t_0,b)} \triangleleft \mathtt{t} = 0 \wedge \mathtt{n} = 0 \triangleright \delta\ +\\
\quad \sum_{\mathtt{b_0:Bool}} \mathtt{ra(b_0).Q(0,t_0,t_0,b_0)} \triangleleft \mathtt{n} = 1 \triangleright \delta\ +\\
\quad \mathtt{tick2.Q(n,t,t_0,b)} \triangleleft \mathtt{n} = 1 \triangleright \delta
\end{array}$$

The translation is completed when we write the initialisation line:

$$\mathtt{init}\ \partial_{\{\mathtt{sa,\ ra,\ tick2}\}}\mathtt{(P(0,3,3,2)}\ \ \mid \{\mathtt{tick2}\} \mid\ \ \mathtt{Q(0,2,2,F))}$$

Here, $\mid \{\mathtt{tick2}\} \mid$ is a special operator as defined in section 3. Notice that we encapsulate `tick2` eventually, which results in the fact that the synchronisation of a number of `tick2` actions (without any `tick` action) will not lead to an action in the system.

Now that this translation is finished, we move on to linearise and post-process it to introduce urgent communication. After linearisation we have the following, where the original parameters `n`, `t` and $\mathtt{t_0}$ of LPE `Q` have been renamed to `n'`, `t'` and $\mathtt{t_0'}$ in LPE `X` to avoid name clashes:

$$\begin{array}{l}
\mathtt{proc\ X(n{:}Nat,\ t{:}Nat,\ t_0{:}Nat,\ i{:}Nat,\ n'{:}Nat,\ t'{:}Nat,\ t_0'{:}Nat,\ b{:}Bool)} =\\
\quad \mathtt{tick.X(n,t\text{-}1,t_0,i,n',t'\text{-}1,t_0',b)} \triangleleft \mathtt{t} > 0 \wedge \mathtt{n} = 0 \wedge \mathtt{i} \geq 1 \wedge \mathtt{t'} > 0 \wedge \mathtt{n'} = 0 \triangleright \delta\ +\\
\quad \mathtt{tick.X(n,t\text{-}1,t_0,i,n',t',t_0',b)} \triangleleft \mathtt{t} > 0 \wedge \mathtt{n} = 0 \wedge \mathtt{i} \geq 1 \wedge \mathtt{n'} = 1 \triangleright \delta\ +\\
\quad \mathtt{tick.X(n,t,t_0,i,n',t'\text{-}1,t_0',b)}\\
\qquad\qquad\qquad\qquad \triangleleft ((\mathtt{i} < 1 \wedge \mathtt{n} = 0) \vee \mathtt{n} = 1) \wedge \mathtt{t'} > 0 \wedge \mathtt{n'} = 0 \triangleright \delta\ +\\
\quad \mathtt{ca(T).X(3,t,t_0,i,0,t_0',t_0',T)} \triangleleft \mathtt{n} = 0 \wedge \mathtt{i} \geq 2 \wedge \mathtt{n'} = 1 \triangleright \delta\ +\\
\quad \mathtt{ca(F).X(2,t,t_0,i,0,t_0',t_0',F)} \triangleleft \mathtt{n} = 1 \wedge \mathtt{n'} = 1 \triangleright \delta\ +\\
\quad \tau\mathtt{.X(n,t,t_0,i,1,t_0',t_0',b)} \triangleleft \mathtt{t'} = 0 \wedge \mathtt{n'} = 0 \triangleright \delta\ +\\
\quad \tau\mathtt{.X(0,t_0,t_0,i\text{-}1,n',t',t_0',b)} \triangleleft \mathtt{n} = 3 \triangleright \delta\ +\\
\quad \tau\mathtt{.X(0,t_0,t_0,i{+}1,n',t',t_0',b)} \triangleleft \mathtt{n} = 2 \triangleright \delta\ +\\
\quad \tau\mathtt{.X(1,t_0,t_0,i,n',t',t_0',b)} \triangleleft \mathtt{t} = 0 \wedge \mathtt{n} = 0 \wedge \mathtt{i} \geq 1 \triangleright \delta
\end{array}$$

