CONCURRENCY THEORY: A HISTORICAL PERSPECTIVE ON COINDUCTION AND PROCESS CALCULI

Jos Baeten, Davide Sangiorgi

Readers: Luca Aceto and Robert van Glabbeek

1 INTRODUCTION

Computer science is a very young science, that has its origins in the middle of the twentieth century. For this reason, describing its history, or the history of an area of research in computer science, is an activity that receives scant attention.

The discipline of computer science does not have a clear demarcation, and even its name is a source of debate. Computer science is sometimes called informatics or information science or information and communication science. A quote widely attributed to Edsger Dijkstra is

> Computer science is no more about computers than astronomy is about telescopes.

If computer science is not about computers, what is it about? We claim it is about *behavior*. This behavior can be exhibited by a computer executing a program, but equally well by a human being performing a series of actions. In general, behavior can be shown by any *agent*. Here, an agent is simply an entity that can act. So, computer science is about behavior, about processes, about things happening over time. We are concerned with agents that are executing actions, taking input, emitting output, communicating with other agents.

Thus, an agent has behavior, e.g., the execution of a software system, the actions of a machine or even the actions of a human being. Behavior is the total of events or actions that an agent can perform, the order in which they can be executed and maybe other aspects of this execution such as timing, probabilities or evolution. Always, we describe certain aspects of behavior, disregarding other aspects, so we are considering an abstraction or idealization of the 'real' behavior. Rather, we can say that we have an *observation* of behavior, and an action is the chosen unit of observation.

But behavior is also the subject of dynamics or mechanics in physics. How does computer science distinguish itself from these fields? Well, the most important difference is that dynamics or mechanics considers behavior using continuous mathematics (differential equations, analysis), and that computer science considers behavior using discrete mathematics (algebra, logic).

So, behavior considered in computer science is *discrete*, takes place at separate moments in time, and we can observe separate things happening consecutively, different actions are separated in time. This is why a process is sometimes also called a *discrete event system*. Discrete behavior is important in systems biology, e.g. considering metabolic pathways or behavior of a living cell or components of a cell. We see more and more scientists active in systems biology with a computer science background. Another example of discrete behavior is the working of an organization or part of an organization. This is called workflow.

There are two more ingredients involved before we can get to a definition and a demarcation of computer science. These are interaction and information. Computer science started off considering a single agent executing actions by itself, but with the birth of concurrency theory, it was realized that *interaction* is also an essential part of computer science. Agents interact amongst themselves, and interact with the outside world. Usually, a computer is not stand-alone, with limited interaction with the user, executing a batch process, but is always connected to other devices and the Internet. *Information* is the stuff of informatics, the things that are communicated, processed, transformed, sent around.

So now we have all ingredients in place and can formulate the following definition of computer science. It may be obvious that we find informatics a better name than computer science.

> Informatics or computer science is the study of discrete behavior of interacting information processing agents.

Given this definition of informatics, we can explain familiar notions in terms of it. A computer program is a prescription of behavior, an interpreter or compiler can turn this into specific behavior. An algorithm is a description of behavior. Computation refers to sequential behavior, not considering interaction. Communication is interaction with information exchange. Data is a manifestation of information. Intelligence has to do with a comparison between different agents, in particular between a human being and a computer.

Finally, concurrency theory is that part of the foundations of computer science where interaction is considered explicitly.

Acknowledgements

The authors are grateful to Luca Aceto and Rob van Glabbeek for comments on an earlier draft of the paper. Sangiorgi's work has been partially supported by the ANR project 12IS02001 "PACE".

2 CONCURRENCY

The simplest model of behavior is to see behavior as an input/output function. A value or input is given at the beginning of the process, and at some moment there is a(nother) value as outcome or output. This view is in close agreement with the classic notion of "algorithmic problem", which is described by giving its legal inputs and, for each legal input, the expected output. The input/output model was used to advantage as the simplest model of the behavior of a computer program in computer science, from the start of the subject in the middle of the twentieth century. It was instrumental in the development of (finite state) *automata theory*. In automata theory, a process is modeled as an automaton. An automaton has a number of *states* and a number of *transitions*, going from one state to a(nother) state. A transition denotes the execution of an (elementary) action, the basic unit of behavior. Besides, there is an initial state (sometimes, more than one) and a number of final states. A behavior is a run, i.e. a path from the initial state to a final state. Important from the beginning is when to consider two automata equal, expressed by a notion of equivalence. On automata, the basic notion of equivalence is language equivalence: a behavior is characterized by the set of executions from the initial state to a final state.

Later on, this model was found to be lacking in several situations. Basically, what is missing is the notion of *interaction*: during the execution from initial state to final state, an agent may interact with another agent. This is needed in order to describe parallel or distributed systems, or so-called *reactive* systems (see [Harel and Pnueli, 1985]). When dealing with interacting systems, we say we are doing *concurrency theory*, so concurrency theory is the theory of interacting, parallel and/or distributed systems.

The history of concurrency theory can be traced back to the early sixties of the twentieth century with the theory of Petri nets, conceived by Petri starting from his thesis in 1962 [Petri, 1962]. This remained, for some time, a separate strand in the development of the foundations of computer science, with not so many connections to other foundational research. On the other hand, the theory of Petri nets is a strong and continuing contribution to the foundations of computer science, and it has yielded many and beautiful results. The history of Petri nets has been described elsewhere (see e.g. [Brauer and Reisig, 2006]), so we will not consider it any further at this point.

In 1970, we can distinguish three main styles of formal reasoning about computer programs, focusing on giving semantics (meaning) to programming languages.

1. Operational semantics. A computer program is modeled as an execution of an abstract machine, an automaton. A state of such a machine is a valuation of variables, a transition between states describes the effect of an elementary program instruction. Pioneer of this field is McCarthy [McCarthy, 1963].

2. Denotational semantics. This is more abstract than operational semantics; computer programs are usually modeled by a function transforming input

into output. Pioneers of this line of research are Scott and Strachey [Scott and Strachey, 1971].

3. Axiomatic semantics. Here, emphasis is put on proof methods to prove programs correct. Central notions are program assertions, proof triples consisting of precondition, program statement and postcondition, and invariants. Pioneers are Floyd [Floyd, 1967] and Hoare [Hoare, 1969].

Then, the question was raised how to give semantics to programs containing a notion of parallelism. It was found that this is difficult using the methods of denotational, operational or axiomatic semantics as they existed at that time, although several attempts were made (later on, it became clear how to extend the different types of semantics to parallel programming, see e.g. [Owicki and Gries, 1976] or [Plotkin, 1976]).

There are two paradigm shifts that needed to be made, before a theory of parallel programs could be developed. First of all, the idea of a behavior as an input/output function needed to be abandoned. A program could still be modeled as an automaton, but the notion of language equivalence is no longer appropriate. This is because the interaction a process has between input and output influences the outcome, disrupting functional behavior. Secondly, the notion of *global* variables needed to be overcome. Using global variables, a state of a modeling automaton is given as a valuation of the program variables, that is, a state is determined by the values of the variables. The independent execution of parallel processes makes it difficult or impossible to determine the values of global variables at a given moment. It turned out to be simpler to let each process have its own local variables, and to denote exchange of information explicitly.

For each of these paradigm shifts, we will consider an important notion exemplifying this development. For the shift away from functional behavior we consider the history of *bisimulation*, for the shift away from global variables we consider the history of *process calculi*. Examining the developments through which certain concepts have come to light and have been discovered is not a matter of purely intellectual curiosity, but it also useful to understand the concepts themselves (e.g., problems and motivations behind them, relationship with other concepts, alternatives, etc.).

3 BISIMULATION AND COINDUCTION

Bisimulation and coinduction are generally considered as one of the most important contributions of concurrency theory to computer science. In concurrency, the bisimulation equality, called *bisimilarity*, is the most studied form of behavioural equality for processes, and is widely used, for a number of reasons, notably the following ones.

- Bisimilarity is accepted as *the finest behavioural equivalence* one would like to impose on processes.

- The *bisimulation proof method* is exploited *to prove equalities* among processes. This occurs even when bisimilarity is not the behavioural equivalence chosen for the processes. For instance, one may be interested in trace equivalence and yet use the bisimulation proof method, since bisimilarity implies trace equivalence.
- The efficiency of the algorithms for bisimilarity checking and the compositionality properties of bisimilarity are exploited to *minimise* the state-space of processes.
- Bisimilarity, and variants of it such as similarity, are used *to abstract* from certain details of the systems of interest. For instance, we may want to prove behavioural properties of a server that do not depend on the data that the server manipulates. Further, abstracting from the data may turn an infinite-state server into a finite one.

Bisimulation has also spurred the study of coinduction; indeed bisimilarity is an example of a coinductive definition, and the bisimulation proof method is an instance of the coinduction proof method.

Bisimulation and, more generally, coinduction are employed today in a number of areas of computer science: functional languages, object-oriented languages, types, data types, domains, databases, compiler optimisations, program analysis, verification tools, etc.. Today, they are also used in other fields, e.g., artificial intelligence, cognitive science, mathematics, modal logics, philosophy, and physics.

In this section, we look at the origins of bisimulation (and bisimilarity). Bisimulation has been discovered independently, and more or less at the same time, not only in computer science, but also in philosophical logic (more precisely, modal logics), and in set theory. Thus while we focus on computer science, we also give an account of the discovery of the concepts in these other areas. It is fair to say, however, that the motivations for the study of bisimulation and coinduction that came from computer science were decisive in the development of the concept and its wide success. For more details we refer to [Sangiorgi, 2009]

In computer science, philosophical logic, and set theory, bisimulation has been derived through refinements of notions of morphism between algebraic structures. Roughly, morphisms are maps (i.e., functions) that are "structure-preserving". The notion is fundamental in all mathematical theories in which the objects of study have some kind of structure, or algebra. The most basic forms of morphism are the *homomorphisms*. These essentially give us a way of embedding a structure (the source) into another one (the target), so that all the relations in the source are present in the target. The converse however, need not be true; for this, stronger notions of morphism are needed. One such notion is *isomorphism*, which is however extremely strong—isomorphic structures must be essentially the same, i.e., "algebraically identical". It is the quest for notions in between homomorphism and isomorphism that led to the discovery of bisimulation.

The kind of structures studied in computer science, in philosophical logic, and in set theory were forms of rooted directed graphs. On such graphs bisimulation is coarser than graph isomorphism because, intuitively, bisimulation allows us to observe a graph only through the movements that are possible along its edges. By contrast, with isomorphisms the identity of the nodes is observable too. For instance, isomorphic graphs have the same number of nodes, which need not be the case for bisimilar graphs (bisimilarity between two graphs indicates that their roots are related in a bisimulation).

Before digging into history, however, we recall a few basic definitions and results for bisimulation and bisimilarity. For this, we use some very simple set theory; a reader even with little knowledge of the topic should be able to grasp the essential ideas.

3.1 Bisimulation

We present bisimulation on *Labelled Transition Systems* (LTSs) because these are the most common structures on which bisimulation has been studied. LTSs are essentially "edge-labelled" directed graphs. Bisimulation can be defined on variant structures, such as relational structures (i.e., unlabeled directed graphs), non-deterministic finite automata or Kripke structures, in a similar way.

We let $\mathcal{R}$ range over relations on sets, i.e., if $\wp$ denotes the powerset construct, then a relation $\mathcal{R}$ on a set W is an element of $\wp(W \times W)$. The composition of relations $\mathcal{R}_1$ and $\mathcal{R}_2$ is $\mathcal{R}_1\mathcal{R}_2$ (i.e., $(s, s') \in \mathcal{R}_1\mathcal{R}_2$ holds if for some s'', both $(s, s'') \in \mathcal{R}_1$ and $(s'', s') \in \mathcal{R}_2$ hold). We often use the infix notation for relations; hence $s \mathrel{\mathcal{R}} t$ means $(s, t) \in \mathcal{R}$.

DEFINITION 1 (Labelled Transition Systems). A *Labelled Transition System* is a triple $(W, \mathit{Act}, \{\xrightarrow{a} : a \in \mathit{Act}\})$ with domain W as above, set of *labels Act*, and for each label a, a (binary) relation $\xrightarrow{a}$ on W called the *transition relation*.

In the two definitions above, the elements of W will be called *states* or *points*, sometimes even *processes* as this is the usual terminology in concurrency. We use s, t to range over such elements, and μ to range over the labels in *Act*. Following the infix notation for relations, we write $s \xrightarrow{\mu} t$ when $(s, t) \in \xrightarrow{\mu}$; in this case we call t a *μ-derivative of* s, or sometimes simply a *derivative of* s.

DEFINITION 2 (Bisimulation). A binary relation $\mathcal{R}$ on the states of an LTS is a *bisimulation* if whenever $s_1 \mathrel{\mathcal{R}} s_2$:

- for all s_1' with $s_1 \xrightarrow{\mu} s_1'$, there is s_2' such that $s_2 \xrightarrow{\mu} s_2'$ and $s_1' \mathrel{\mathcal{R}} s_2'$;
- the converse, on the transitions emanating from s_2 (i.e., for all s_2' with $s_2 \xrightarrow{\mu} s_2'$, there is s_1' such that $s_1 \xrightarrow{\mu} s_1'$ and $s_1' \mathrel{\mathcal{R}} s_2'$).

Bisimilarity, written $\sim$, is the union of all bisimulations; thus $s \sim t$ holds if there is a bisimulation $\mathcal{R}$ with $s \mathrel{\mathcal{R}} t$.

The definition of bisimilarity has a strong impredicative flavor, for bisimilarity itself is a bisimulation and is therefore part of the union from which it is defined. Also, the definition immediately suggests a proof technique: to demonstrate that

s_1 and s_2 are bisimilar, find a bisimulation relation containing the pair (s_1, s_2). This is the *bisimulation proof method.*

We will not discuss here the effectiveness of this proof method; the interested reader may consult concurrency textbooks in which bisimilarity is taken as the main behavioural equivalence for processes, such as [Milner, 1989]. We wish however to point out two features of the definition of bisimulation that make its proof method practically interesting:

- the *locality* of the checks on the states;
- the lack of a *hierarchy* on the pairs of the bisimulation.

The checks are local because we only look at the immediate transitions that emanate from the states. An example of a behavioural equality that is non-local is *trace equivalence* (two states are trace equivalent if they can perform the same *sequences* of transitions). It is non-local because computing a sequence of transitions starting from a state s may require examining other states, different from s.

