Wan Fokkink

Modelling Distributed Systems

Protocol Verification with μCRL

2nd edition

March 27, 2017

Springer

Preface

A distributed system is driven by its separate concurrent components, which are being executed in parallel. In today's world of wireless and mobile networking, distributed algorithms and network protocols tend to constitute an important aspect of system design. Verifying the correctness of such algorithms and protocols tends to be a formidable task, as even simple behaviours become wildly complicated when they are executed in parallel.

Much effort is being spent on the development of novel techniques for the formal description and analysis of distributed systems. However, the majority of these techniques have up to now not been used widely, due to the sharp learning curve required to adopt them. Such verification techniques often have non-trivial theoretical underpinnings, and, as a result, according to practitioners, it requires in-depth knowledge and sophisticated mathematical skills to apply them.

The main aim of this book is to provide a gentle guide to some of the most prominent formal verication techniques for distributed systems. For a start, the reader is acquainted with the algebraic specification of distributed systems. The μCRL toolset is used as a vehicle to teach students how to specify and analyse real-life distributed algorithms and network protocols with the support of specialised tools. μCRL consists of a specification language and verification toolset based on process algebra and abstract data types. Such formal system specifications can be verified at two different levels: either by reasoning about such a specification on a symbolic level, or by generating its state space explicitly. State-of-the-art methods are presented for these two different verification approaches.

Case studies have a valuable role to play both in promoting and demonstrating particular verification techniques, and by providing practical examples of their application. At the same time, case studies help in pushing forward the boundaries of verification techniques. Therefore, formal specifications of several network protocols from the literature are studied in detail, to illustrate how the framework can be applied.

This book was developed from a set of lecture notes for an MSc course on 'Protocol Validation', which I have been lecturing at the Vrije Universiteit Amsterdam since 2001. For prospective lecturers there is a set of slides available on the Web, which can be used to present a course based on this book. Also lab exercises and example specifications are available. I strongly recommend that lecturers include one substantial and open-ended practical exercise, in which the students should (in teams of two or three) specify and verify a real-life distributed system. The book offers one such case study, in the form of a trolley bed on which a patient can lie inside a medical scanning machine for magnetic resonance imaging.

My earlier textbook *Introduction to Process Algebra* [46], which appeared in the same series in 2000, can in principle be used as a companion. In that book, the theoretical foundations of process algebra are explained in full detail. Here, a more pragmatic view is taken, in that the basics of process algebra and abstract data types are only explained up to a level that suffices for using them in the specification and verification of distributed algorithms and network protocols. The mathematical proofs underlying the verification techniques are largely omitted.

I would like to thank the teaching assistants and students who took part in the course 'Protocol Validation' for their constructive comments and suggestions regarding the lecture notes. For the structure of Chapters 2 and 3, I benefited from reading the chapter on *Algebraic Process Verification* in the *Handbook of Process Algebra* by Jan Friso Groote and Michel Reniers [61]; they also provided useful feedback on earlier versions of this textbook. Moreover, Jan Friso Groote provided the system description of the patient support system.

Utrecht, *Wan Fokkink*
March 2007

Contents

1

Introduction

In the context of hardware and software systems, formal verification is the act of proving or disproving a property of a system with respect to a formal specification, using methods rooted in mathematics, such as logic and graph theory. A formal specification of a system can help to obtain not only a better (more modular) description, but also a better understanding and a more abstract view of the system. Formal verification, supported with (semi-)automated tools, can detect errors in the design that are not easily found using testing, and can be used to establish the correctness of the design. Formal verification has, for instance, been applied to communication and cryptographic protocols, distributed algorithms, combinatorial circuits, and software expressed as source code. A comprehensive introduction into the field of concurrency and formal verification can be found in [11, 91].

Process algebra focuses on the specification and manipulation of process terms as induced by a collection of operator symbols. Such a process term constitutes a formal specification of a system. Typically, process algebras contain action names, to express atomic events, and the two basic operators alternative and sequential composition to build finite processes. Recursion allows one to capture infinite behaviour.

Verifying the correctness of distributed systems is a challenge, due to their inherent parallelism. In order to study the behaviour of distributed systems in detail, it is imperative that they are dissected into their concurrent components. Fundamental to process algebra is therefore a parallel operator, to break down distributed systems into their concurrent components, at the same time expressing the communication of corresponding send and receive events at different components. An encapsulation operator takes care that such corresponding send and receive events can only occur in synchronisation. Finally, a hiding operator allows one to abstract away from the resulting communication events, and from the internal events at a component.

In process algebras, each operator in the language is given meaning through a characterising set of equations, called axioms. If two process terms (built from the aforementioned operators) can be equated by means of the axioms,

then they represent equivalent system behaviours. Thus the axioms form an elementary basis for equational reasoning about processes. Process algebras such as CCS [27, 86], CSP [69, 98] and ACP [12, 6, 46] offer an excellent framework for the description of distributed systems, and they are well equipped for the study of their behavioural properties. Temporal logics can be used to formally express such properties.

System behaviour generally consists of a mix of processes and data. Processes are the control mechanisms for the manipulation of data. While processes are dynamic and active, data are static and passive. In algebraic specification [77], each data type is defined by declaring a collection of function symbols, from which one can build data terms, together with a set of axioms, saying which data terms are equal. Algebraic specification allows one to give relatively simple and precise definitions of abstract data types. A major advantage of this approach is that it is easily explained and formally defined, and that it constitutes a uniform framework for defining general data types. Moreover, all properties of a data type must be denoted explicitly, and henceforth it is clear which assumptions can be used when proving properties about data or processes. Term rewriting [104] provides a straightforward method for implementing algebraic specifications of abstract data types. Concluding, as long as one is interested in clear and precise specifications, and not in optimised implementations, algebraic specification is the best available method. However, one should be aware that it does not allow one to conveniently use high-level constructs for compact specification of complex data types, nor optimisations supporting fast computation (such as decimal representations of natural numbers).