Post-processing LPE `X` leads to extended guards of the three lines beginning with `tick`. More specific, 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 then 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`, which is a translation of the χ system PQ, with urgent communication:

```
proc X(n:Nat, t:Nat, t0:Nat, i:Nat, n':Nat, t':Nat, t0':Nat, b:Bool) =
  tick.X(n,t-1,t0,i,n',t'-1,t0',b) ◁ t > 0 ∧ n = 0 ∧ i ≥ 1 ∧ t' > 0 ∧ n' = 0 ▷ δ +
  tick.X(n,t-1,t0,i,n',t',t0',b)
          ◁t > 0 ∧ n = 0 ∧ i ≥ 1 ∧ n' = 1∧¬(n = 0 ∧ i ≥ 2 ∧ n' = 1) ▷ δ +
  tick.X(n,t,t0,i,n',t'-1,t0',b)
                              ◁((i < 1 ∧ n = 0) ∨ n = 1) ∧ t' > 0 ∧ n' = 0 ▷ δ +
  ca(T).X(3,t,t0,i,0,t0',t0',T) ◁ n = 0 ∧ i ≥ 2 ∧ n' = 1 ▷ δ +
  ca(F).X(2,t,t0,i,0,t0',t0',F) ◁ n = 1 ∧ n' = 1 ▷ δ +
  τ.X(n,t,t0,i,1,t0',t0',b) ◁ t' = 0 ∧ n' = 0 ▷ δ +
  τ.X(0,t0,t0,i-1,n',t',t0',b) ◁ n = 3 ▷ δ +
  τ.X(0,t0,t0,i+1,n',t',t0',b) ◁ n = 2 ▷ δ +
  τ.X(1,t0,t0,i,n',t',t0',b) ◁ t = 0 ∧ n = 0 ∧ i ≥ 1 ▷ δ
```

References

1. J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.
2. D.A. van Beek, A. van der Ham, and J.E. Rooda. Modelling and Control of Process Industry Batch Production Systems. In *Proc. IFAC'02*, 2002.
3. 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. Technical Report 04-37, Eindhoven University of Technology, 2004.
4. J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification.* ACM Press Frontier Series. ACM/Addison Wesley, 1989.
5. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.
6. J.A. Bergstra and J.W. Klop. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, 1998.
7. M. Bernardo, W.R. Cleaveland, S.T. Sims, and W.J. Stewart. TwoTowers: A Tool Integrating Functional and Performance Analysis of Concurrent Systems. In *Proc. FORTE/PSTV '98*, pages 457–467. Kluwer, 1998.

8. M. Bezem and J.F. Groote. Invariants in Process Algebra with Data. In *Proc. CONCUR '94*, volume 836 of *LNCS*, pages 401–416, 1994.
9. 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 *Proc. CAV 2001*, volume 2102 of *LNCS*, pages 250–254, 2001.
10. S.C.C. Blom, N. Ioustinova, and N. Sidorova. Timed verification with μCRL. In *Proc. PSI 2003*, volume 2890 of *LNCS*, pages 178–192, 2003.
11. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.
12. E. 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. Technical Report 04-23, Eindhoven University of Technology, 2004. http://www.cwi.nl/~wijs/TIPSy.
13. V. Bos and J.J.T. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer Integrated Manufacturing*, 17:185–198, 2001.
14. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1989.
15. 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 *Proc. CAV 2002*, volume 2404 of *LNCS*, pages 359–364, 2002.
16. J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP - a protocol validation and verification toolbox. In *Proc. CAV'96*, volume 1102 of *LNCS*, pages 437–440, 1996.
17. W.J. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2000.
18. W.J. Fokkink, J.F. Groote, J. Pang, B. Badban, and J.C. van de Pol. Verifying a Sliding Window Protocol in μCRL. In *Proc. AMAST 2004*, volume 3116 of *LNCS*, pages 148–163, 2004.
19. W.J. Fokkink, J.F. Groote, and M. Reniers. Modelling Distributed Systems. Unpublished manuscript, 2002.
20. 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 *Proc. CONCUR 2002*, volume 2421 of *LNCS*, pages 1–23, 2002.
21. H. Garavel and H. Hermanns. On Combining Functional Verification and Performance Evaluation Using CADP. In *Proc. FME 2002*, volume 2391 of *LNCS*, pages 410–429, 2002.
22. J.F. Groote. The Syntax and Semantics of timed μCRL. Technical Report SEN-R9709, CWI, 1997.
23. J.F. Groote, F. Monin, and J.C. van de Pol. Checking verifications of protocols and distributed systems by computer. In *Proc. CONCUR'98*, volume 1466 of *LNCS*, pages 629–655, 1998.
24. 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.
25. H. Hermanns and J.-P. Katoen. Performance Evaluation := (Process Algebra + Model Checking) $\times$ Markov Chains. In *Proc. CONCUR 2001*, volume 2154 of *LNCS*, pages 59–81, 2001.
26. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
27. ITU-T. *Recommendation Z.100: Specification and Description Language (SDL).* ITU-T, Geneva, June 1994.
28. H. Lin. Symbolic transition graph with assignment. In V. Sassone, editor, *Proc. CONCUR'96*, number 1119 in LNCS, pages 50–65. Springer-Verlag, 1996.

29. J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* Wiley-Teubner, Chichester, Stuttgart, 1996.
30. S.P. Luttik. *Choice Quantification in Process Algebra.* PhD thesis, University of Amsterdam, 2002.
31. N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
32. S. Owre, J.M. Rushby, and N. Shankar. PVS: a Prototype Verification System. In *Proc. CADE'92*, volume 607 of *LNCS*, pages 748–752, 1992.
33. J.C. van de Pol. A prover for the μCRL toolset with applications. Technical Report SEN-R0106, CWI, Amsterdam, 2001.