There is no hierarchy on the pairs of a bisimulation in that no temporal order on the checks is required: all pairs are on a par. As a consequence, bisimilarity can be effectively used to reason about infinite or circular objects. This is in sharp contrast with inductive techniques, that require a hierarchy, and that therefore are best suited for reasoning about finite objects. For instance, here is a definition of equality that is local but inherently inductive:

> $s_1 = s_2$ if:
> for all s'_1 with $s_1 \xrightarrow{\mu} s'_1$, there is s'_2 such that $s_2 \xrightarrow{\mu} s'_2$ and $s'_1 = s'_2$;
> the converse, on the transitions from s_2.

This definition requires a hierarchy, as the checks on the pair (s_1, s_2) must follow those on derivative pairs such as (s'_1, s'_2). Hence the definition is ill-founded if the state space of the derivatives reachable from (s_1, s_2) is infinite or includes loops. (The definition would actually yield bisimilarity if we add that we refer to the largest relation "=" that satisfies the above clauses; what we wish to stress here is that, as it stands, the definition makes sense only if a hierarchy on the states exists in which the derivative pair is before the initial pair.)

We will also sometimes mention *simulations,* which are "half bisimulations".

DEFINITION 3 (Simulation). A binary relation $\mathcal{R}$ on the states of an LTS is a *simulation* if $s_1 \mathcal{R} s_2$ implies that for all s'_1 with $s_1 \xrightarrow{\mu} s'_1$ there is s'_2 such that $s_2 \xrightarrow{\mu} s'_2$ and $s'_1 \mathcal{R} s'_2$. *Similarity* is the union of all simulations.

We have presented the standard definitions of bisimulation and bisimilarity. A number of variants have been proposed and studied. For instance, on LTSs in which labels have a structure, which may be useful when processes may exchange values in communications; or on LTSs equipped with a special action to represent movements internal to processes, in which case one may wish to abstract from such action in the bisimulation game yielding the so-called *weak bisimulations* and *weak bisimilarity*. Examples of these kinds may be found, e.g., in [Milner, 1989; Sangiorgi and Walker, 2001; Aceto *et al.*, 2007; Baeten and Weijland, 1990; Sangiorgi, 2012]. Also, we do not discuss in this paper enhancements of the bisimulation proof method, intended to relieve the amount of work needed to prove bisimilarity results, such as *bisimulation up-to* techniques; see, e.g., [Pous and Sangiorgi, 2012].

3.2 Approximants of bisimilarity

We can approximate bisimilarity using the following inductively-defined relations and their intersection. Similar constructions can be given for similarity.

DEFINITION 4 (Stratification of bisimilarity). Let W be the set of states of an LTS. We set:

- $\sim_0 \overset{\text{def}}{=} W \times W$
- $s \sim_{n+1} t$, for $n \geq 0$, if
  1. for all s' with $s \xrightarrow{\mu} s'$, there is t' such that $t \xrightarrow{\mu} t'$ and $s' \sim_n t'$;
  2. the converse, i.e., for all t' with $t \xrightarrow{\mu} t'$, there is s' such that $s \xrightarrow{\mu} s'$ and $s' \sim_n t'$.
- $\sim_\omega \overset{\text{def}}{=} \bigcap_{n \geq 0} \sim_n$

In general, $\sim_\omega$ does not coincide with $\sim$, as the following example shows.

EXAMPLE 5. Suppose $a \in Act$, and let a^0 be a state with no transitions, a^ω a state whose only transition is

$$a^\omega \xrightarrow{a} a^\omega \ ,$$

and a^n, for $n \geq 1$, states with only transitions

$$a^n \xrightarrow{a} a^{n-1} \ .$$

Also, let s, t be states with transitions

$$s \xrightarrow{a} a^n \qquad \text{for all } n \geq 0$$

and

$$\begin{array}{ll} t \xrightarrow{a} a^n & \qquad \text{for all } n \geq 0 \\ t \xrightarrow{a} a^\omega & \end{array}$$

It is easy to prove, by induction on n, that, for all n, $s \sim_n t$, hence also $s \sim_\omega t$. However, it holds that $s \not\sim t$: the transition $t \xrightarrow{a} a^\omega$ can only be matched by s with one of the transitions $s \xrightarrow{a} a^n$. But, for all n, we have $a^\omega \not\sim a^n$, as only from the former state a sequence of $n+1$ transitions is possible.

In order to reach $\sim$, in general we need to replace the ω-iteration that defines $\sim_\omega$ with a transfinite iteration, using the ordinal numbers. However, the situation changes if the LTS is *finitely branching*, meaning that for all s the set $\{s' \,:\, s \xrightarrow{\mu} s',\ \text{for some } \mu\}$ is finite. (In Example 5, the LTS is not finitely branching.) Indeed, on finitely branching LTSs, relations $\sim$ and $\sim_\omega$ coincide. (The result also holds with the weaker condition of *image finiteness*, requiring that for all s and μ the set $\{s' \,:\, s \xrightarrow{\mu} s'\}$ is finite.)

3.3 Coinduction

Intuitively, a set A is defined *coinductively* if it is the *greatest* solution of an inequation of a certain form; then the *coinduction proof principle* just says that any set that is a solution of the same inequation *is contained* in A. Dually, a set A is defined *inductively* if it is the *least* solution of an inequation of a certain form, and the *induction principle* then says that any other set that is a solution to the same equation *contains* A. Familiar inductive definitions and proofs can be formalised in this way. To see how bisimulation and its proof method fit the coinductive schema, let $(W, Act, \{\xrightarrow{a} \,:\, a \in Act\})$ be an LTS, and consider the function $F_\sim : \wp(W \times W) \to \wp(W \times W)$ so defined:

$F_\sim(\mathcal{R})$ is the set of all pairs (s, t) such that:

1. for all s' with $s \xrightarrow{\mu} s'$, there is t' such that $t \xrightarrow{\mu} t'$ and $s' \,\mathcal{R}\, t'$.

2. for all t' with $t \xrightarrow{\mu} t'$, there is s' such that $s \xrightarrow{\mu} s'$ and $s' \,\mathcal{R}\, t'$.

We call $F_\sim$ the *functional associated to bisimulation*, for we have:

PROPOSITION 6.

1. $\sim$ is the greatest fixed point of $F_\sim$;

2. $\sim$ is the largest relation $\mathcal{R}$ such that $\mathcal{R} \subseteq F_\sim(\mathcal{R})$; thus $\mathcal{R} \subseteq \sim$ for all $\mathcal{R}$ with $\mathcal{R} \subseteq F_\sim(\mathcal{R})$.

Proposition 6 is a simple application of fixed-point theory, in particular the Fixed-Point Theorem, that we discuss below. We recall that a *complete lattice* is a partially ordered set with all joins (i.e., all its subsets have a supremum, also called least upper bound); this implies that there are also all meets (i.e., all subsets have an infimum, also called greatest lower bound). Using $\leq$ to indicate the partial order, a point x in the lattice is a *post-fixed point* of an endofunction F on the lattice if $x \leq F(x)$; it is a *pre-fixed point* if $F(x) \leq x$.

THEOREM 7 (Fixed-Point Theorem). *On a complete lattice, a monotone endofunction (i.e., a function from the lattice onto itself) has a complete lattice of fixed points. In particular the greatest fixed point of the function is the join of all its post-fixed points, and the least fixed point is the meet of all its pre-fixed points.*

We deduce from the theorem that:

- a monotone endofunction on a complete lattice has a greatest fixed point;
- for an endofunction F on a complete lattice the following rule is sound:

  $$(1) \quad \frac{F \text{ monotone} \qquad x \leq F(x)}{x \leq \text{gfp}\,(F)}$$

  where $\text{gfp}\,(F)$ indicates the greatest fixed point of F.

The existence of the greatest fixed point justifies coinductive definitions, while rule (1) expresses the coinduction proof principle, following the Fixed-Point Theorem.

Proposition 6 is a consequence of the Fixed-Point Theorem because the functional associated to bisimulation gives us precisely the clauses of a bisimulation, and is monotone on the complete lattice of the relations on W, in which the join is given by relational union, the meet by relational intersection, and the partial order by relation inclusion:

LEMMA 8.

- *$\mathcal{R}$ is a bisimulation iff $\mathcal{R} \subseteq F_\sim(\mathcal{R})$;*
- *$F_\sim$ is monotone (that is, if $\mathcal{R} \subseteq \mathcal{S}$ then also $F_\sim(\mathcal{R}) \subseteq F_\sim(\mathcal{S})$).*

For such functional $F_\sim$, (1) asserts that any bisimulation only relates pairs of bisimilar states. Example 5 shows that $\sim_\omega$ is not a fixed point for it.

Also the approximation of bisimilarity using the natural numbers, mentioned at the end of Section 3.2, can be seen as an application of fixed-point theory, in which one uses an extra hypothesis of the functional (cocontinuity, which is stonger than monotonicity) [Sangiorgi, 2012].

Complete lattices are "dualisable" structures: we can reverse the partial order and get another complete lattice. Thus the definitions and results above about joins, post-fixed points, greatest fixed points, cocontinuity have a dual in terms of meets, pre-fixed points, least fixed points, and continuity. As the results we gave justify coinductive definitions and the coinductive proof method, so the dual theorems can be used to justify familiar inductive definitions and inductive proofs for sets. More details can be found in [Sangiorgi, 2012; Sangiorgi and Rutten, 2012].

Another well-known example of application of coinduction is in definition and proofs involving *divergence*. Divergence represents an infinite computation and can be elegantly defined coinductively; then the coinduction proof method can be used to prove that specific computations diverge.

3.4 The origins of bisimulation in computer science

In computer science, the search for the origins of bisimulation takes us back to the algebraic theory of automata, well-established in the 1960s. A good reference is Ginzburg's book [1968]. Homomorphisms can be presented on different forms of automata. From the bisimulation perspective, the most interesting notions are formulated on *Mealy automata*. In these automata, there are no initial and final states; however, an output is produced whenever an input letter is consumed. Thus Mealy automata can be compared on the set of output strings produced. Formally, a Mealy automaton is a 5-tuple $(W, \Sigma, \Theta, \mathcal{T}, \mathcal{O})$ where

- W is the finite set of *states*;
- Σ is the finite set of *inputs*;
- Θ is a finite set of *outputs*;
- $\mathcal{T}$ is the *transition function*, that is a set of partial functions $\{\mathcal{T}_a \ : \ a \in \Sigma\}$ from W to W;
- $\mathcal{O}$ is the *output function*, that is, a set of partial functions $\{\mathcal{O}_a \ : \ a \in \Sigma\}$ from W to Θ.

The output string produced by a Mealy automaton is the *translation* of the input string with which the automaton was fed; of course the translation depends on the state in which the automaton is started. Since transition and output functions of a Mealy automaton are partial, not all input strings are consumed entirely.

Homomorphism is defined on Mealy automata following the standard notion in algebra, e.g., in group theory: a mapping that commutes with the operations defined on the objects of study. Below, if A is an automaton, then W^A is the set of states of A, and similarly for other symbols. As we deal with partial functions, it is convenient to view these as relations, and thereby use for them relational notations. Thus fg is the composition of the two function f and g where f is used first (that is, $(fg)(a) = g(f(a))$); for this, one requires that the codomain of f be included in the domain of g. Similarly, $f \subseteq g$ means that whenever f is defined then so is g, and they give the same result.

A *homomorphism* from the automaton A to the automaton B is a surjective function F from W^A to W^B such that for all $a \in \Sigma$:

1. $\mathcal{T}_a^A F \subseteq F \mathcal{T}_a^B$ (condition on the states); and
2. $\mathcal{O}_a^A \subseteq F \mathcal{O}_a^B$ (condition on the outputs).

(We assume here for simplicity that the input and output alphabets are the same, otherwise appropriate coercion functions would be needed.)

At the time (the 1960s), homomorphism and similar notions are all expressed in purely algebraic terms. Today we can make an operational reading of them, which for us is more enlightening. Writing $s \underset{b}{\overset{a}{\rightarrow}} t$ if the automaton, on state s and input a, produces the output b and evolves into the state t, and assuming for simplicity that $\mathcal{O}_a^A$ and $\mathcal{T}_a^A$ are undefined exactly on the same points, the two conditions above become:

- for all $s, s' \in W^A$, if $s \underset{b}{\overset{a}{\rightarrow}} s'$ then also $F(s) \underset{b}{\overset{a}{\rightarrow}} F(s')$.

Homomorphisms are used in that period to study a number of properties of automata. For instance, minimality of an automaton becomes the condition that the automaton has no proper homomorphic image. Homomorphisms are also used to compare automata. Mealy automata are compared using the notion of *covering* (written $\leq$): $A \leq B$ (read "automaton B covers automaton A") if B can do, statewise, at least all the translations that A does. That is, there is a total function ψ from the states of A to the states of B such that, for all states s of A, all translations performed by A when started in s can also be performed by B when started in $\psi(s)$. Note that B can however have states with a behaviour completely unrelated to that of any state of A; such states of B will not be the image of states of A. If both $A \leq B$ and $B \leq A$ hold, then the two automata are deemed *equivalent.*

Homomorphism implies covering, i.e., if there is a homomorphism from A to B then $A \leq B$. The converse result is (very much) false. The implication becomes stronger if one uses *weak homomorphisms.* These are obtained by relaxing the functional requirement of homomorphism into a relational one. Thus a weak homomorphism is a total relation $\mathcal{R}$ on $W^A \times W^B$ such that for all $a \in \Sigma$:

1. $\mathcal{R}^{-1}\mathcal{T}_a^A \subseteq \mathcal{T}_a^B\mathcal{R}^{-1}$ (condition on the states); and
2. $\mathcal{R}^{-1}\mathcal{O}_a^A \subseteq \mathcal{O}_a^B$ (condition on the outputs).

where relational composition, inverse, and inclusion are defined in the usual way for relations (and functions are taken as special forms of relations). In an operational interpretation as above, the conditions give:

- whenever $s \;\mathcal{R}\; t$ and $s \overset{a}{\underset{b}{\rightarrow}} s'$ hold in A, then there is t' such that $t \overset{a}{\underset{b}{\rightarrow}} t'$ holds in B and $s' \;\mathcal{R}\; t'$.

(On the correspondence between the algebraic and operational definitions, see also Remark 9 below.) Weak homomorphism reminds us of the notion of simulation for Labelled Transition Systems (LTSs). The former is however stronger, because the relation $\mathcal{R}$ is required to be *total.* (Also, in automata theory, the set of states and the sets of input and output symbols are required to be finite, but this difference is less relevant.)