Process algebras tend to lack the ability to handle data. In case data become part of a process theory, one often has to resort to infinite sets of axioms where variables are indexed with data values. In order to make data a first class citizen in the formal specification of systems, the language μCRL [59] has been developed. Basically, μCRL is based on the process algebra ACP, extended with the algebraic specification of abstract data types. In order to intertwine processes with data, the action names and recursion variables that are used to express process behaviour can be parametrised with data types. Moreover, a conditional (if-then-else) construct can be used to let data elements influence the course of a process, and the alternative composition operator is allowed to range over possibly infinite data domains. Despite its lack of 'advanced' features, μCRL has been shown to be remarkably apt for the description of real-life distributed systems.

A proof theory for μCRL has been developed [58], based in part on the axiomatic semantics of the process algebra ACP and on some basic abstract data types. This proof theory, in combination with proof methods that were developed in e.g. [16, 63], has enabled the verification of distributed systems in a precise and logical way, which is slowly turning into a routine. Theorem provers such as PVS [88], Isabelle/HOL [87] and Coq [14] are being used to help in finding and checking derivations in μCRL. A considerable number of

distributed systems from the literature and from industry have been verified in μCRL, e.g. [31, 54, 89, 99, 105], often with the help of a theorem prover, e.g. [5, 15, 55]. Typically, these verifications lead to the detection of a number of mistakes in the specification of the system under scrutiny, and the support of theorem provers helps to detect flaws in the correctness proof, or even in the statement of correctness.

To each μCRL specification there belongs a directed graph, called the state space, in which the states are process terms, and the edges are labelled with actions. In this state space, an edge $p \stackrel{a(d)}{\rightarrow} p'$ means that process term p can perform action a, parametrised with datum d, to evolve into process term p'. If the state space belonging to a μCRL specification is finite, then the μCRL toolset [18], in combination with the CADP toolset [48], can generate and visualise this state space. Model checking [37] provides a framework to efficiently prove interesting properties of large state spaces, formulated in some temporal logic. While the process algebraic proofs that were discussed earlier can cope with an open environment, such as an unspecified data type or network topology, the generation of a state space belonging to a distributed system requires that the environment is given in full detail. This means that for instance each unspecified data type (typically, the set of objects that can be received by the distributed system from the 'outside world') has to be instantiated with an ad hoc finite collection of elements, and that a particular configuration of the network topology has to be chosen.

A severe complication in the generation of state spaces is that, in real life, a distributed system typically contains in the order of 2^{100} states or more. In that sense a μCRL specification is like Pandora's Box; as soon as it is opened, the state space may explode. This means that generating, storing and analysing a state space becomes problematic, to say the least. Several methods are being developed to tackle large state spaces. Distributed state space generation and verification algorithms make it possible to store a state space on a number of processors, and analyse it in a distributed fashion [17, 20]. On-the-fly analysis [70] allows one to generate only part of a state space. Structural symmetries in the description of a system can often be exploited to reduce the resulting state space [36]. Scenario-based verification [41] takes as its starting point a certain scenario of inputs from the outside world, to restrict the behavioural possibilities of a distributed system. A μCRL specification may be manipulated in such a way that the resulting state space becomes significantly smaller [52]. And the ATerm library [26] allows one to store state spaces in an efficient way by maximal sharing, meaning that if two states (i.e., two process terms) contain the same subterm, then this subterm is shared in the memory space.

This text is set up as follows. Chapter 2 gives an introduction into the algebraic specification of abstract data types. Chapter 3 provides an overview of process algebra, and presents the basics of the specification language μCRL. In Chapter 4 it is explained how one can abstract away from the internal and

communication events of a process. Chapter 5 contains a number of μCRL specifications of network protocols from the literature, together with extensive explanations to guide the reader through these specifications. In Chapter 6 it is described how a μCRL specification can be reduced to a linear form, from which a state space can be generated; moreover, verification algorithms on state spaces are explained. In Chapter 7, techniques are presented to analyse μCRL specifications at a symbolic level. Also a symbolic verification of the tree identify protocol is presented. Finally, Appendix A contains a brief explanation on how to use the μCRL and CADP toolsets.

2

Abstract Data Types

This chapter contains an introduction to the algebraic specification of abstract data types, by means of a set of equations. See [77] for a lucid overview of this field.

2.1 Algebraic Specification

We start with a standard example.

Example 1. We specify the natural numbers with addition and multiplication. The *signature* consists of the function 0 with no arguments, the successor function S with one argument, and the functions addition *plus* and multiplication *mul* with both two arguments. The *equality relation* $=$ on the data terms over this signature is specified by four *axioms*:

$$\begin{aligned} \mathit{plus}(x, 0) &= x \\ \mathit{plus}(x, S(y)) &= S(\mathit{plus}(x, y)) \\ \mathit{mul}(x, 0) &= 0 \\ \mathit{mul}(x, S(y)) &= \mathit{plus}(\mathit{mul}(x, y), x) \end{aligned}$$

The *initial model* of these axioms consists of the distinct classes

$$[\![0]\!],\ [\![S(0)]\!],\ [\![S^2(0)]\!],\ [\![S^3(0)]\!], \ldots$$

The first three classes, with some typical representatives of each of these classes, are depicted in Fig. 2.1.

For example, the equation $\mathit{plus}(S(S(S(0))), S(0)) = \mathit{mul}(S(S(0)), S(S(0)))$ (i.e., $3 + 1 = 2 \cdot 2$) can be derived from the axioms for the natural numbers as follows.

$$\mathit{plus}(S(S(S(0))), S(0)) = S(\mathit{plus}(S(S(S(0))), 0)) = S(S(S(S(0))))$$

and

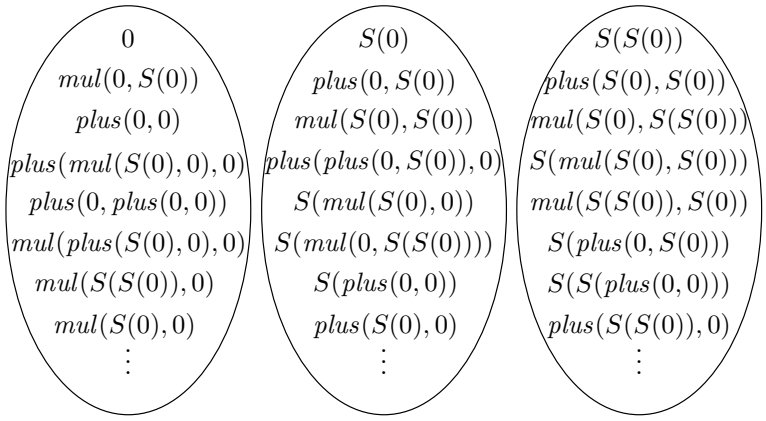


Fig. 2.1. Initial model for the natural numbers

$$\begin{aligned}
mul(S(S(0)), S(S(0))) &= plus(mul(S(S(0)), S(0)), S(S(0))) \\
&= plus(plus(mul(S(S(0)), 0), S(S(0))), S(S(0))) \\
&= plus(plus(0, S(S(0))), S(S(0))) \\
&= plus(S(plus(0, S(0))), S(S(0))) \\
&= plus(S(S(plus(0, 0))), S(S(0))) \\
&= plus(S(S(0)), S(S(0))) \\
&= S(plus(S(S(0)), S(0))) \\
&= S(S(plus(S(S(0)), 0))) \\
&= S(S(S(S(0)))).
\end{aligned}$$

In general, an *algebraic specification* consists of three parts.

1. A *signature*, consisting of *sort names* and *function symbols*, from which one can build *data terms*. Each function symbol f is assigned a type $f : D_1 \times \cdots \times D_n \to D$ for some *arity* $n \geq 0$, where $D_1, \ldots, D_n, D$ are sort names. This means that if $d_1, \ldots, d_n$ are data terms of sorts $D_1, \ldots, D_n$, respectively, then $f(d_1, \ldots, d_n)$ is a data term of sort D. (If $n = 0$, then we write f instead of $f()$.)
2. For each sort D a countably infinite set of *data variables* of sort D.
3. A set of *axioms*, i.e., equations $d = e$ between data terms (possibly containing data variables) of the same sort, which induces an equality relation on data terms.

A *context* $C[]$ is a data term containing exactly one occurrence of the special symbol $[]$. For each data term d, $C[d]$ denotes the data term that is obtained if one replaces in $C[]$ the symbol $[]$ by d. The envisioned equality relation on data terms is obtained by applying to the axioms all possible substitutions of data terms for data variables, and closing the relation under contexts and equivalence. To be more precise:

- if $d = e$ is an axiom, then $\sigma(d) = \sigma(e)$ holds for all possible substitutions σ from data variables to data terms;
- if $d = e$ holds, then $C[d] = C[e]$ holds for all possible contexts $C[]$;
- $d = d$ holds for all data terms d;
- if $d = e$ holds, then $e = d$ holds;
- if $d_1 = d_2$ and $d_2 = d_3$ hold, then $d_1 = d_3$ holds.

In brief, the *initial model* of an algebraic specification is obtained as follows. The equality relation induced by the axioms constitutes an equivalence relation on the set of data terms. Each equivalence class of derivably equal data terms constitutes an element in the initial model of the algebraic specification. The function symbols in the signature can be lifted to the initial model in a straightforward fashion. If f is a function symbol of arity n, and $\overline{d}$ denotes the equivalence class of a data term d, then $f(\overline{d_1}, \ldots, \overline{d_n})$ equals $\overline{f(d_1, \ldots, d_n)}$.

In μCRL, each sort name is declared using the keyword **sort**. Each declared sort represents a non-empty set of data elements. For example, declaring the sort of the booleans is simply done by:

sort *Bool*

μCRL uses algebraic specification of abstract data types, with an explicit recognition of so-called *constructor* function symbols, which intuitively cannot be eliminated from data terms. For example, in the case of the natural numbers, the zero 0 and the successor function S are constructors, while addition *plus* and multiplication *mul* are not constructors. The explicit recognition of constructor symbols makes it possible to apply induction over such function symbols.

Function symbols are declared using the keywords **func** and **map**. With **func** one can declare constructors, which define the structure of the data type. For example,

sort *Bool*
func $\mathtt{T}, \mathtt{F} :\rightarrow \mathit{Bool}$

declares that `T` (true) and `F` (false) are (the only) elements of sort *Bool*. We say that `T` and `F` are the constructors of sort *Bool*.

As booleans will be used in the if-then-else construct in the process language, which will be presented in Chapter 3, the sort *Bool* with its constructors `T` and `F` must be declared in every μCRL specification.

The natural numbers can be declared using the constructors zero 0 and successor S:

sort *Nat*
func $0 :\rightarrow \mathit{Nat}$
$S : \mathit{Nat} \rightarrow \mathit{Nat}$

This says that each natural number can be written as the application of zero or more successor symbols to 0.

If for a sort D no constructor with target sort D is given, then it is assumed that D is an arbitrary data domain. This can be useful, for instance when defining a data transfer protocol that can transfer data elements from an arbitrary domain D. In such a case it suffices to declare

sort D

The keyword **map** is used to declare functions symbols that are not constructors. (The distinction between constructors and non-constructors in μCRL is essential for the implementation of the summation operator over a data type; see Section 3.7.) For instance, declaring conjunction $\wedge$ on the booleans, or declaring addition *plus* on natural numbers, can be done by adding the following lines to a specification, where *Nat* and *Bool* have already been declared:

map $\wedge : Bool \times Bool \to Bool$
$plus : Nat \times Nat \to Nat$

The meaning of such functions is defined by means of equations, called axioms. For example, the meaning of the two functions declared above is specified by the following equations:

var x:*Bool*
n, m:*Nat*
rew $x \wedge \mathtt{T} = x$
$x \wedge \mathtt{F} = \mathtt{F}$
$plus(n, 0) = n$
$plus(n, S(m)) = S(plus(n, m))$

The keyword **rew** refers to the fact that in the μCRL toolset, the axioms are applied as *rewrite rules* from left to right; see Section 2.2. Note that before each group of axioms one must declare with the keyword **var** the data variables that occur in these axioms.