REMARK 9. *To understand the relationship between weak homomorphisms and simulations, we can give an algebraic definition of simulation on LTSs, taking these to be triples* $(W, \Sigma, \{\mathcal{T}_a \,:\, a \in \Sigma\})$ *whose components have the same interpretation as for automata. A simulation between two LTSs A and B becomes a relation* $\mathcal{R}$ *on* $W^A \times W^B$ *such that, for all* $a \in \Sigma$, *condition (1) of weak homomorphism holds, i.e.*

- $\mathcal{R}^{-1}\mathcal{T}_a^A \subseteq \mathcal{T}_a^B\mathcal{R}^{-1}$

This is precisely the notion of simulation defined operationally in Definition 3. Indeed, given a state $t \in W^B$ *and a state* $s' \in W^A$, *we have* $t\,\mathcal{R}^{-1}\,\mathcal{T}_a^A\, s'$ *whenever*

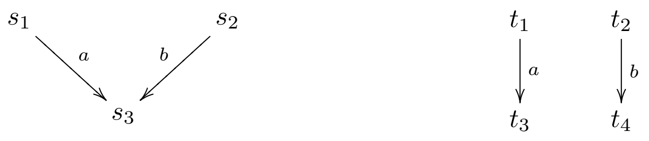

Figure 1. On homomorphisms and weak homomorphisms

there is $s \in W^A$ such that $s \xrightarrow{a} s'$. Then, requiring that the pair (t, s') is also in $\mathcal{T}_a^B \mathcal{R}^{-1}$ is the demand that there is t' such that $t \xrightarrow{a} t'$ and $s' \mathrel{\mathcal{R}} t'$.

As homomorphisms, so weak homomorphisms imply covering. The result for weak homomorphism is stronger as the homomorphisms are strictly included in the weak homomorphisms. An example of the strictness is given in Figure 1, where the states s_i belong to an automaton and the states t_i to another one, there are two input letters a and b, and for simplicity we ignore the automata outputs. We cannot establish a homomorphism from the automaton on the left to the automaton on the right, since a homomorphism must be surjective. Even leaving the surjective condition aside, a homomorphism cannot be established because the functional requirement prevents us from relating s_3 with both t_3 and t_4. By contrast, a weak homomorphism exists, relating s_1 with t_1, s_2 with t_2, and s_3 with both t_3 and t_4.

As weak homomorphisms are total relations, however, covering does not imply weak homomorphism. Indeed we are not aware, in the literature of that time, of characterisations of covering, or equivalence, in terms of notions akin to homomorphism. Such characterisations would have taken us closer to the idea of bisimulation.

In conclusion: in the algebraic presentation of automata in the 1960s we find concepts that remind us of bisimulation, or better, simulation. However there are noticeable differences, as we have outlined above. But the most important difference is due to the fact that the objects studied are deterministic. To see how significant this is, consider the operational reading of weak homomorphism, namely "whenever $s \mathrel{\mathcal{R}} t$ ... then there is t' such that....". As automata are deterministic, the existential quantifier in front of t' does not play a role. Thus the alternation of universal and existential quantifiers—a central aspect of the definitions of bisimulation and simulation—does not really show up in the setting of deterministic automata.

Robin Milner Decisive progress towards bisimulation is made by Robin Milner in the 1970s. Milner transplants the idea of weak homomorphism into the study of the behaviour of programs in a series of papers in the early 1970s ([Milner, 1970; Milner, 1971b; Milner, 1971a], with [Milner, 1971a] being a synthesis of the previous two). He studies programs that are sequential, imperative, and that may not terminate. He works on the comparisons among such programs. The aim is to develop techniques for proving the correctness of programs, and for abstracting from irrelevant details so that it is clear when two programs are realisations of the same algorithm. In short, the objective is to understand when and why two programs can be considered "intensionally" equivalent.

To this end, Milner proposes — appropriately adapting it to his setting — the algebraic notion of weak homomorphism that we have described in Section 3.4. He renames weak homomorphism as *simulation*, a term that better conveys the idea of the application in mind. Although the definition of simulation is still algebraic, Milner now clearly spells out its operational meaning. But perhaps the most important contribution in his papers is the proof technique associated to simulation that he strongly advocates. This techniques amounts to exhibiting the set of pairs of related states, and then checking the simulation clauses on each pair. The strength of the technique is precisely the *locality* of the checks that have to be made, in the sense that we only look at the immediate transitions that emanate from the states, as commented in Section 3.1. The technique is proposed to prove not only results of simulation, but also results of input/output correctness for programs, as a simulation between programs implies appropriate relationships on their inputs and outputs. Besides the algebraic theory of automata, other earlier works that have been influential for Milner are those on program correctness, notably Floyd [Floyd, 1967], Manna [Manna, 1969], and Landin [Landin, 1969], who pioneers the algebraic approach to programs.

Formally, however, Milner's simulation remains the same as weak homomorphism and as such it is not today's simulation. Programs for Milner are deterministic, with a total transition function, and these hypotheses are essential. Non-deterministic and concurrent programs or, more generally, programs whose computations are described by trees rather than sequences, are mentioned in the conclusions for future work. It is quite possible that if this challenge had been quickly taken up, then today's notion of simulation (or even bisimulation) would have been discovered much earlier.

Milner himself, later in the 1970s, does study concurrency very intensively, but under a very different perspective: he abandons the view of parallel programs as objects with an input/output behaviour akin to functions, in favor of the view of parallel programs as *interactive* objects. This leads Milner to develop a new theory of processes and a calculus — the Calculus of Communicating Systems, CCS — in which the notion of behavioural equivalence between processes is fundamental. Milner however keeps, from his earlier works, the idea of "locality" — an equivalence should be based on outcomes that are local to states.

The behavioural equivalence that Milner puts forward, and that is prominent in the first book on CCS [Milner, 1980], is inductively defined. It is the stratification of bisimilarity, $\sim_\omega$, that we discuss in Section 3.2. Technically, in contrast with weak homomorphisms, $\sim_\omega$ has also the reverse implication (on the transitions of the second components of the pairs in the relation), and can be used on non-deterministic structures. The addition of a reverse implication was not obvious.

For instance, a natural alternative would have been to maintain an asymmetric basic definition, possibly refine it, and then take the induced equivalence closure to obtain a symmetric relation (if needed). Indeed, among the main behavioural equivalences in concurrency — there are several of them, see [Glabbeek, 1993; Glabbeek, 1990b] — bisimilarity is the only one that is not naturally obtained as the equivalence-closure of a preorder.

With Milner's advances, the notion of bisimulation is almost there: what was left was to turn an inductive definition into a coinductive one. This will be David Park's contribution.

It is worth pointing out that, towards the end of the 1970s, homomorphism-like notions appear in other attempts at establishing "simulations", or even "equivalences", between concurrent models — usually variants of Petri Nets. Good examples are John S. Gourlay, William C. Rounds, and Richard Statman [Gourlay *et al.*, 1979] and Kurt Jensen [Jensen, 1980], which develop previous work by Daniel Brand [Brand, 1978] and Y. S. Kwong [Kwong, 1977]. Gourlay, Rounds, and Statman's homomorphisms (called *contraction*) relate an abstract system with a more concrete realisation of it — in other words, a specification with an implementation. Jensen's proposal (called *simulation*), which is essentially the same as Kwong's *strict reduction* [Kwong, 1977], is used to compare the expressiveness of different classes of Petri Nets. The homomorphisms in both papers are stronger than today's simulation or bisimulation; for instance they are functions rather than relations. Interestingly, in both cases there are forms of "reverse implications" on the correspondences between the transitions of related states. Thus these homomorphisms, but especially those in [Gourlay *et al.*, 1979], remind us of bisimulation, at least in the intuition. In [Gourlay *et al.*, 1979] and [Jensen, 1980], as well as other similar works of that period, the homomorphisms are put forward because they represent conditions that are sufficient to preserve certain important properties (such as Church-Rosser and deadlock freedom). In contrast with Milner, little emphasis is given to the proof technique based on local checks that they support. For instance the definitions of the homomorphisms impose correspondence on *sequences* of actions from related states.

David Park In 1980, Milner returns to Edinburgh after a six-month appointment at Aarhus University, and completes his first book on CCS. Towards the end of that year, David Park begins a sabbatical in Edinburgh, and stays at the top floor of Milner's house.

Park is one of the leading experts in fixed-point theory at the time. He makes the final step in the discovery of bisimulation precisely guided by fixed-point theory. Park notices that the inductive notion of equivalence that Milner is using for his equivalence on CCS processes is based on a monotone functional over a complete lattice. And by adapting an example by Milner, he sees that Milner's equivalence ($\sim_\omega$) is not a fixed point for the functional, and that therefore the functional is not cocontinuous. He then defines bisimilarity as the greatest fixed point of the functional, and derives the bisimulation proof method from the theory of greatest fixed points. Further, Park knows that, to obtain the greatest fixed point of the functional in an inductive way, the ordinals and transfinite induction, rather then the naturals and standard induction, are needed ([Sangiorgi, 2012]). Milner immediately and enthusiastically adopts Park's proposal.

Milner knew that $\sim_\omega$ is not invariant under transitions. Indeed he is not so much struck by the difference between $\sim_\omega$ and bisimilarity as behavioural equivalences, as the processes exhibiting such differences can be considered rather artificial. What excites him is the coinductive proof technique for bisimilarity. Both bisimilarity and $\sim_\omega$ are rooted in the idea of locality, but the coinductive method of bisimilarity further facilitates proofs. In the years to come Milner makes bisimulation popular and the cornerstone of the theory of CCS [Milner, 1989].

In computer science, the standard reference for bisimulation and the bisimulation proof method is Park's paper "Concurrency on Automata and Infinite Sequences" [Park, 1981a] (one of the most quoted papers in concurrency). However, Park's discovery is only partially reported in [Park, 1981a], whose main topic is a different one, namely omega-regular languages (extensions of regular languages to infinite sequences) and operators for fair concurrency. Bisimulation appears at the end, as a secondary contribution, as a proof technique for trace equivalence on automata. Bisimulation is first given on finite automata, but only as a way of introducing the concept on the Büchi-like automata investigated in the paper. Here, bisimulation has additional clauses that make it non-transitive and different from the definition of bisimulation we know today. Further, bisimilarity and the coinduction proof method are not mentioned in the paper.

Indeed, Park never writes a paper to report on his findings about bisimulation. It is possible that this does not appear to him a contribution important enough to warrant a paper: he considers bisimulation a variant of the earlier notion of simulation by Milner [Milner, 1970; Milner, 1971a]; and it is not in Park's style to write many papers. A good account of Park's discovery of bisimulation and bisimilarity are the summary and the slides of his talk at the 1981 Workshop on the Semantics of Programming Languages [Park, 1981b].

3.5 Discussion

It remains puzzling why bisimulation has been discovered so late in computer science. For instance, in the 1960s weak homomorphism is well-known in automata theory and, as discussed in Section 3.4, this notion is not that far from simulation. Another emblematic example, again from automata theory, is given by the algorithm for minimisation of deterministic automata, already known in the 1950s [Huffman, 1954; Moore, 1956] (also related to this is the Myhill-Nerode theorem [Nerode, 1958]). The aim of the algorithm is to find an automaton equivalent to a given one but minimal in the number of states. The algorithm proceeds by progressively constructing a relation $\mathcal{S}$ with all pairs of non-equivalent states. It roughly goes as follows. First step (a) below is applied, to initialise $\mathcal{S}$; then step (b), where new pairs are added to $\mathcal{S}$, is iterated until a fixed point is reached, i.e., no further pairs can be added.

a. For all states s, t, if s final and t is not, or vice versa, then $s \, \mathcal{S} \, t$.

b. For all states s, t such that $\neg(s \; \mathcal{S} \; t)$: if there is a such that $\mathcal{T}_a(s) \; \mathcal{S} \; \mathcal{T}_a(t)$ then $s \, \mathcal{S} \, t$.

The final relation gives all pairs of non-equivalent states. Then its complement, say $\overline{\mathcal{S}}$, gives the equivalent states. In the minimal automaton, the states in the same equivalence class for $\overline{\mathcal{S}}$ are collapsed into a single state.

The algorithm strongly reminds us of the partition refinement algorithms for computing bisimilarity and for minimisation modulo bisimilarity, discussed in [Aceto *et al.*, 2012]. Indeed, the complement relation $\overline{\mathcal{S}}$ that one wants to find has a natural coinductive definition, as a form of bisimilarity, namely the largest relation $\mathcal{R}$ such that

1. if $s \, \mathcal{R} \, t$ then either both s and t are final or neither is;

2. for each a, if $s \, \mathcal{R} \, t$ then $\mathcal{T}_a(s) \; \mathcal{R} \; \mathcal{T}_a(t)$.

Further, any relation $\mathcal{R}$ that satisfies the conditions (1) and (2) — that is, any bisimulation — only relates pairs of equivalent states and can therefore be used to determine equivalence of specific states.

The above definitions and algorithm are for deterministic automata. Bisimulation would have been interesting also on non-deterministic automata. Although on such automata bisimilarity does not coincide with trace equivalence — the standard equality on automata — at least bisimilarity implies trace equivalence and the bisimilarity-checking problem has a better complexity (P-complete [Alvarez *et al.*, 1991; Balcázar *et al.*, 1992], rather than PSPACE-complete [Meyer and Stockmeyer, 1972; Kanellakis and Smolka, 1990]).

Lumpability in probability theory An old concept in probability theory that today may be viewed as somehow reminiscent of bisimulation is Kemeny and Snell's *lumpability* [Kemeny and Snell, 1960]. A lumping equivalence is a partition of the states of a continuous-time Markov chain. The partition must satisfy certain conditions on probabilities guaranteeing that related states of the partition can be collapsed (i.e., “lumped”) into a single state. These conditions, having to do with sums of probabilities, are rather different from the standard one of bisimulation. (Kemeny and Snell's lumpability roughly corresponds to what today is called bisimulation for continuous-time Markov chains in the special case where there is only one label for transitions.)

The first coinductive definition of behavioural equivalence, as a form of bisimilarity, that takes probabilities into account appears much later, put forward by Larsen and Skou [Larsen and Skou, 1991]. This paper is the initiator of a vast body of work on coinductive methods for probabilistic systems in computer science.

Larsen and Skou were not influenced by lumpability. The link with lumpability was in fact noticed much later [Buchholz, 1994].

In conclusion: in retrospective we can see that Kemeny and Snell's lumpability corresponds to a very special form of bisimulation (continuous-time Markov chains, only one label). However, Kemeny and Snell's lumpability has not contributed to the discovery of coinductive concepts such as bisimulation and bisimilarity.

Coinduction In computer science, the discovery of bisimulation and bisimilarity led to the formulation of the more general principles of coinduction. For this, an important paper has been Milner and Tofte's [Milner and Tofte, 1991], who use coinduction to prove the soundness of a type system, and describe coinduction to explain the analogy between the technique for types in their paper and the bisimulation techniques. The main objective of the paper is indeed to advocate the proof technique and to suggest the name coinduction for it.

Coinduction has to do with greatest-fixed points, and certainly these had already appeared in computer science. For instance, David Park, throughout the 1970s, works intensively on fairness issues for programs that may contain constructs for parallelism and that may not terminate. The fixed-point techniques he uses are rather sophisticated, involving alternation of least and greatest fixed points. Park discusses his findings in several public presentations. A late overview paper is [Park, 1979]; we already pointed out, talking of Park, that he did not publish much. Willem-Paul de Roever [de Roever, 1977] strongly advocates the coinduction principle as a proof technique (he calls it "greatest fixed point induction"). De Roever uses the technique to reason about divergence, bringing up the duality between this technique and inductive techniques that had been proposed previously to reason on programs. Coinduction and greatest fixed points are implicit in a number of earlier works in the 1960s and 1970s. Important examples, with a huge literature, are the works on unification, for instance on structural equivalence of graphs, and the works on invariance properties of programs. Fixed points are also central in *stream processing* systems (including *data flow* systems). The introduction of streams in computer science is usually attributed to Peter Landin, in the early 1960s (see [Landin, 1965a; Landin, 1965b] where Landin discusses the semantics of Algol 60 as a mapping into a language based on the λ-calculus and Landin's SECD machine [Landin, 1964], and historical remarks in [Burge, 1975]). However, fixed points are explicitly used to describe stream computations only after Scott's theory of domain, with the work of Gilles Kahn [1974].

3.6 The origins of bisimulation in Modal Logics

Philosophical Logic studies and applies logical techniques to problems of interest to philosophers, somewhat similarly to what Mathematical Logic does for problems that interest mathematicians. Of course, the problems do not only concern philosophers or mathematicians; for instance nowadays both philosophical and mathematical logics have deep and important connections with computer science.

Strictly speaking, in philosophical logic a modal logic is any logic that uses *modalities*. A modality is an operator used to qualify the truth of a statement, that is, it creates a new statement that makes an assertion about the truth of the original statement.

Labelled Transition Systems (LTSs) with a valuation, also called *Kripke models*, are the standard models for modal logics; these are like the LTSs of Definition 1 except that each state is associated to a set of proposition letters. For the discussion below we use the following logic:

$$\phi \overset{\text{def}}{=} p \;\Big|\; \neg\phi \;\Big|\; \phi_1 \wedge \phi_2 \;\Big|\; \langle\mu\rangle\phi \;\Big|\; \bot$$

where p is a proposition letter. Formula $\langle\mu\rangle\phi$ holds at a state t if ϕ holds in at least one of the μ-derivatives of t; and p holds at t if p is among the letters assigned to t; the interpretation of the other operators is the standard one of propositional logic.

Today, some of the most interesting results on the expressiveness of modal logics rely on the notion of bisimulation. Bisimulation is indeed discovered in modal logic when researchers begin to investigate seriously issues of expressiveness for the logics, in the 1970s. For this, important questions tackled are: When is the truth of a formula preserved when the model changes? Or, even better, under which model constructions are modal formulas invariant? Which properties of models can modal logics express? (When moving from a model $\mathcal{M}$ to another model $\mathcal{N}$, preserving a property means that if the property holds in $\mathcal{M}$ then it holds also when one moves to $\mathcal{N}$; the property being invariant means that also the converse is true, that is, the property holds in $\mathcal{M}$ iff it holds when one moves to $\mathcal{N}$.)

To investigate such questions, it is natural to start from the most basic structure-preserving construction, that of *homomorphism*. A homomorphism from a model $\mathcal{M}$ to a model $\mathcal{N}$ is a function F from the points of $\mathcal{M}$ to the points of $\mathcal{N}$ such that

- whenever a proposition letter holds at a point s of $\mathcal{M}$ then the same letter also holds at $F(s)$ in $\mathcal{N}$;
- whenever there is a μ-transition between two points s, s' in $\mathcal{M}$ then there is also a μ-transition between $F(s)$ and $F(s')$ in $\mathcal{N}$.

Thus, contrasting homomorphism with bisimulation, we note that

(i) homomorphism is a functional, rather than relational, concept;

(ii) in the definition of homomorphism there is no back condition; i.e., the reverse implication, from transitions in $\mathcal{N}$ to those in $\mathcal{M}$, is missing.

Homomorphisms are too weak to respect the truth of modal formulas. That is, a homomorphism H from a model $\mathcal{M}$ to a model $\mathcal{N}$ does not guarantee that if a

$$\mathcal{M} \; = \begin{tikzcd} P \arrow[d] \\ Q \end{tikzcd} \qquad\qquad \mathcal{N} \; = \begin{tikzcd} & R \arrow[dl] \arrow[dr] & \\ S & & T \end{tikzcd}$$

Figure 2. On p-morphisms and p-relations

formula holds at a point t of $\mathcal{M}$ then the same formula also holds at $H(t)$ in $\mathcal{N}$. For instance, consider a model M with just one point and no transitions, and a model N with two points and μ-transitions between them. A homomorphism can send the point of M onto any of the points of N. The formula $\neg\langle\mu\rangle\neg\bot$, however, which holds at points that have no transitions, will be true in M, and false in N.

The culprit for the failure of homomorphisms is the lack of a back condition. Krister Segerberg added a back condition in his famous dissertation [Segerberg, 1971], as the requirement of *p-morphisms*:

- if a propositional letter holds at $F(s)$ in $\mathcal{N}$ then it also holds at s in $\mathcal{M}$; and if in $\mathcal{N}$ there is a transition $F(s) \xrightarrow{\mu} t$, for some point t, then in $\mathcal{M}$ there exists a point s' such that $s \xrightarrow{\mu} s'$ and $t = F(s')$.

Segerberg starts the study of morphisms between models of modal logics that preserve the truth of formulas in [Segerberg, 1968]. The p-morphisms can be regarded as the natural notion of homomorphism in LTSs or Kripke models.

Still, p-morphisms do not capture all situations of invariance. That is, there can be states s of a model $\mathcal{M}$ and t of a model $\mathcal{N}$ that satisfy exactly the same modal formulas and yet there is no p-morphisms that takes s into t or vice versa.

The next step is made by Johan van Benthem in his PhD thesis [Benthem, 1976] (the book [Benthem, 1983] is based on the thesis), who generalises the directional relationship between models in a p-morphism (the fact that a p-morphism is a function) to a symmetric one. This leads to the notion of bisimulation, which van Benthem calls *p-relation.* (Later [Benthem, 1984] he renames p-relations as *zigzag relations.*) On Kripke models, a p-relation between models $\mathcal{M}$ and $\mathcal{N}$ is a total relation $\mathcal{S}$ on the states of the models (the domain of $\mathcal{S}$ are the states of $\mathcal{M}$ and the codomain the states of $\mathcal{N}$) such that whenever $s \; \mathcal{S} \; t$ then: a propositional letter holds at s iff it holds at t; for all s' with $s \xrightarrow{\mu} s'$ in $\mathcal{M}$ there is t' such that $t \xrightarrow{\mu} t'$ in $\mathcal{N}$ and $s' \; \mathcal{S} \; t'$; the converse of the previous condition, on the transitions from t.

To appreciate the difference between p-morphisms and p-relations, consider the models in Figure 2 (where the letters are used to name the states, they do not represent proposition letters — there are no proposition letters, in fact). There is no p-morphisms from $\mathcal{M}$ to $\mathcal{N}$: the image of Q must be either S or T; in any case, there is always a transition from R that P cannot match. We can however establish a p-relation on the models, relating P with R, and Q with both S and T. (There is a p-morphism in the opposite direction, from $\mathcal{N}$ to $\mathcal{M}$; but the example could be developed a bit so that there is no p-morphisms in either direction.)

Van Benthem defines p-relations while working on *Correspondence Theory*, precisely the relationship between modal and classical logics. Van Benthem's objective is to characterise the fragment of first-order logic that "corresponds" to modal logic — an important way of measuring expressiveness. He gives a sharp answer to the problem, via a theorem that is today called "van Benthem Characterisation Theorem". In today's terminology, van Benthem's theorem says that a first-order formula A containing one free variable is equivalent to a modal formula iff A is invariant for bisimulations. That is, modal logic is the fragment of first-order logic whose formulas have one free variable and are invariant for bisimulation. We refer to [Stirling, 2012] for discussions on this theorem.

After van Benthem's theorem, bisimulation has been used extensively in modal logic, for instance, to analyze the expressive power of various dialects of modal logics, to understand which properties of models can be expressed in modal logics, and to define operations on models that preserve the validity of modal formulas.

3.7 The origins of bisimulation in set theory

In Mathematics, bisimulation and coinduction have been introduced in the study of the foundations of theories of non-well-founded sets. Non-well-founded sets are, intuitively, sets that are allowed to contain themselves; they are 'infinite in depth'. More precisely, the membership relation on sets may give rise to infinite descending sequences

$$\dots A_n \in A_{n-1} \in \dots \in A_1 \in A_0 \ .$$

For instance, a set Ω which satisfies the equation $\Omega = \{\Omega\}$ is circular and as such non-well-founded. A set can also be non-well-founded without being circular; this can happen if there is an infinite membership chain through a sequence of sets that are all different from each other. Bisimulation was introduced as a tool for defining the meaning of equality on non-well-founded sets; in other words, for defining what it means for two infinite sets to have 'the same' internal structure. In model theory, this issue is technically called *extensionality*: it guarantees that equal objects cannot be distinguished within the given model. When the structure of the objects, or the way in which the objects are supposed to be used, are non-trivial, the 'correct' definition of extensionality may be non-obvious. This is certainly the case for non-well-founded sets, as they are objects with an infinite depth. Bisimulation was derived from the notion of isomorphism (and homomorphism), intuitively with the objective of obtaining an equality relation that is coarser than isomorphism but still with the guarantee that related sets have 'the same' internal structure.

In ordinary (i.e., *well-founded*) sets, the notion of equality is expressed by Zermelo's *extensionality axiom*: two sets are equal if they have exactly the same elements. In other words, a set is precisely determined by its elements. This is uncontroversial because it is very intuitive and because it naturally allows us to reason on equality proceeding by (transfinite) induction on the membership relation. For instance, we can thus establish that the relation of equality is unique. Non-well-founded sets, by contrast, may be infinite in depth, and therefore inductive arguments may not be applicable. For instance, consider the sets A and B defined via the equations $A = \{B\}$ and $B = \{A\}$. If we try to establish that they are equal via the extensionality axiom we end up with a tautology ("A and B are equal iff A and B are equal") that takes us nowhere. Indeed, to reason on non-well-founded sets we need *coinductive* techniques, bisimulation *in primis*.

A major motivation for the study of non-well-founded sets in Mathematics has been the need of giving semantics to processes, following Robin Milner's work in concurrency theory. Similarly, the development of Final Semantics [Aczel, 1988; Rutten and Turi, 1994; Rutten and Jacobs, 2012], an area of mathematics based on coalgebras and category theory and used in the semantics of programming languages, has been largely motivated by the interest in bisimulation. As a subject, Final Semantics is today well developed, and gives us a rich and deep perspective on the meaning of coinduction and its duality with induction.

Bisimulation is first introduced in set theory by Forti and Honsell [Forti and Honsell, 1983] (a similar notion, independently, was used by Hinnion, [Hinnion, 1980; Hinnion, 1981]), around the beginning of the 1980s). It is recognised and becomes important with the work of Aczel and Barwise, see [Aczel, 1988; Barwise and Moss, 1996]. We briefly describe below the most important works, those by Forti and Honsell, and by Aczel.

In [Forti and Honsell, 1983], Forti and Honsell study a number of anti-foundation axioms, including axioms that had already appeared in the literature (such as Scott's [Scott, 1960]), and a new one, called X_1, that gives the strongest extensionality properties, in the sense that it equates more sets (we discuss X_1 below in the section, together with Aczel's version of it). The main objective of the paper is to compare the axioms, and define models that prove their consistency. Bisimulations and similar relations are used in the constructions to guarantee the extensionality of the models.

Forti and Honsell use, in their formulation of bisimulation, functions $f : A \mapsto \wp(A)$ from a set A to its powerset $\wp(A)$. Bisimulations are called *f-conservative* relations and are defined along the lines of the fixed-point interpretation of bisimulation in Section 3.3. We can make a state-transition interpretation of their definitions, for a comparison with today's definition (Definition 2). If f is the function from A to $\wp(A)$ in question, then we can think of A as the set of the possible states, and of f itself as the (unlabeled) transition function; so that $f(x)$ indicates the set of possible "next states" for x. Forti and Honsell define the fixed point behaviour of f on the relations on A, via the functional F defined as follows[1]. If $\mathcal{R}$ is a relation on A, and $s, t \in A$, then $(s, t) \in F(\mathcal{R})$ if:

- for all $s' \in f(s)$ there is $t' \in f(t)$ such that $s' \mathrel{\mathcal{R}} t'$;

[1]We use a notation different from Forti and Honsell here.

$$\begin{tikzcd}
 & 2 \arrow[dl] \arrow[dr] & & & & c \arrow[dl] \arrow[dr, bend left] & \\
0 & & 1 \arrow[ll] & & a & & b \arrow[ul, bend left]
\end{tikzcd}$$

Figure 3. Sets as graphs

- the converse, i.e. for all $t' \in f(t)$ there is $s' \in f(s)$ such that $s' \mathcal{R} t'$.

A reflexive and symmetric relation $\mathcal{R}$ is f-conservative if $\mathcal{R} \subseteq F(\mathcal{R})$; it is f-admissible if it is a fixed point of F, i.e., $\mathcal{R} = F(\mathcal{R})$. The authors note that F is monotone over a complete lattice, hence it has a greatest fixed point (the largest f-admissible relation). They also prove that such greatest fixed point can be obtained as the union over all f-conservative relations (the coinduction proof principle), and also, inductively, as the limit of a sequence of decreasing relations over the ordinals that starts with the universal relation $A \times A$. The main difference between f-conservative relations and today's bisimulations is that the former are required to be reflexive and symmetric.