The idea behind the two axioms for conjunction is that `T` and `F` are the only constructors of the sort *Bool*, so that the second argument of conjunction can in principle be equated to one of these two forms. Then the two axioms for conjunction make it possible to eliminate the occurrence of the conjunction symbol. Similarly, the idea behind the two axioms for *plus* is that 0 and S are the only two constructors of the sort *Nat*, so that the second argument of addition can in principle be equated to a data term $S^n(0)$, for some $n \geq 0$. The second axiom for addition peels off the successor symbols of the second argument. Finally, when this argument has become 0, the first axiom for addition makes it possible to eliminate the occurrence of the addition symbol.

The machine-readable syntax of the μCRL toolset only allows prefix function symbols in ASCII characters. For example, in this machine-readable syntax conjunction could read `and(b,b')` (instead of $b \wedge b'$). There is a precise syntax for μCRL that prescribes what specifications must look like in plain text, which can be found in the defining document of the language [59]. That

syntax is meant for specifications intended to be processed by a computer, in which case syntactic objects must be unambiguous for a parser. However, in this text we will freely declare function symbols infix or postfix, and in μCRL specifications we will use LaTeX type setting features at will, if this increases readability.

Function names may be overloaded, as long as for every function its name together with the list of sorts of its arguments is unique. For example, it is allowed to declare functions $label : Nat \to Nat$ and $label : Bool \to Nat$, but it is not allowed to declare functions $f : Nat \to Nat$ and $f : Nat \to Bool$.

For data terms like $head([])$, representing the head of the empty list (see Exercise 4 on the data type *List*), it is customary to introduce a special error element, and to equate $head([])$ to this error element. However, such error elements can significantly complicate the data types, without bringing much joy. We recommend, if possible, to refrain from including error elements in data types. In that case a data term like $head([])$ cannot be simplified, which will manifest itself as a bug in the overall μCRL specification. So if the mistake is made to apply *head* to the empty list, then this will automatically come to light in the formal analysis, as one would desire.

In μCRL, constructor terms are stored in a very compact way as so-called ATerms [26]. An ATerm consists of a head symbol and some parameters, which are again ATerms. They are stored by means of maximal sharing, meaning that each ATerm is stored only once, but may be referenced multiple times. Unreferenced ATerms are reclaimed by a garbage collector. Next to efficient use of memory, an additional advantage of ATerms is that equality checking between constructor terms reduces to a single comparison of references, instead of a traversal of the term structures. ATerms however may come at a price, as we will see in Section 6.7.

2.2 Term Rewriting

Term rewriting [104] provides a straightforward method for implementing algebraic specifications of abstract data types. A term rewriting system consists of rewrite rules $term \to term$, where the first term is not a single variable, and variables that occur in the second term also occur in the first term. (If one of these two restrictions is violated, the term rewriting system would for certain give rise to infinite computations, which is undesirable from an implementation point of view.) Intuitively, a rewrite rule is a directed equation that can only be applied from left to right.

Example 2. We direct the four equations for natural numbers (see Example 1) from left to right:

$$
\begin{aligned}
&1. && plus(x, 0) \to x \\
&2. && plus(x, S(y)) \to S(plus(x, y)) \\
&3. && mul(x, 0) \to 0 \\
&4. && mul(x, S(y)) \to plus(mul(x, y), x)
\end{aligned}
$$

Using these rewrite rules, we can reduce the data term $mul(S(0), S(S(0)))$ to its *normal form* $S(S(0))$, by the following sequence of rewrite steps. In each rewrite step, the subterm that is being reduced is underlined, and the number of the rewrite rule that is being applied is given above the arrow.

$$
\begin{aligned}
\underline{mul(S(0), S(S(0)))} &\xrightarrow{4} plus(\underline{mul(S(0), S(0))}, S(0)) \\
&\xrightarrow{4} plus(plus(\underline{mul(S(0), 0)}, S(0)), S(0)) \\
&\xrightarrow{3} plus(\underline{plus(0, S(0))}, S(0)) \\
&\xrightarrow{2} plus(S(\underline{plus(0, 0)}), S(0)) \\
&\xrightarrow{1} \underline{plus(S(0), S(0))} \\
&\xrightarrow{2} S(\underline{plus(S(0), 0)}) \\
&\xrightarrow{1} S(S(0))
\end{aligned}
$$

Ideally, each reduction of a term by means of a term rewriting system eventually leads to a normal form, which is built entirely from constructor symbols, so that it cannot be reduced any further (*termination*). Moreover, ideally each term can be reduced to no more than one normal form (*confluence*). Assuming a set of axioms, one can try to derive an equation $d = e$ by giving a direction to each of the axioms, to obtain a term rewriting system, and attempting to reduce d and e to the same normal form. If the resulting term rewriting system is terminating and confluent, then this procedure is guaranteed to return a derivation of the equation $d = e$ if such a derivation exists.

It can be the case that more than one rewrite rule can be applied to a term. In the toolset of μCRL, one of the applicable rewrite rules is selected (depending on the order in which the rewrite rules are given). Furthermore, it can be the case that several subterms of a term can be reduced by applications of rewrite rules. In the toolset of μCRL, *innermost* rewriting is used, meaning that a subterm is selected as close as possible to the leaves of the parse tree of the term. The implementation of innermost rewriting tends to be more efficient than *outermost* rewriting, which selects a subterm as close as possible to the root of the parse tree of the term.

2.3 Equality Functions

In μCRL one needs to specify an *equality* function $eq : D \times D \to Bool$ for data domains D, reflecting equality between data terms of sort D. (Actually, such an equality function is only needed for data types that are used as data parameters of actions that occur in a communication; see Section 3.3.) That is, $eq(d, e) = \mathtt{T}$ if $d = e$ and $eq(d, e) = \mathtt{F}$ if $d \neq e$, for all $d, e \in D$. For example, for the booleans one could define an equality function as follows.

rew $eq(\mathtt{T}, \mathtt{T}) = \mathtt{T}$
$eq(\mathtt{F}, \mathtt{F}) = \mathtt{T}$
$eq(\mathtt{T}, \mathtt{F}) = \mathtt{F}$
$eq(\mathtt{F}, \mathtt{T}) = \mathtt{F}$

In the case of the natural numbers it is no longer possible to define (in)equality for all separate elements, as there are infinitely many of them. In this case one can define a function eq using the fact that $S(n) = S(m)$ if and only if $n = m$.

var $n, m{:}Nat$
rew $eq(0, 0) = \mathtt{T}$
$eq(S(n), 0) = \mathtt{F}$
$eq(0, S(n)) = \mathtt{F}$
$eq(S(n), S(m)) = eq(n, m)$

2.4 Induction

We show how one can use induction to derive equalities between data terms. The crux is that we may assume that every data term can be equated to a data term containing only constructor symbols.

A typical example of induction on booleans is the following derivation of $b \wedge b = b$ for booleans b, from the two axioms for conjunction that were given in Section 2.1: $x \wedge \mathtt{T} = x$ and $x \wedge \mathtt{F} = \mathtt{F}$. By induction it suffices to prove that the equation $b \wedge b = b$ can be derived for the constructors $b = \mathtt{T}$ and $b = \mathtt{F}$. In other words, we must show that $\mathtt{T} \wedge \mathtt{T} = \mathtt{T}$ and $\mathtt{F} \wedge \mathtt{F} = \mathtt{F}$. These are simply instances of the defining axioms for $\wedge$ mentioned before.

As a second example, suppose that we have declared the natural numbers with constructors 0 and S as in Example 1. We can for instance derive by induction that $plus(0, k) = k$ for all natural numbers k. First we consider the base case $k = 0$, meaning that we must derive $plus(0, 0) = 0$. This is an instance of the first axiom on addition. Second we consider the inductive case that k has the form $S(k')$, meaning that we must derive $plus(0, S(k')) = S(k')$. As k' is smaller than k, in this case we may assume that the property to be proved holds for k', i.e., $plus(0, k') = k'$. Then we obtain:

$$plus(0, S(k')) = S(plus(0, k')) = S(k')$$

Exercises

Exercise 1. Declare the functions disjunction $\vee$, negation $\neg$, implication $\Rightarrow$ and bi-implication $\Leftrightarrow$ on the booleans, and provide equations for them.

Exercise 2. Give algebraic specifications of 'greater than or equal' $\geq$, 'greater than' $>$, the *power* function (with $power(m, n) = m^n$), the cut-off minus function $\dot{-}$ (with $n \dot{-} m = 0$ if $n \leq m$), and the function *even* (with $even(n) =$ `T` if and only if n is an even number) on the natural numbers.

Exercise 3. Provide equations for $divides : Nat \times Nat \to Bool$, where $divides(m, n)$ returns `T` if and only if m divides n.

Exercise 4. Given a sort *List* over an arbitrary non-empty data domain D, with as constructors the empty list $[] :\to List$, and $in : D \times List \to List$ to insert an element from D at the beginning of a list. Provide equations for the following non-constructor functions: $head : List \to D$ and $toe : List \to D$ to obtain the first and the last element of a non-empty list, respectively; $tail : List \to List$ and $untoe : List \to List$ to remove the first and the last element from a list, respectively; $+\!+ : List \times List \to List$ to concatenate two lists; $append : D \times List \to List$ to insert an element from D at the end of a list; $nonempty : List \to Bool$ to check whether a list is non-empty; and $length : List \to Nat$ to compute the length of a list.

Exercise 5. Assuming an equality function *eq* on the data type D, provide equations for an equality function *eq* on the data type *List* of lists over D.

Exercise 6. Given is a data domain D with equality function $eq : D \times D \to Bool$. The data type *Set* of *sets of elements from* D is specified by the empty set $empty\text{-}set :\to Set$ and the inclusion function $in : D \times Set \to Set$, specified by:

$$in(d, in(e, \sigma)) = in(e, in(d, \sigma))$$
$$in(d, in(d, \sigma)) = in(d, \sigma)$$

Unlike lists, sets disregard the order in which elements have been included or whether elements have been included multiple times.

1. Specify the function $eq : Set \times Set \to Bool$ that checks for equality of sets. You may want to specify help functions like $element : D \times Set \to Bool$ to test whether an element is in a set, or $remove : D \times Set \to Set$ to remove an element from a set.
2. Consider now the data type of *multi-sets* over D, which take into account the number of occurrences of an element, meaning that the second equation above is skipped. Adapt your specification to obtain an equality function for multi-sets.

Exercise 7. In order to reduce the state space generated by a process term (see Chapter 3), it is advisable to represent message buffers and messages in

channels as *ordered* lists. Suppose that a total order $<: D \times D \to \mathit{Bool}$ has been imposed. Specify a non-constructor function $\mathit{add} : D \times \mathit{List} \to \mathit{List}$ that, given a datum d and a sorted list λ, outputs a sorted list, by placing d at the right place in λ.

Exercise 8. Prove by induction: $b \vee b = b$ and $\neg\neg b = b$, for all booleans b.

Exercise 9. Prove by induction that:

1. $\mathtt{F} \wedge b = \mathtt{F}$
2. $b \Rightarrow \mathtt{T} = \mathtt{T}$
3. $b \Rightarrow \mathtt{F} = \neg b$
4. $b_1 \Rightarrow b_2 = \neg b_2 \Rightarrow \neg b_1$
5. $b \Leftrightarrow \mathtt{T} = b$
6. $b \Leftrightarrow b = \mathtt{T}$
7. $b_1 \Leftrightarrow \neg b_2 = \neg(b_1 \Leftrightarrow b_2)$
8. $(b_1 \vee b_2) \Leftrightarrow b_1 = b_1 \vee \neg b_2$
9. $\mathit{even}(\mathit{plus}(k, \ell)) = \mathit{even}(k) \Leftrightarrow \mathit{even}(\ell)$
10. $\mathit{even}(\mathit{mul}(k, \ell)) = \mathit{even}(k) \vee \mathit{even}(\ell)$

Exercise 10. Prove by induction that:

1. $\mathit{plus}(0, k) = k$
2. $\mathit{mul}(0, k) = 0$
3. $\mathit{plus}(\mathit{plus}(k, \ell), m) = \mathit{plus}(k, \mathit{plus}(\ell, m))$
4. $\mathit{mul}(k, \mathit{plus}(\ell, m)) = \mathit{plus}(\mathit{mul}(k, \ell), \mathit{mul}(k, m))$
5. $\mathit{mul}(\mathit{mul}(k, \ell), m) = \mathit{mul}(k, \mathit{mul}(\ell, m))$
6. $\mathit{mul}(\mathit{power}(m, k), \mathit{power}(m, \ell)) = \mathit{power}(m, \mathit{plus}(k, \ell))$

Exercise 11. Describe the concatenation of a list ℓ from which the last element has been removed to a list ℓ' into which the last element of ℓ has been inserted. Prove, using your equations from Exercise 4 and induction, that if ℓ is non-empty, this concatenation is equal to the concatenation of ℓ and ℓ'.

3

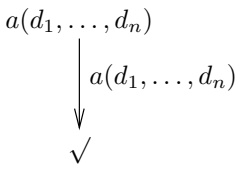
Process Algebra

This chapter presents the process operators that form the heart of the specification language μCRL. Its framework consists of process algebra, for describing system behaviour, enhanced with abstract data types, which were explained in the previous chapter.

Similar to abstract data types, the meaning of each process operator in the language is captured by means of a number of axioms. These axioms form an elementary basis for equational reasoning about processes.

3.1 Actions

Actions represent atomic events in the real world. An action consists of an action name followed by zero or more data arguments. Intuitively, an action $a(d_1, \ldots, d_n)$ can execute itself, after which it terminates successfully:



Such an expression is called a *transition*. The symbol $\surd$ in the transition above represents successful termination after the execution of $a(d_1, \ldots, d_n)$.

In μCRL, actions are declared using the keyword **act**, followed by an action name and the sorts of its *data parameters*. If an action name a does not carry data parameters, then $a()$ is abbreviated to a. The set of all action names that are declared in a μCRL specification is denoted by Act. As an example, below is declared the action name *time-out* without data parameters, and the action name *send* that is parametrised by a pair of a data element of sort D and a natural number:

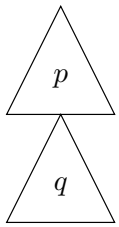
act *time-out*
send : $D \times Nat$

In μCRL the data types of an action name need not be unique. That is, it is allowed to declare an action name more than once, as long as these declarations all carry different data types. For example, one could declare an action name a with a single sort Nat and with a pair of sorts $D \times Bool$.

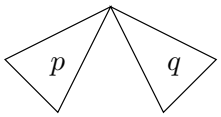
We proceed to introduce a number of *operators* to express process behaviour. The process terms that can be built from these operators together with the action names represent system behaviour. The operators give rise to transitions between process terms: these transitions are labelled by actions. Thus each process term constitutes a *system state*.

3.2 Alternative and Sequential Composition

Two elementary operators to construct process terms are the *sequential composition* operator, written $p{\cdot}q$, and the *alternative composition* operator, written $p + q$. The process term $p{\cdot}q$ first executes p, until p terminates, and then continues with executing q. In other words, the *state space* of $p{\cdot}q$ is obtained by replacing each successful termination transition $r \xrightarrow{a} \surd$ in the state space of p by $r \xrightarrow{a} q$:



The process term $p + q$ behaves as p or q, depending on which of these two arguments performs the first action. In other words, the state space of $p + q$ is obtained by joining the state spaces of p and q at their initial states:



In general, the link between a process term p and its transitions $p \xrightarrow{a} p'$ and $p \xrightarrow{a} \surd$ can be formally defined by means of a *structural operational semantics* [2, 92]. This consists of giving some inductive proof rules, called *transition rules*, for all process operators. For example, in case of actions, the alternative and the sequential composition operator, the transition rules are as follows (where $\mathbf{d}$ abbreviates a sequence of data terms $d_1, \ldots, d_n$):

$$\frac{}{a(\mathbf{d}) \overset{a(\mathbf{d})}{\to} \surd}$$

$$\frac{x_1 \overset{a(\mathbf{d})}{\to} \surd}{x_1 + x_2 \overset{a(\mathbf{d})}{\to} \surd} \qquad \frac{x_1 \overset{a(\mathbf{d})}{\to} y}{x_1 + x_2 \overset{a(\mathbf{d})}{\to} y} \qquad \frac{x_2 \overset{a(\mathbf{d})}{\to} \surd}{x_1 + x_2 \overset{a(\mathbf{d})}{\to} \surd} \qquad \frac{x_2 \overset{a(\mathbf{d})}{\to} y}{x_1 + x_2 \overset{a(\mathbf{d})}{\to} y}$$

$$\frac{x_1 \overset{a(\mathbf{d})}{\to} \surd}{x_1 \cdot x_2 \overset{a(\mathbf{d})}{\to} x_2} \qquad \frac{x_1 \overset{a(\mathbf{d})}{\to} y}{x_1 \cdot x_2 \overset{a(\mathbf{d})}{\to} y \cdot x_2}$$

We will refrain from spelling out the transition rules for all the operators, and limit ourselves to explaining in an intuitive fashion the operational semantics of each operator. The reader is referred to [59] for a complete overview of the transition rules underlying the μCRL process operators. And in [46], transition rules of many of the core process algebraic operators are given, together with a description of how one can obtain *congruence* results for free, by inspecting the syntactic form of the transition rules. An equivalence relation on the process terms is a congruence if it is respected by all the operators. We will come back to this issue in Sections 3.10 and 4.4.