However, while the bisimulation proof method is introduced, as derived from the theory of fixed points, it remains rather hidden in Forti and Honsell's works, whose main goal is to prove the consistency of anti-foundation axioms. For this the main technique uses the f-admissible relations.

Aczel reformulates Forti and Honsell's anti-foundation axiom X_1. In Forti and Honsell [1983], the axiom says that from every relational structure there is a unique homomorphism onto a transitive set (a relational structure is a set equipped with a relation on its elements; a set A is transitive if each set B that is an element of A has the property that all the elements of B also belong to A; that is, all composite elements of A are also subsets of A). Aczel calls the axiom AFA and expresses it with the help of graph theory, in terms of graphs whose nodes are decorated with sets. For this, sets are thought of as (pointed) graphs, where the nodes represent sets, the edges represent the converse membership relation (e.g., an edge from a node x to a node y indicates that the set represented by y is a member of the set represented by x), and the root of the graph indicates the starting point, that is, the node that represents the set under consideration. For instance, the sets $\{\emptyset, \{\emptyset\}\}$ and $D = \{\emptyset, \{D\}\}$ naturally corresponds to the graphs of Figure 3 (where for convenience nodes are named) with nodes 2 and c being the roots. The graphs for the well-founded sets are those without infinite paths or cycles, such as the graph on the left in Figure 3. AFA essentially states that each graph represents a unique set. This is formalised via the notion of *decoration.* A decoration for a graph is an assignment of sets to nodes that respects the structure of the edges; that is, the set assigned to a node is equal to the set of the sets assigned to the children of the node. For instance, the decoration for the graph on the left of Figure 3 assigns $\emptyset$ to node 0, $\{\emptyset\}$ to node 1, and $\{\emptyset, \{\emptyset\}\}$ to node 2, whereas that for the graph on the right assigns $\emptyset$ to a, $\{D\}$ to b, and $\{\emptyset, \{D\}\}$ to c. Axiom AFA stipulates that *every graph has a unique decoration.* (In Aczel, the graph plays the role of the relational structure in Forti and Honsell, and the decoration the role of the homomorphism into a transitive set.) In this, there are two important facts: the existence of the decoration, and its uniqueness. The former tells us that the non-well-founded sets we need do exist. The latter tell us what is equality for them. Thus two sets are equal if they can be assigned to the same node of a graph. For instance the sets Ω, A and B mentioned at the beginning of this section are equal because the graph

$$\begin{tikzcd}\bullet \arrow[r, bend left] & \bullet \arrow[l, bend left]\end{tikzcd}$$

has a decoration in which both nodes receive Ω, and another decoration in which the node on the left receives A and that on the right B. Bisimulation comes out when one tries to extract the meaning of equality. A bisimulation relates sets A and B such that

- for all $A_1 \in A$ there is $B_1 \in B$ with A_1 and B_1 related; and the converse, for the elements of B_1.

Two sets are equal precisely if there is a bisimulation relating them. The bisimulation proof method can then be used to prove equalities between sets, for instance the equality between the sets A and B above.

Aczel formulates AFA towards end 1983; he does not publish it immediately having then discovered the earlier work of Forti and Honsell and the equivalence between AFA and X_1. Instead, he goes on developing the theory of non-well-founded sets, mostly through a series of lectures in Stanford between January and March '85, which leads to the book [Aczel, 1988]. Aczel shows how to use the bisimulation proof method to prove equalities between non-well-founded sets, and develops a theory of coinduction that sets the basis for the coalgebraic approach to semantics (Final Semantics).

Up to Aczel's book [Aczel, 1988], all the works on non-well-founded sets had remained outside the mainstream. This changes with Aczel, for two main reasons: the elegant theory that he develops, and the concrete motivations for studying non-well-founded sets that he brings up, namely mathematical foundations of processes, in this prompted by the work of Milner on CCS and his way of equating processes with an infinite behaviour via a bisimulation quotient.

4 PROCESS CALCULI

At this point we switch our attention from bisimulation and coinduction to process calculi. Process calculi start from a syntax, a language describing the objects of interest, elements of concurrent behavior and how they are put together. In this section the history of process calculi is traced back to the early seventies of the twentieth century, and developments since that time are sketched.

The word ‘process’ refers to discrete behavior of agents, as discussed in Section 2. The word ‘calculus’ refers to doing calculations with processes, in order to calculate a property of a process, or to prove that processes are equal. We sketch the state of research in the early seventies, and state which breakthroughs were needed in order for the theories to appear. We consider the development of CCS, CSP and ACP. In Section 4.7, we sketch the main developments since then.

The calculations are based on a basic set of *laws* that are established or postulated for processes. These laws are usually stated in the form of an *algebra*, using techniques and results from mathematical universal algebra (see e.g. [MacLane and Birkhoff, 1967]). To emphasize the algebraical basis, the term ‘process algebra’ is often used instead of ‘process calculus’. Strictly speaking, a process algebra only uses laws stated in the form of an algebra, while a process calculus can also use laws that use multiple sorts and binding variables, thus going outside the realm of universal algebra. A process calculus can start from a given syntax (set of operators) and try to find the laws concerning these operators that hold in a given semantical domain, while a process algebra can start from a given syntax and a set of laws or axioms concerning these operators, and next consider all the different semantical domains where these laws hold. Comparing process calculi that have different semantical domains works best by considering the set of laws that they have [Glabbeek, 1990a].

On the basis of the set of laws or axioms, we can calculate, perform equational reasoning. To compare, calculations with automata can be done by means of the algebra of regular expressions (see e.g. [Linz, 2001]).

Since a process calculus addresses interaction, agents acting in parallel, a process calculus will usually (but not necessarily) have a form of parallel composition as a basic operator.

To repeat, the study of process calculi is the study of the behavior of parallel or distributed systems based on a set of (algebraic) laws. It offers means to describe or *specify* such systems, and thus it has means to talk about parallel composition. Besides this, it can usually also talk about alternative composition (choice) and a form of sequential composition (sequencing). By means of calculation, we can do *verification*, i.e. we can establish that a system satisfies a certain property.

What are these basic laws of process algebra? We can list some, that can be called *structural* laws. We start out from a given set of atomic actions, and use the basic operators to compose these into more complicated processes. We use notations from [Baeten *et al.*, 2010], that unifies the literature on process calculi. As basic operators, we use $+$ denoting alternative composition, $\cdot$ denoting sequential composition and $\|$ denoting parallel composition. Usually, there are also neutral elements for some or all of these operators, but we do not consider these yet. Some basic laws are the following ($+$ binding weakest, $\cdot$ binding strongest).

- $x + y = y + x$ (commutativity of alternative composition)
- $x + (y + z) = (x + y) + z$ (associativity of alternative composition)
- $x + x = x$ (idempotence of alternative composition)

- $(x + y) \cdot z = x \cdot z + y \cdot z$ (right distributivity of $+$ over $\cdot$)
- $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ (associativity of sequential composition)
- $x \parallel y = y \parallel x$ (commutativity of parallel composition)
- $(x \parallel y) \parallel z = x \parallel (y \parallel z)$ (associativity of parallel composition)

These laws list some general properties of the operators involved. Note there is a law stating the right distributivity of $+$ over $\cdot$, but no law of left distributivity. Adding the left distributivity law leads to a so-called *linear time* theory. Usually, left distributivity is absent, and we speak of a *branching time* theory, where the moment of choice is relevant.

We can see that there is a law connecting alternative and sequential composition. In some cases, other connections are considered. On the other hand, we list no law connecting parallel composition to the other operators. It turns out such a connection is at the heart of process algebra, and it is the tool that makes calculation possible. In most process calculi, this law allows one to express parallel composition in terms of the other operators, and is called an expansion theorem. Process calculi with an expansion theorem are called *interleaving* process calculi, those without (such as a calculus of Petri nets) are called *partial order* or *true concurrency* calculi. For a discussion concerning this dichotomy, see [Baeten, 1993].

So, we can say that any mathematical structure with three binary operations satisfying these 7 laws is a process algebra. Most often, these structures are formulated in terms of automata-like models, namely the labeled transition systems of Definition 1. The notion of equivalence studied is usually not language equivalence. Prominent among the equivalences studied is bisimulation, as discussed in the previous section. Strictly speaking, the study of labeled transition systems, ways to define them and equivalences on them are not part of a process calculus. We can use the term *process theory* as a wider notion, that encompasses also semantical issues.

Below, we describe the history of process algebra from the early seventies to the early eighties, by focusing on the central people involved. By the early eighties, we can say process algebra is established as a separate area of research. Subsection 4.7 will consider the main extensions since the early eighties until the present time.

4.1 Bekič

One of the people studying the semantics of parallel programs in the early seventies was Hans Bekič. He was born in 1936, and died due to a mountain accident in 1982. In the period we are speaking of, he worked at the IBM lab in Vienna, Austria. The lab was well-known in the sixties and seventies for the work on the definition and semantics of programming languages, and Bekič played a part in this, working on the denotational semantics of ALGOL and PL/I. Growing out of his work on PL/I, the problem arose how to give a denotational semantics for parallel composition. Bekič tackled this problem in [Bekič, 1971]. This internal report, and indeed all the work of Bekič is made accessible to us through the work of Cliff Jones [Bekič, 1984]. On this book, we base the following remarks.

In [Bekič, 1971], Bekič addresses the semantics of what he calls "quasi-parallel execution of processes". From the introduction, we quote:

> Our plan to develop an *algebra of processes* may be viewed as a *high-level* approach: we are interested in how to compose complex processes from simpler (still arbitrarily complex) ones.

Bekič uses global variables, so a state ξ is a valuation of variables, and a program determines an action A, which gives, in a state (non-deterministically) either *null* iff it is an end-state, or an elementary step f, giving a new state $f\xi$ and rest-action A'. Further, there are $\sqcup$ and *cases* denoting alternative composition, ; denoting sequential composition, and // denoting (quasi-)parallel composition.

On page 183 in [Bekič, 1984], we see the following law for quasi-parallel composition:

$$\begin{array}{rll} (A//B)\xi & = & \\ (\text{cases} & & A\xi : \text{null} \to B\xi \\ & & \quad (f, A') \to f, (A'//B)) \\ \sqcup & & \\ (\text{cases} & & B\xi : \text{null} \to A\xi \\ & & \quad (g, B') \to g, (A//B')) \end{array}$$

and this is called the "unspecified merging" of the elementary steps of A and B. This is definitely a pre-cursor of what later would be called the expansion law of process calculi. It also makes explicit that Bekič has made the first paradigm shift: the next step in a merge is not determined, so we have abandoned the idea of a program as a function.

The book [Bekič, 1984] goes on with clarifications of [Bekič, 1971] from a lecture in Amsterdam in 1972. Here, Bekič states that an action is tree-like, behaves like a scheduler, so that for instance $f; (g \sqcup h)$ is not the same as $(f; g) \sqcup (f; h)$ for elementary steps f, g, h, another example of non-functional behavior. In a letter to Peter Lucas from 1975, Bekič is still struggling with his notion of an action, and writes:

> These actions still contain enough information so that the normal operations can be defined between them, but on the other hand little enough information to fulfil certain desirable equivalences, such as:
>
> $$a; 0 = a \quad a; (b; c) = (a; b); c \quad a//b = b//a$$
>
> etc.

In a lecture on the material in 1974 in Newcastle, Bekič has changed the notation of // to $\|$ and calls the operator parallel composition. In giving the equations, we even encounter a "left-parallel" operator, with laws, with the same meaning that Bergstra and Klop will later give to their left-merge operator [Bergstra and Klop, 1982].

Concluding, we can say that Bekič contributed a number of basic ingredients to the emergence of process algebra, but we see no coherent comprehensive theory yet.

4.2 CCS

The central person in the history of process calculi without a doubt is Robin Milner —we already mentioned his relevance for concurrency theory in Section 3.4, discussing bisimulation. A.J.R.G. Milner, who was born in 1934 and died in 2010, developed his process theory CCS (Calculus of Communicating Systems) over the years 1973 to 1980, culminating in the publication of the book [Milner, 1980] in 1980.

The oldest publications concerning the semantics of parallel composition are [Milner, 1973; Milner, 1975], formulated within the framework of denotational semantics, using so-called transducers. He considers the problems caused by non-terminating programs, with side effects, and non-determinism. He uses the operations $*$ for sequential composition, ? for alternative composition and $\|$ for parallel composition. He refers to [Bekič, 1971] as related work.

Next, chronologically, are the articles [Milner, 1979; Milne and Milner, 1979]. Here, Milner introduces *flow graphs*, with ports where a named port synchronizes with the port with its co-name. Operators are $|$ for parallel composition, restriction and relabeling. The symbol $\|$ is now reserved for restricted parallel composition. Structural laws are stated for these operators.

The following two papers are [Milner, 1978a; Milner, 1978b], putting in place most of CCS as we know it. The operators prefixing and alternative composition are added and provided with laws. Synchronization trees are used as a model. The prefix τ occurs as a *communication trace* (what remains of a synchronization of a name and a co-name). The paradigm of message passing is taken over from [Hoare, 1978]. Interleaving is introduced as the observation of a single observer of a communicating system, and the expansion law is stated. Sequential composition is not a basic operator, but a derived one, using communication, abstraction and restriction.

The paper [Hennessy and Milner, 1980], with Matthew Hennessy, formulates basic CCS, with observational equivalence and strong equivalence defined inductively. Also, so-called Hennessy-Milner logic is introduced, which provides a logical characterization of process equivalence. Next, the book [Milner, 1980] is the standard process calculus reference. Here we have for the first time in history a complete process calculus, with a set of equations and a semantical model. He presents the equational laws as truths about his chosen semantical domain, rather than considering the laws as primary, and investigating the range of models that they have.

We pointed out in Section 3.4 that an important contribution, realized just after the appearance of [Milner, 1980], is the formulation of bisimulation. This became a central notion in process theory subsequently. The book [Milner, 1980] was later updated in [Milner, 1989].

A related development is the birth of *structural operational semantics* in [Plotkin, 1981]. More can be read about this in the historical paper [Plotkin, 2004b].

To recap, CCS has the following syntax:

- A constant 0 that is the neutral element of alternative composition, the process that shows no behavior. It is the seed process from which other processes can be constructed using the operators.

- For each action a from a given set of actions A, the action prefix operator $a._$, that prefixes a given process with the action a: after execution of a, the process continues. This is a restricted form of sequential composition.

- Alternative composition $+$. It is important to note that the choice between the given alternatives is made by the execution of an action from one of them, thereby discarding the alternative, not before. As a consequence, there is the law $x + 0 = x$, as 0 is an alternative that cannot be chosen.

- Parallel composition $\mid$. In a parallel composition $x \mid y$, an action from x or from y can be executed, or they can jointly execute a communication action. The set of actions A is divided into a set of names and a set of co-names (for each name a there is a co-name $\bar{a}$). The joint execution of a name and its corresponding co-name results in the execution of the special communication action τ. The action prefix operator $\tau._$ has a special set of laws called the τ-laws, that allow one to eliminate the τ action in a number of cases. Thus, the parallel composition operator does two things at a time: it allows a communication, and hides the result of the communication in a number of cases (a form of abstraction).

- Recursion or fixed point construction. If P is a process expression possibly containing the variable x, then $\mu x.P$ is the smallest fixed point of P, the process showing the least behavior satisfying the equation $\mu x.P = P[\mu x.P/x]$, where the last construct is the process expression P with all occurrences of the variable x replaced by $\mu x.P$. The notions "smallest" and "least" refer to the fact that only behavior is included that can be inferred from the behavior of P and this equation. This construct is used to define processes that can execute an unrestricted number of actions, a so-called *reactive* process. In later work, Milner does not use binding of variables, but instead sees the fixed point as a new constant X, whose behavior is given by the recursive equation $X = P$.

- Restriction or encapsulation ∂_H, where H is a set of names and their corresponding co-names, will block execution of the actions in H. By blocking execution of the names and co-names, but always allowing τ, communication in a parallel composition can be enforced. CCS uses a different notation for this operator.

- Relabeling or renaming ρ_f, where f is a function on actions that preserves the co-name relation and does not rename τ. This operator is useful to obtain different instances of some generically defined process. CCS uses a different notation for this operator.

4.3 CSP

A very important contributor to the development of process calculi is Tony Hoare. C.A.R. Hoare, born in 1934, published the influential paper [Hoare, 1978] as a technical report in 1976. The important step is that he does away completely with global variables, and adopts the message passing paradigm of communication, thus realizing the second paradigm shift. The language CSP (Communicating Sequential Processes) described in [Hoare, 1978] has synchronous communication and is a guarded command language (based on [Dijkstra, 1975]). No model or semantics is provided. This paper inspired Milner to treat message passing in CCS in the same way.

A model for CSP was elaborated in [Hoare, 1980]. This is a model based on trace theory, i.e. on the sequences of actions a process can perform. Later on, it was found that this model was lacking, for instance because deadlock behavior is not preserved. For this reason, a new model based on failure pairs was presented in [Brookes *et al.*, 1984] for the language that was then called TCSP (*Theoretical CSP*). Later, TCSP was called CSP again. Some time later it was established that the failure model is the least discriminating model that preserves deadlock behavior (see e.g. [Glabbeek, 2001]). In the language, due to the presence of two alternative composition operators, it is possible to do without a silent step like τ altogether. The book [Hoare, 1985] gives a good overview of CSP.

Between CCS and CSP, there is some debate concerning the nature of alternative composition. Some say the + of CCS is difficult to understand ("the weather of Milner"), and CSP proposes to distinguish between internal and external non-determinism, using two separate operators. See also [Hennessy, 1988].

The syntax of CSP from [Hoare, 1985] constitutes:

- A constant called $STOP$ that acts like the 0 CCS, but also a constant called $SKIP$ (that we call 1) that is the neutral element of sequential composition. Thus, 0 stands for unsuccessful termination and 1 for successful termination. Both processes do not execute any action.

- Action prefix operators $a.\_$ as in CCS. There is no τ prefixing.

- CSP has *two* alternative composition operators, $\sqcap$ denoting non-deterministic or internal choice, and $\Box$ denoting external choice. The internal-choice operator denotes a non-deterministic choice that cannot be influenced by the environment (other processes in parallel) and can simply be defined in CCS terms as follows:

  $$x \sqcap y = \tau.x + \tau.y.$$

  The external choice operator is not so easily defined in terms of CCS (see [Glabbeek, 1986]). It denotes a choice that can be influenced by the environment. If the arguments of the operator have initial silent non-determinism, then these τ-steps can be executed without making a choice, and the choice will be made as soon as a visible (non-τ) action occurs.

  Because of the presence of two choice operators and a semantics that equates more processes than bisimilarity, all silent actions that might occur in a process expression can be removed [Bergstra *et al.*, 1987].

- There is action prefixing like in CCS, but also full sequential composition.

- The parallel composition operator of CSP allows interleaving but also synchronization on the same name, so that execution of an action a by both components results in a communication action again named a. This enables multi-way synchronization.

- Recursion is handled as in CCS.

- There is a *concealment* or abstraction operator that renames a set of actions into τ. These introduced τ can subsequently be removed from an expression.

4.4 Some Other Process Calculi

Around 1980, concurrency theory and in particular process theory is a vibrant field with a lot of activity world wide. We already mentioned research on Petri nets, that is an active area [Petri, 1980]. Another partial order process theory is given in [Mazurkiewicz, 1977]. Research on temporal logic has started, see e.g. [Pnueli, 1977].

Some other process calculi can be mentioned. We already remarked that Hoare investigated trace theory. More work was done in this direction, e.g. by Rem [Rem, 1983]. There is also the invariants calculus [Apt *et al.*, 1980].

Another process theory that should be mentioned is the metric approach by De Bakker and Zucker [Bakker and Zucker, 1982a; Bakker and Zucker, 1982b]. There is a notion of distance between processes: processes that do not differ in behavior before the nth step have a distance of at most 2^{-n}. This turns the domain of processes into a metric space, that can be completed, and solutions to guarded recursive equations (a type of well-behaved recursive equations) exist by application of Banach's fixed point theorem [Banach, 1922].

4.5 ACP

Jan Bergstra and Jan Willem Klop in 1982 started work on a question of De Bakker as to what can be said about solutions of unguarded recursive equations. As a result, they wrote the paper [Bergstra and Klop, 1982]. In this paper, the phrase "process algebra" is used for the first time. We quote:

> A *process algebra* over a set of atomic actions A is a structure $\mathcal{A} = \langle \mathsf{A}, +, \cdot, \llfloor, a_i (i \in I) \rangle$ where A is a set containing A, the a_i are constant symbols corresponding to the $a_i \in A$, and $+$ (*union*), $\cdot$ (*concatenation* or *composition*, left out in the axioms), $\llfloor$ (*left merge*) satisfy for all $x, y, z \in \mathsf{A}$ and $a \in A$ the following axioms:
>
> | | |
> |---|---|
> | PA1 | $x + y = y + x$ |
> | PA2 | $x + (y + z) = (x + y) + z$ |
> | PA3 | $x + x = x$ |
> | PA4 | $(xy)z = x(yz)$ |
> | PA5 | $(x + y)z = xz + yz$ |
> | PA6 | $(x + y) \llfloor z = x \llfloor z + y \llfloor z$ |
> | PA7 | $ax \llfloor y = a(x \llfloor y + y \llfloor x)$ |
> | PA8 | $a \llfloor y = ay$ |

This clearly establishes a process calculus in the framework of universal algebra. In the paper, process algebra was defined with alternative, sequential and parallel composition, but without communication. A model was established based on projective sequences (a process is given by a sequence of approximations by finite terms), and in this model, it is established that all recursive equations have a solution. In adapted form, this paper was later published as [Bergstra and Klop, 1992]. In [Bergstra and Klop, 1984b], this process algebra PA was extended with communication to yield the theory ACP (Algebra of Communicating Processes). The book [Baeten and Weijland, 1990] gives an overview of ACP.

The syntax of ACP comprises:

- A constant 0 denoting inaction, as in CCS and CSP (written δ).
- A set of actions A, each element denoting a constant in the syntax. Expressed in terms of prefixing and successful termination, each such constant can be denoted as $a.1$, execution of a followed by successful termination. This lumping together of two notions causes problems when the theory is extended with explicit timing, see [Baeten, 2003].
- Alternative composition $+$ as in CCS.
- Sequential composition $\cdot$ as in CSP.

- The set of actions A has given on it a partial binary commutative and associative communication function that tells when two actions can synchronize in a parallel composition, and what is the resulting communication action. ACP does not have a special silent action τ, each communication action is just a regular action. Parallel composition then has interleaving and communication. The finite axiomatization of parallel composition then uses an auxiliary operator left merge as shown above, and in addition another auxiliary operator called communication merge.
- Encapsulation ∂_H blocking a subset of actions H of A as in CCS.
- Recursion. A process constant X can be defined by means of a recursive equation $X = P$, where the constant may appear in the expression P. Also, a set of constants can be defined by a set of recursive equations, one for each constant.

4.6 CCS, CSP and ACP

Comparing the three most well-known process calculi CCS, CSP and ACP, we can say there is a considerable amount of work and applications realized in all three of them. In that sense, there seem to be no fundamental differences between the theories with respect to the range of applications. Historically, CCS was the first with a complete theory. Different from the other two, CSP has a least distinguishing equational theory. More than the other two, ACP emphasizes the algebraic aspect: there is an equational theory with a range of semantical models. Also, ACP has a more general communication scheme: in CCS, communication is combined with abstraction, in CSP, there is also a restricted communication scheme.

In ensuing years, other process calculi were developed. We can mention SCCS [Milner, 1983], CIRCAL [Milne, 1983], MEIJE [Austry and Boudol, 1984], the process calculus of Hennessy [Hennessy, 1988].

We see that over the years many process calculi have been developed, each making its own set of choices in the different possibilities. The reader may wonder whether this is something to be lamented. In the paper [Baeten *et al.*, 1991], it is argued that this is actually a good thing, as long as there is a good exchange of information between the different groups, as each different process calculus will have its own set of advantages and disadvantages. When a certain notion is used in two different process calculi with the same underlying intuition, but with a different set of equational laws, there are some who argue for the same notation, in order to show that we are really talking about the same thing, and others who argue for different notations, in order to emphasize the different semantical setting.

With the book [Baeten *et al.*, 2010], an integrated overview is presented of all features of CCS, CSP and ACP, based on an algebraic presentation, together with highlights of the main extensions since the 1980s. A good overview of developments is also provided by the impressive handbook [Bergstra *et al.*, 2001].

4.7 Developments

Theory

A nice overview of the most important theoretical results since the start of process calculi is the paper [Aceto, 2003]. Also, remaining open problems are stated there. For a process calculus based on partial order semantics, see [Best *et al.*, 2001]. There is a wealth of results concerning process calculi extended with some form of recursion, see e.g. the complete axiomatization of regular processes by Milner [Milner, 1984] or the overview on decidability in [Burkart *et al.*, 2001]. Also, there is a whole range of *expressiveness* results, some examples can be found in [Bergstra and Klop, 1984a].

Tooling

Over the years, several software systems have been developed in order to facilitate the application of process calculi in the analysis of systems. Here, we only mention general process calculus tools. Tools that deal with specific extensions are mentioned below.

The most well-known general tool is the Concurrency Workbench, see [Moller and Stevens, 1999], dealing with CCS-type process calculus. There is also the variant CWB-NC, see [Zhang *et al.*, 2003] for the current state of affairs. There is the French set of tools CADP, see e.g. [Fernandez *et al.*, 1996]. Further, in the CSP tradition, there is the FDR tool (see `http://www.fsel.com/`).

The challenge in tool development is to combine an attractive user interface with a powerful and fast verification engine.

Verification

A measure of success of process calculus is the systems that have been successfully verified by means of techniques that come from process calculus. A good overview can be found in [Groote and Reniers, 2001]. Process calculus focuses on equational reasoning. Other successful techniques are model checking and theorem proving. Combination of these different approaches proves to be very promising.

Data

Process calculi are very successful in describing the dynamic behavior of systems. In describing the static aspects, treatment of data is very important. Actions and processes are parametrized with data elements. The combination of processes and data has received much attention over the years. A standardized formal description technique is LOTOS, see [Brinksma, 1989]. Another combination of processes and data is PSF, see [Mauw, 1991] with associated tooling. The process calculus with data μCRL (succeeded by mCRL2 [Groote and Mousavi, 2013]) has tooling focusing on equational verification, see e.g. [Groote and Lisser, 2001].

Time

Research on process calculus extended with a quantitative notion of time started with the work of Reed and Roscoe in the CSP context, see [Reed and Roscoe, 1988]. A textbook in this tradition is [Schneider, 2000].

There are many variants of CCS with timing, see e.g. [Yi, 1990], [Moller and Tofts, 1990]. In the ACP tradition, work starts with [Baeten and Bergstra, 1991]. An integrated theory, involving both discrete and dense time, both relative and absolute time, is presented in the book [Baeten and Middelburg, 2002]. Also the theory ATP can be mentioned, see [Nicollin and Sifakis, 1994]. An overview and comparison of different process algebras with timing can be found in [Baeten, 2003].

Tooling has been developed for processes with timing mostly in terms of timed automata, see e.g. UPPAAL [Larsen *et al.*, 1997] or KRONOS [Yovine, 1997]. Equational reasoning is investigated for μCRL with timing [Usenko, 2002].

Mobility

Research on networks of processes where processes are mobile and configuration of communication links is dynamic has been dominated by the π-calculus. An early reference is [Engberg and Nielsen, 1986], the standard reference is [Milner *et al.*, 1992] and the textbooks are [Milner, 1999; Sangiorgi and Walker, 2001]. The associated tool is the Mobility Workbench, see [Victor, 1994]. Also in this domain, it is important to gain more experience with protocol verification. On the theory side, there are a number of different equivalences that have been defined, and it is not clear which is the 'right' one to use.

Following, other calculi concerning mobility have been developed, notably the *ambient calculus*, see [Cardelli and Gordon, 2000]. As to unifying frameworks for different mobile calculi, Milner investigated action calculus [Milner, 1996] and bigraphs [Milner, 2001].

Over the years, the π-calculus is considered more and more as the standard process calculus to use. Important extensions that simplify some things are the psi-calculi, see [Bengtson *et al.*, 2011].