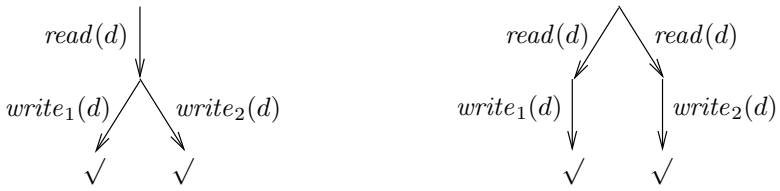
Table 3.1. Axioms for alternative and sequential composition

A1	$x + y = y + x$
A2	$x + (y + z) = (x + y) + z$
A3	$x + x = x$
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$

In Table 3.1 axioms A1–5 are listed, describing the elementary properties of the alternative and sequential composition operators. In these axioms we use process variables x, y and z that can be instantiated by process terms. Similar as for data terms (see Section 2.1), the equality relation on process terms is obtained by applying to the axioms all possible substitutions of process terms for process variables, and closing the relation under contexts and equivalence. The axioms A1, A2 and A3 express that $+$ is commutative, associative and idempotent, A4 expresses that $+$ is right-distributive, and A5 expresses that $\cdot$ is associative.

Note that $+$ is not left-distributive, i.e., in general $x{\cdot}(y+z) \neq x{\cdot}y + x{\cdot}z$. In a distributed setting, left-distributivity of $+$ breaks down.

Example 3. Consider the two state spaces below:



The first initial state reads datum d, and then decides whether it writes d on disc 1 or on disc 2. The second initial state makes a choice for disc 1 or disc 2 before it reads datum d. The initial states of both state spaces display the same strings of actions, $read(d){\cdot}write_1(d)$ and $read(d){\cdot}write_2(d)$. Still, there is a crucial distinction between the two initial states, which becomes apparent if for instance disc 1 crashes. In that case the first initial state always saves datum d on disc 2, while the second initial state may get into a deadlock (i.e., may get stuck); see Section 3.4 for a formal definition of such deadlocks.

In Definition 1, we will introduce an equivalence relation on the states in a state space, which distinguishes states where the state space has a different branching structure. This means that even states that exhibit the same traces are not always equivalent.

As binding convention we assume that the $\cdot$ binds stronger than the $+$. For example, $a{\cdot}b + a{\cdot}c$ represents $(a{\cdot}b) + (a{\cdot}c)$. The sequential composition operator in process terms will often be omitted. That is, pq denotes $p{\cdot}q$.

p and q are called *summands* of $p+q$; moreover, all summands of p and q are also summands of $p+q$. We use the shorthand $x \subseteq y$ for $x+y=y$, and write $x \supseteq y$ for $y \subseteq x$. The derivation of an equation from the axioms can be divided into proving two of such summand inclusions (see Exercise 14).

3.3 Parallel Processes

A binary operator called *merge* can be used to put process terms in parallel. The behaviour of $p \parallel q$ is the arbitrary interleaving of actions of the arguments p and q. For example, if there is no communication possible between the action names a and b, then the process term $a \parallel b$ behaves as $a{\cdot}b + b{\cdot}a$.

Moreover, it is possible to let the process terms p and q in $p \parallel q$ *communicate*, by declaring in a communication section that certain action names in p and q can synchronise to some other action name. Typically,

comm $a \mid b = c$

Suppose that two actions $a(\mathbf{d})$ and $b(\mathbf{d})$ can happen in parallel (where $\mathbf{d}$ abbreviates a sequence of data terms $d_1, \ldots, d_n$). Then the communication

declaration implies that they may synchronise; this communication is denoted by $c(\mathbf{d})$. For example, the process term $a \parallel b$ now behaves as $a{\cdot}b+b{\cdot}a+c$. Two actions can only synchronise if their data parameters are exactly the same. In the communication declaration above it is required implicitly that the action names a, b and c have been declared with exactly the same data parameters.

The equality function $eq : D \times D \to Bool$, with $eq(d,e) = \mathtt{T}$ if and only if $d = e$ (see Section 2.3), is needed for data types D that are used as data parameters of actions that occur in a communication. For example, if $a \mid b = c$, then in the μCRL toolset an expression $a(d) \mid b(e)$ is transformed into $c(d) \lhd eq(d,e) = \mathtt{T} \rhd \delta$.

Communication between actions is commutative and associative:

$$a \mid b = b \mid a$$
$$(a \mid b) \mid c = a \mid (b \mid c)$$

The commutative counterpart $b \mid a = c$ of the communication declaration above is generated automatically within the μCRL toolset. It is up to the user to specify communication in such a way that it is associative; the μCRL toolset does check whether a communication declaration is indeed associative.

Example 4. Let the communication of two actions from $\{a, b, c\}$ always result in c. The state space of the process term $(a{\cdot}b) \parallel (b{\cdot}a)$ is depicted in Fig. 3.1.

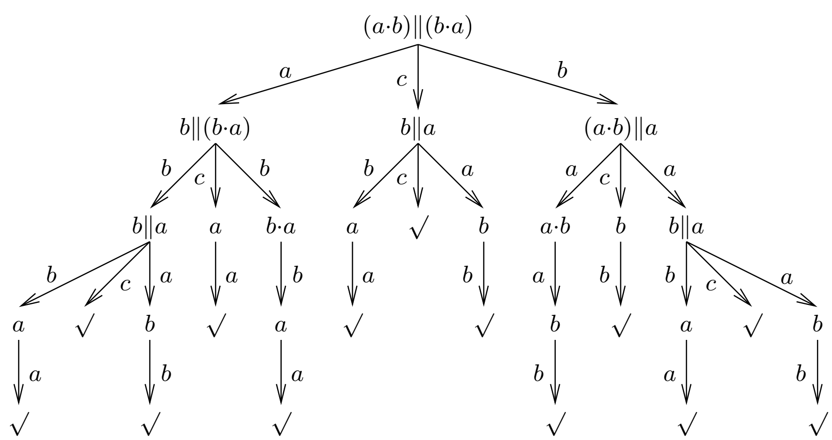


Fig. 3.1. State space of $(a{\cdot}b) \parallel (b{\cdot}a)$

Example 4 shows that the merge of two simple process terms produces a relatively large state space. The state space tends to grow in an exponential fashion with respect to the number of concurrent components of a distributed system; this is known as the state explosion problem. This partly explains the strength of an algebraic theory of communicating processes, as such a theory