Probabilities and Stochastics

Process calculi extended with probabilistic or stochastic information have generated a lot of research. An early reference is [Hansson, 1991]. In the CSP tradition, there is [Lowe, 1993], in the CCS tradition [Hillston, 1996], in the ACP tradition [Baeten *et al.*, 1995]. There is the process algebra TIPP with associated tool, see e.g. [Götz *et al.*, 1993], and EMPA, see e.g. [Bernardo and Gorrieri, 1998].

The insight that both (unquantified) alternative composition and probabilistic choice are needed for a useful theory has gained attention, see e.g. the work in [D'Argenio, 1999] or [Andova, 2002].

Notions of abstraction are still a matter of continued research. The goal is to combine functional verification with performance analysis. A notion of approximation is very important here, see e.g. [Desharnais *et al.*, 2004]. Some recent references are [Jonsson *et al.*, 2001; Markovski, 2008; Georgievska, 2011].

Hybrid Systems

Systems that in their behavior depend on continuously changing variables other than time are the latest challenge to be addressed by process calculi. System descriptions involve differential algebraic equations, so here we get to the border of computer science with dynamics, in particular dynamic control theory. When discrete events are leading, but aspects of evolution are also taken into account, this is part of computer science, but when dynamic evolution is paramount, and some switching points occur, it becomes part of dynamic control theory. Process calculus research that can be mentioned is [Bergstra and Middelburg, 2005; Cuijpers and Reniers, 2003].

In process theory, work centres around *hybrid automata* [Alur *et al.*, 1995] and *hybrid I/O automata* [Lynch *et al.*, 1995]. A tool is HyTech, see [Henzinger *et al.*, 1995]. A connection with process calculus can be found in [Willemse, 2003; Baeten *et al.*, 2008].

Other Application Areas

Application of process calculus in other areas can be mentioned. A process calculus dealing with shared resources is ACSR [Lee *et al.*, 1994]. Process calculus has been used to give semantics of specification languages, such as POOL [Vaandrager, 1990] or MSC [Mauw and Reniers, 1994]. There is work on applications in security, see e.g. [Focardi and Gorrieri, 1995], [Abadi and Gordon, 1999] or [Schneider, 2001]. Work can be mentioned on the application of process calculi to biological processes, see e.g. [Priami *et al.*, 2001]. Other application areas are web services [Bravetti and Zavattaro, 2008; Laneve and Padovani, 2013], ubiquitous computing [Honda, 2006] and workflow [Puhlmann and Weske, 2005].

5 CONCLUSION

In this chapter, a brief history is sketched of concurrency theory, following two central breakthroughs. Early work centred around giving semantics to programming languages involving a parallel construct. Two breakthroughs were needed: first of all, abandoning the idea that a program is a transformation from input to output, replacing this by an approach where all intermediate states are important. We consider this development by considering the history of bisimulation. The second breakthrough consisted of replacing the notion of global variables by the paradigm of message passing and local variables. We consider this development by considering the history of process calculi.

In the seventies of the twentieth century, both these steps were taken, and full concurrency theories evolved. In doing so, concurrency theory became the underlying theory of parallel and distributed systems, extending formal language and automata theory with the central ingredient of *interaction*.

In the following years, much work has been done, and many concurrency theories have been formulated, extended with data, time, mobility, probabilities and stochastics. The work is not finished, however. We formulated some challenges for the future. More can be found in [Aceto *et al.*, 2005].

An interesting recent development is a reconsideration of the foundations of computation including interaction from concurrency theory. This yields a theory of executability, which is computability integrated with interaction, see [Baeten *et al.*, 2012].

BIBLIOGRAPHY

[Abadi and Gordon, 1999] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999.

[Aceto *et al.*, 2005] L. Aceto, W.J. Fokkink, A. Ingólfsdóttir, and Z. Ésik. Guest editors' foreword: Process algebra. *Theor. Comput. Sci.*, 335(2-3):127–129, 2005.

[Aceto *et al.*, 2007] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, 2007.

[Aceto *et al.*, 2012] L. Aceto, A. Ingolfsdottir, and J. Srba. The algorithmics of bisimilarity. In Sangiorgi and Rutten [2012].

[Aceto, 2003] L. Aceto. Some of my favourite results in classic process algebra. *Bulletin of the EATCS*, 81:90–108, 2003.

[Aczel, 1988] P. Aczel. *Non-well-founded Sets.* CSLI lecture notes, no. 14, 1988.

[Alur *et al.*, 1995] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[Alvarez *et al.*, 1991] Carme Alvarez, José L. Balcázar, Joaquim Gabarró, and Miklos Santha. Parallel complexity in the design and analysis on conurrent systems. In *Proc. PARLE '91: Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms*, volume 505 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 1991.

[Andova, 2002] S. Andova. *Probabilistic Process Algebra.* PhD thesis, Technische Universiteit Eindhoven, 2002.

[Apt *et al.*, 1980] K.R. Apt, N. Francez, and W.P. de Roever. A proof system for communicating sequential processes. *TOPLAS*, 2:359–385, 1980.

[Austry and Boudol, 1984] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Science*, 30:91–131, 1984.

[Baeten and Bergstra, 1991] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[Baeten and Middelburg, 2002] J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing.* EATCS Monographs. Springer Verlag, 2002.

[Baeten and Weijland, 1990] J. Baeten and W. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge Uuniversity Press, 1990.

[Baeten *et al.*, 1991] J.C.M. Baeten, J.A. Bergstra, C.A.R. Hoare, R. Milner, J. Parrow, and R. de Simone. The variety of process algebra. Deliverable ESPRIT Basic Research Action 3006, CONCUR, 1991.

[Baeten *et al.*, 1995] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. Axiomatizing probabilistic processes: ACP with generative probabilities. *Information and Computation*, 121(2):234–255, 1995.

[Baeten *et al.*, 2008] J.C.M. Baeten, D.A. van Beek, P.J.L. Cuijpers, M.A. Reniers, J.E. Rooda, R.R.H. Schiffelers, and R.J.M. Theunissen. Model-based engineering of embedded systems using the hybrid process algebra Chi. In C. Palamidessi and F.D. Valencia, editors, *Electronic Notes in Theoretical Computer Science*, volume 209. Elsevier Science Publishers, 2008.

[Baeten *et al.*, 2010] J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Number 50 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2010.

[Baeten *et al.*, 2012] J.C.M. Baeten, B. Luttik, and P. van Tilburg. Turing meets Milner. In M. Koutny and I. Ulidowski, editors, *Proceedings CONCUR 2012*, number 7454 in Lecture Notes in Computer Science, pages 1–20, 2012.

[Baeten, 1993] J.C.M. Baeten. The total order assumption. In S. Purushothaman and A. Zwarico, editors, *Proceedings First North American Process Algebra Workshop*, Workshops in Computing, pages 231–240. Springer Verlag, 1993.

[Baeten, 2003] J.C.M. Baeten. Embedding untimed into timed process algebra: The case for explicit termination. *Mathematical Structures in Computer Science*, 13:589–618, 2003.

[Bakker and Zucker, 1982a] J.W. de Bakker and J.I. Zucker. Denotational semantics of concurrency. In *Proceedings 14th Symposium on Theory of Computing*, pages 153–158. ACM, 1982.

[Bakker and Zucker, 1982b] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

[Balcázar *et al.*, 1992] José L. Balcázar, Joaquim Gabarró, and Miklos Santha. Deciding Bisimilarity is P-Complete. *Formal Asp. Comput.*, 4(6A):638–648, 1992.

[Banach, 1922] S. Banach. Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales. *Fundamenta Mathematicae*, 3:133–181, 1922.

[Barwise and Moss, 1996] J. Barwise and L. Moss. *Vicious Circles: On the Mathematics of Non-Wellfounded Phenomena*. CSLI (Center for the Study of Language and Information), 1996.

[Bekič, 1971] H. Bekič. Towards a mathematical theory of processes. Technical Report TR 25.125, IBM Laboratory Vienna, 1971.

[Bekič, 1984] H. Bekič. *Programming Languages and Their Definition (Selected Papers edited by C.B. Jones)*. Number 177 in LNCS. Springer Verlag, 1984.

[Bengtson *et al.*, 2011] J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7:1–44, 2011.

[Benthem, 1976] J. van Benthem. *Modal Correspondence Theory*. PhD thesis, Mathematisch Instituut & Instituut voor Grondslagenonderzoek, University of Amsterdam, 1976.

[Benthem, 1983] J. van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, 1983.

[Benthem, 1984] J. van Benthem. Correspondence theory. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume 2, pages 167–247. Reidel, 1984.

[Bergstra and Klop, 1982] J.A. Bergstra and J.W. Klop. Fixed point semantics in process algebra. Technical Report IW 208, Mathematical Centre, Amsterdam, 1982.

[Bergstra and Klop, 1984a] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings 11th ICALP*, number 172 in LNCS, pages 82–95. Springer Verlag, 1984.

[Bergstra and Klop, 1984b] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[Bergstra and Klop, 1992] J.A. Bergstra and J.W. Klop. A convergence theorem in process algebra. In J.W. de Bakker and J.J.M.M. Rutten, editors, *Ten Years of Concurrency Semantics*, pages 164–195. World Scientific, 1992.

[Bergstra and Middelburg, 2005] J.A. Bergstra and C.A. Middelburg. Process algebra for hybrid systems. *Theoretical Computer Science*, 335, 2005.

[Bergstra *et al.*, 1987] J.A. Bergstra, J.W. Klop, and E.-R. Olderog. Failures without chaos: A new process semantics for fair abstraction. In M. Wirsing, editor, *Proceedings IFIP Conference on Formal Description of Programming Concepts III*, pages 77–103. North-Holland, 1987.

[Bergstra *et al.*, 2001] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, Amsterdam, 2001.

[Bernardo and Gorrieri, 1998] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with non-determinism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, 1998.

[Best *et al.*, 2001] E. Best, R. Devillers, and M. Koutny. A unified model for nets and process algebras. In [Bergstra *et al.*, 2001], pp. 945–1045, 2001.

[Brand, 1978] D. Brand. Algebraic simulation between parallel programs. Research Report RC 7206, Yorktown Heights, N.Y., 39 pp., 1978.

[Brauer and Reisig, 2006] Wilfried Brauer and Wolfgang Reisig. Carl Adam Petri und die "Petrinetze". *Informatik Spektrum*, 29:369–374, 2006.

[Bravetti and Zavattaro, 2008] M. Bravetti and G. Zavattaro. A foundational theory of contracts for multi-party service composition. *Fundamenta Informaticae*, 89:451–478, 2008.

[Brinksma, 1989] E. Brinksma, editor. *Information Processing Systems, Open Systems Interconnection, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, volume IS-8807 of *International Standard*. ISO, Geneva, 1989.

[Brookes *et al.*, 1984] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[Buchholz, 1994] P. Buchholz. Markovian process algebra: composition and equivalence. In U. Herzog and M. Rettelbach, editors, *Proc. 2nd Workshop on Process Algebras and Performance Modelling*, pages 11–30. Arbeitsberichte des IMMD, Band 27, Nr. 4, 1994.

[Burge, 1975] William H. Burge. Stream processing functions. *IBM Journal of Research and Development*, 19(1):12–25, 1975.

[Burkart *et al.*, 2001] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In [Bergstra *et al.*, 2001], pp. 545–623, 2001.

[Cardelli and Gordon, 2000] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000.

[Cuijpers and Reniers, 2003] P.J.L. Cuijpers and M.A. Reniers. Hybrid process algebra. Technical Report CS-R 03/07, Technische Universiteit Eindhoven, Dept. of Comp. Sci., 2003.

[D'Argenio, 1999] P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. PhD thesis, University of Twente, 1999.

[de Roever, 1977] Willem P. de Roever. On backtracking and greatest fixpoints. In Arto Salomaa and Magnus Steinby, editors, *Fourth Colloquium on Automata, Languages and Programming (ICALP)*, volume 52 of *Lecture Notes in Computer Science*, pages 412–429. Springer, 1977.

[Desharnais *et al.*, 2004] J. Desharnais, V. Gupta, R. Jagadeesan, and P. Panangaden. Metrics for labeled Markov systems. *Theoretical Computer Science*, 318:323–354, 2004.

[Dijkstra, 1975] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Engberg and Nielsen, 1986] U. Engberg and M. Nielsen. A calculus of communicating systems with label passing. Technical Report DAIMI PB-208, Aarhus University, 1986.

[Fernandez *et al.*, 1996] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CAESAR/ALDEBARAN development package): A protocol validation and verification toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 437–440. Springer Verlag, 1996.

[Floyd, 1967] R. W. Floyd. Assigning meaning to programs. In *Proc. Symposia in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[Focardi and Gorrieri, 1995] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3:5–33, 1995.

[Forti and Honsell, 1983] M. Forti and F. Honsell. Set theory with free construction principles. *Annali Scuola Normale Superiore, Pisa, Serie IV*, X(3):493–522, 1983.

[Georgievska, 2011] S. Georgievska. *Probability and Hiding in Concurrent Processes*. PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science, Eindhoven, the Netherlands, 2011.

[Ginzburg, 1968] A. Ginzburg. *Algebraic Theory of Automata*. Academic Press, 1968.

[Glabbeek, 1986] R.J. van Glabbeek. Notes on the methodology of CCS and CSP. Technical Report CS-R8624, Centrum Wiskunde & Informatica, Amsterdam, 1986.

[Glabbeek, 1990a] R.J. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[Glabbeek, 1990b] R.J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In Jos C. M. Baeten and Jan Willem Klop, editors, *First Conference on Concurrency Theory (CONCUR'90)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.

[Glabbeek, 1993] R.J. van Glabbeek. The linear time — branching time spectrum II (the semantics of sequential systems with silent moves). In E. Best, editor, *Fourth Conference on Concurrency Theory (CONCUR'93)*, volume 715, pages 66–81. Springer, 1993.

[Glabbeek, 2001] R.J. van Glabbeek. The linear time – branching time spectrum I. The semantics of concrete, sequential processes. In [Bergstra *et al.*, 2001], pp. 3–100, 2001.

[Götz *et al.*, 1993] N. Götz, U. Herzog, and M. Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In L. Donatiello and R. Nelson, editors, *Performance Evaluation of Computer and Communication Systems*, number 729 in LNCS, pages 121–146. Springer, 1993.

[Gourlay *et al.*, 1979] John S. Gourlay, William C. Rounds, and Richard Statman. On properties preserved by contraction of concurrent systems. In Gilles Kahn, editor, *International Symposium on Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 1979.

[Groote and Lisser, 2001] J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. Technical Report SEN-R0117, CWI, Amsterdam, 2001.

[Groote and Mousavi, 2013] J.F. Groote and M.R. Mousavi. *Modelling and Analysis of Communicating Systems.* MIT Press, 2013.

[Groote and Reniers, 2001] J.F. Groote and M.A. Reniers. Algebraic process verification. In [Bergstra *et al.*, 2001], pp. 1151–1208, 2001.

[Hansson, 1991] H. Hansson. *Time and Probability in Formal Design of Distributed Systems.* PhD thesis, University of Uppsala, 1991.

[Harel and Pnueli, 1985] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems. Springer, 1985.

[Hennessy and Milner, 1980] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J.W. de Bakker and J. van Leeuwen, editors, *Proceedings 7th ICALP*, number 85 in Lecture Notes in Computer Science, pages 299–309. Springer Verlag, 1980.

[Hennessy, 1988] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[Henzinger *et al.*, 1995] T.A. Henzinger, P. Ho, and H. Wong-Toi. Hy-Tech: The next generation. In *Proceedings RTSS*, pages 56–65. IEEE, 1995.

[Hillston, 1996] J. Hillston. *A Compositional Approach to Performance Modelling.* PhD thesis, Cambridge University Press, 1996.

[Hinnion, 1980] R. Hinnion. Contraction de structures et application à NFU. *Comptes Rendus Acad. des Sciences de Paris*, 290, Sér. A:677–680, 1980.

[Hinnion, 1981] R. Hinnion. Extensional quotients of structures and applications to the study of the axiom of extensionality. *Bulletin de la Société Mathmatique de Belgique*, XXXIII (Fas. II, Sér. B):173–206, 1981.

[Hoare, 1969] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[Hoare, 1978] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoare, 1980] C.A.R. Hoare. A model for communicating sequential processes. In R.M. McKeag and A.M. Macnaghten, editors, *On the Construction of Programs*, pages 229–254. Cambridge University Press, 1980.

[Hoare, 1985] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Honda, 2006] K. Honda. Process algebras in the age of ubiquitous computing. *Electr. Notes Theor. Comput. Sci.*, 162:217–220, 2006.

[Huffman, 1954] D.A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute (Mar. 1954) and (Apr. 1954)*, 257(3–4):161–190 and 275–303, 1954.

[Jensen, 1980] Kurt Jensen. A method to compare the descriptive power of different types of petri nets. In Piotr Dembinski, editor, *Proc. 9th Mathematical Foundations of Computer Science 1980 (MFCS'80), Rydzyna, Poland, September 1980*, volume 88 of *Lecture Notes in Computer Science*, pages 348–361. Springer, 1980.

[Jonsson *et al.*, 2001] B. Jonsson, Yi Wang, and K.G. Larsen. Probabilistic extensions of process algebras. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 685–710. North-Holland, 2001.

[Kahn, 1974] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475. North-Holland, 1974.

[Kanellakis and Smolka, 1990] Paris C. Kanellakis and Scott A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.*, 86(1):43–68, 1990.

[Kemeny and Snell, 1960] J. Kemeny and J. L. Snell. *Finite Markov Chains.* Van Nostrand Co. Ltd., London, 1960.

[Kwong, 1977] Y. S. Kwong. On reduction of asynchronous systems. *Theoretical Computer Science*, 5(1):25–50, 1977.

[Landin, 1964] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[Landin, 1965a] Peter J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: Part I. *Commun. ACM*, 8(2):89–101, 1965.

[Landin, 1965b] Peter J. Landin. A correspondence between ALGOL 60 and Church's Lambda-notations: Part II. *Commun. ACM*, 8(3):158–167, 1965.

[Landin, 1969] P. Landin. A program-machine symmetric automata theory. *Machine Intelligence*, 5:99–120, 1969.

[Laneve and Padovani, 2013] C. Laneve and L. Padovani. An algebraic theory for web service contracts. In E. Broch Johnsen and L. Petre, editors, *IFM*, volume 7940 of *Lecture Notes in Computer Science*, pages 301–315. Springer, 2013.

[Larsen and Skou, 1991] Kim Guldstrand Larsen and Arne Skou. Bisimulation through probabilistic testing. *Inf. Comput.*, 94(1):1–28, 1991. Preliminary version in POPL'89, 344–352, 1989.

[Larsen *et al.*, 1997] K.G. Larsen, P. Pettersson, and Wang Yi. Uppaal in a nutshell. *Journal of Software Tools for Technology Transfer*, 1, 1997.

[Lee *et al.*, 1994] I. Lee, P. Bremond-Gregoire, and R.Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, 1994. Special Issue on Real-Time.

[Linz, 2001] P. Linz. *An Introduction to Formal Languages and Automata.* Jones and Bartlett, 2001.

[Lowe, 1993] G. Lowe. *Probabilities and Priorities in Timed CSP.* PhD thesis, University of Oxford, 1993.

[Lynch *et al.*, 1995] N. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O automata. In T. Henzinger, R. Alur, and E. Sontag, editors, *Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science. Springer Verlag, 1995.

[MacLane and Birkhoff, 1967] S. MacLane and G. Birkhoff. *Algebra.* MacMillan, 1967.

[Manna, 1969] Z. Manna. The correctness of programs. *J. Computer and System Sciences*, 3(2):119–127, 1969.

[Markovski, 2008] J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science, Eindhoven, the Netherlands, 2008.

[Mauw and Reniers, 1994] S. Mauw and M.A. Reniers. An algebraic semantics for basic message sequence charts. *The Computer Journal*, 37:269–277, 1994.

[Mauw, 1991] S. Mauw. *PSF: a Process Specification Formalism.* PhD thesis, University of Amsterdam, 1991. See `http://carol.science.uva.nl/~psf/`.

[Mazurkiewicz, 1977] A. Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report DAIMI PB-78, Aarhus University, 1977.

[McCarthy, 1963] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[Meyer and Stockmeyer, 1972] Albert R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *13th Annual Symposium on Switching and Automata Theory (FOCS)*, pages 125–129. IEEE, 1972.

[Milne and Milner, 1979] G.J. Milne and R. Milner. Concurrent processes and their syntax. *Journal of the ACM*, 26(2):302–321, 1979.

[Milne, 1983] G.J. Milne. CIRCAL: A calculus for circuit description. *Integration*, 1:121–160, 1983.

[Milner and Tofte, 1991] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991. Also Tech. Rep. ECS-LFCS-88-65, University of Edinburgh, 1988.

[Milner *et al.*, 1992] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[Milner, 1970] R. Milner. A formal notion of simulation between programs. Memo 14, Computers and Logic Resarch Group, University College of Swansea, U.K., 1970.

[Milner, 1971a] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int. Joint Conferences on Artificial Intelligence*. British Comp. Soc. 1971.

[Milner, 1971b] R. Milner. Program simulation: an extended formal notion. Memo 17, Computers and Logic Resarch Group, University College of Swansea, U.K., 1971.

[Milner, 1973] R. Milner. An approach to the semantics of parallel programs. In *Proceedings Convegno di informatica Teoretica*, pages 285–301, Pisa, 1973. Instituto di Elaborazione della Informazione.

[Milner, 1975] R. Milner. Processes: A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Proceedings Logic Colloquium*, number 80 in Studies in Logic and the Foundations of Mathematics, pages 157–174. North-Holland, 1975.

[Milner, 1978a] R. Milner. Algebras for communicating systems. In *Proc. AFCET/SMF joint colloquium in Applied Mathematics*, Paris, 1978.

[Milner, 1978b] R. Milner. Synthesis of communicating behaviour. In J. Winkowski, editor, *Proc. 7th MFCS*, number 64 in LNCS, pages 71–83, Zakopane, 1978. Springer Verlag.

[Milner, 1979] R. Milner. Flowgraphs and flow algebras. *Journal of the ACM*, 26(4):794–818, 1979.

[Milner, 1980] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Milner, 1983] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[Milner, 1984] R. Milner. A complete inference system for a class of regular behaviours. *Journal of Computer System Science*, 28:439–466, 1984.

[Milner, 1989] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Milner, 1996] R. Milner. Calculi for interaction. *Acta Informatica*, 33:707–737, 1996.

[Milner, 1999] R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[Milner, 2001] R. Milner. Bigraphical reactive systems. In K.G. Larsen and M. Nielsen, editors, *Proceedings CONCUR '01*, number 2154 in LNCS, pages 16–35. Springer Verlag, 2001.

[Moller and Stevens, 1999] F. Moller and P. Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from `http://www.dcs.ed.ac.uk/home/cwb/`, 1999.

[Moller and Tofts, 1990] F. Moller and C. Tofts. A temporal calculus of communicating systems. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR'90*, number 458 in LNCS, pages 401–415. Springer Verlag, 1990.

[Moore, 1956] E.F. Moore. Gedanken experiments on sequential machines. *Automata Studies, Annals of Mathematics Series*, 34:129–153, 1956.

[Nerode, 1958] A. Nerode. Linear automaton transformations. In *Proc. American Mathematical Society*, volume 9, pages 541–544, 1958.

[Nicollin and Sifakis, 1994] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[Owicki and Gries, 1976] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19:279–285, 1976.

[Park, 1979] D. Park. On the semantics of fair parallelism. In *Proc. Abstract Software Specifications, Copenhagen Winter School*, Lecture Notes in Computer Science, pages 504–526. Springer, 1979.

[Park, 1981a] D.M.R. Park. Concurrency on automata and infinite sequences. In P. Deussen, editor, *Conf. on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[Park, 1981b] D.M.R. Park. A new equivalence notion for communicating systems. In G. Maurer, editor, *Bulletin EATCS*, volume 14, pages 78–80, 1981. Abstract of the talk presented at the Second Workshop on the Semantics of Programming Languages, Bad Honnef, March 16–20 1981. Abstracts collected in the Bulletin by B. Mayoh.

[Petri, 1962] C.A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut fuer Instrumentelle Mathematik, Bonn, 1962.

[Petri, 1980] C.A. Petri. Introduction to general net theory. In W. Brauer, editor, *Proc. Advanced Course on General Net Theory, Processes and Systems*, number 84 in LNCS, pages 1–20. Springer Verlag, 1980.

[Plotkin, 1976] G.D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5:452–487, 1976.

[Plotkin, 1981] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981. Reprinted as [Plotkin, 2004a].

[Plotkin, 2004a] G.D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60:17–139, 2004.

[Plotkin, 2004b] G.D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60(1):3–16, 2004.

[Pnueli, 1977] A. Pnueli. The temporal logic of programs. In *Proceedings 19th Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[Pous and Sangiorgi, 2012] Damien Pous and Davide Sangiorgi. Enhancements of the bisimulation proof method. In Sangiorgi and Rutten [2012].

[Priami *et al.*, 2001] C. Priami, A. Regev, W. Silverman, and E. Shapiro. Application of stochastic process algebras to bioinformatics of molecular processes. *Information Processing Letters*, 80:25–31, 2001.

[Puhlmann and Weske, 2005] F. Puhlmann and M. Weske. Using the *pi*-calculus for formalizing workflow patterns. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management*, volume 3649, pages 153–168, 2005.

[Reed and Roscoe, 1988] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[Rem, 1983] M. Rem. Partially ordered computations, with applications to VLSI design. In J.W. de Bakker and J. van Leeuwen, editors, *Foundations of Computer Science IV*, volume 159 of *Mathematical Centre Tracts*, pages 1–44. Mathematical Centre, Amsterdam, 1983.

[Rutten and Jacobs, 2012] Jan Rutten and Bart Jacobs. (co)algebras and (co)induction. In Sangiorgi and Rutten [2012].

[Rutten and Turi, 1994] J. Rutten and D. Turi. Initial algebra and final coalgebra semantics for concurrency. In *Proc. Rex School/Symposium 1993 "A Decade of Concurrency — Reflexions and Perspectives"*, volume 803 of *Lecture Notes in Computer Science*. Springer, 1994.

[Sangiorgi and Rutten, 2012] Davide Sangiorgi and Jan Rutten, editors. *Advanced Topics in Bisimulation and Coinduction*. Cambridge University Press, 2012.

[Sangiorgi and Walker, 2001] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[Sangiorgi, 2009] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4), 2009.

[Sangiorgi, 2012] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2012.

[Schneider, 2000] S.A. Schneider. *Concurrent and Real-Time Systems (the CSP Approach)*. Worldwide Series in Computer Science. Wiley, 2000.

[Schneider, 2001] S.A. Schneider. Process algebra and security. In K.G. Larsen and M. Nielsen, editors, *Proceedings CONCUR '01*, number 2154 in LNCS, pages 37–38. Springer Verlag, 2001.

[Scott and Strachey, 1971] D.S. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings Symposium Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn Press, 1971.

[Scott, 1960] D. Scott. A different kind of model for set theory. Unpublished paper, given at the 1960 Stanford Congress of Logic, Methodology and Philosophy of Science, 1960.

[Segerberg, 1968] K. Segerberg. Decidability of S4.1. *Theoria*, 34:7–20, 1968.

[Segerberg, 1971] Krister Segerberg. An essay in classical modal logic. Filosofiska Studier, Uppsala, 1971.

[Stirling, 2012] Colin Stirling. Bisimulation and logic. In Sangiorgi and Rutten [2012].

[Usenko, 2002] Y.S. Usenko. *Linearization in μCRL*. PhD thesis, Technische Universiteit Eindhoven, 2002.

[Vaandrager, 1990] F.W. Vaandrager. Process algebra semantics of POOL. In J.C.M. Baeten, editor, *Applications of Process Algebra*, number 17 in Cambridge Tracts in Theoretical Computer Science, pages 173–236. Cambridge University Press, 1990.

[Victor, 1994] B. Victor. *A Verification Tool for the Polyadic π-Calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.

[Willemse, 2003] T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing.* PhD thesis, Technische Universiteit Eindhoven, 2003.

[Yi, 1990] Wang Yi. Real-time behaviour of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *Proceedings CONCUR'90*, number 458 in LNCS, pages 502–520. Springer Verlag, 1990.

[Yovine, 1997] S. Yovine. Kronos: A verification tool for real-time systems. *Journal of Software Tools for Technology Transfer*, 1:123–133, 1997.

[Zhang *et al.*, 2003] D. Zhang, R. Cleaveland, and E. Stark. The integrated CWB-NC/PIOAtool for functional verification and performance analysis of concurrent systems. In H. Garavel and J. Hatcliff, editors, *Proceedings TACAS '03*, number 2619 in Lecture Notes in Computer Science, pages 431–436. Springer-Verlag, 2003.