Table 3.2. Axioms for parallelism

CM1	$x \parallel y = (x \between y + y \between x) + x \mid y$	
CM2	$a(\mathbf{d}) \between x = a(\mathbf{d}) \cdot x$	
CM3	$a(\mathbf{d}) \cdot x \between y = a(\mathbf{d}) \cdot (x \parallel y)$	
CM4	$(x + y) \between z = x \between z + y \between z$	
CF	$a(\mathbf{d}) \mid b(\mathbf{d}) = c(\mathbf{d})$	if $a \mid b = c$
CF′	$a(\mathbf{d}) \mid b(\mathbf{e}) = \delta$	if $\mathbf{d} \neq \mathbf{e}$ or a and b do not communicate
CM5	$a(\mathbf{d}) \cdot x \mid b(\mathbf{e}) = (a(\mathbf{d}) \mid b(\mathbf{e})) \cdot x$	
CM6	$a(\mathbf{d}) \mid b(\mathbf{e}) \cdot x = (a(\mathbf{d}) \mid b(\mathbf{e})) \cdot x$	
CM7	$a(\mathbf{d}) \cdot x \mid b(\mathbf{e}) \cdot y = (a(\mathbf{d}) \mid b(\mathbf{e})) \cdot (x \parallel y)$	
CM8	$(x + y) \mid z = x \mid z + y \mid z$	
CM9	$x \mid (y + z) = x \mid y + x \mid z$	

makes it possible to draw conclusions about the full system by studying its separate concurrent components.

The axioms for the merge are presented in Table 3.2. In order to formulate the axioms, two auxiliary operators are introduced. The *left merge* $\between$ is a binary operator that behaves exactly as the merge, except that its first action must come from the left-hand side. The *communication merge* $\mid$ is also a binary operator behaving as the merge, except that the first action must be a synchronisation between its left- and right-hand side. As binding convention we assume that the $\parallel$, $\between$ and $\mid$ bind stronger than the $+$, and weaker than $\cdot$. For instance, $a \cdot b \between c + a \parallel b \cdot c$ represents $((a \cdot b) \between c) + (a \parallel (b \cdot c))$.

The core axiom for the merge is CM1 in Table 3.2. It says that in $x \parallel y$, either x performs the initial transition, represented by the summand $x \between y$, or y performs the initial transition, represented by $y \between x$, or the initial transition of $x \parallel y$ is a communication between initial transitions of x and y, represented by $x \mid y$. All other axioms in Table 3.2 are designed to eliminate occurrences of the left merge and the communication merge in favour of the alternative and the sequential composition.

Axiom CF′ features the special deadlock action δ, which does not display any behaviour. It will be explained in the next section. CF′ expresses that $a(\mathbf{d}) \mid b(\mathbf{e})$ does not display any behaviour if either a and b do not communicate, or $\mathbf{d}$ and $\mathbf{e}$ are distinct. We write $\mathbf{d} = \mathbf{e}$ if these lists have equal length n and $d_i = e_i$ for $i = 1, \ldots, n$.

3.4 Deadlock and Encapsulation

If two action names are able to communicate, then often we only want these action names to occur in communication with each other, and not on their own. For example, let the action $send(d)$ represent sending a datum d into one end of a channel, while $read(d)$ represents receiving this datum at the other end of the channel. Furthermore, let the synchronous communication of these two actions result in transferring the datum d through the channel by the action $comm(d)$. For the outside world, the actions $send(d)$ and $read(d)$ never appear on their own, but only in communication in the form $comm(d)$.

In order to enforce synchronous communication, we introduce a special action name δ called *deadlock*, which does not display any behaviour. This reserved action name is not in Act, it does not carry any data parameters, and it cannot communicate with any action name. Typical properties of δ are:

- $p + \delta = p$: the choice in an alternative composition is determined by the first actions of its arguments, and therefore one can never choose for a summand δ;
- $\delta \cdot p = \delta$: as sequential composition takes its first action from its first argument, $\delta \cdot p$ cannot perform any actions.

These equalities constitute the two defining axioms A6 and A7 for the deadlock in Table 3.3.

Sometimes we want to express that certain actions cannot happen and must be blocked, i.e., must be renamed to δ. Typically, by renaming the action names *send* and *read* into δ, actions $send(d)$ and $read(d)$ can no longer occur on their own, but only in their synchronous communication $comm(d)$. The unary *encapsulation* operators ∂_H, for subsets H of Act, are especially designed for this task. A process term $\partial_H(p)$ can execute all actions of p of which the names are not in H. Typically, $\partial_{\{b\}}(a \cdot b(3) \cdot c) = a \cdot \delta$. A more elaborate example of the use of the encapsulation operator is presented in Example 5. The axioms for deadlock and encapsulation are listed in Table 3.3.

Example 5. Suppose a datum 0 or 1 is sent into a channel, which is expressed by the process term $send(0) + send(1)$. Let this datum be received at the other side of the channel, which is expressed by the process term $read(0) + read(1)$. The communication of $send(d)$ and $read(d)$ results in $comm(d)$, for $d \in \{0, 1\}$, while all other communications between actions result in δ. The behaviour of the channel is described by the process term

$$\partial_{\{send,\, read\}}((send(0) + send(1)) \parallel (read(0) + read(1)))$$

The encapsulation operator enforces that the action $send(d)$ can only occur in communication with the action $read(d)$, for $d \in \{0, 1\}$. So the process term above can only perform $comm(0)$ or $comm(1)$ and terminate successfully, as desired.

Table 3.3. Axioms for deadlock and encapsulation

A6	$x + \delta = x$	
A7	$\delta \cdot x = \delta$	
DD	$\partial_H(\delta) = \delta$	
D1	$\partial_H(a(\mathbf{d})) = a(\mathbf{d})$	if $a \notin H$
D2	$\partial_H(a(\mathbf{d})) = \delta$	if $a \in H$
D3	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$	
D4	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$	
CD1	$\delta \between x = \delta$	
CD2	$\delta \mid x = \delta$	
CD3	$x \mid \delta = \delta$	

Beware not to confuse a transition of the form $s \xrightarrow{a} \delta$ with a transition of the form $s \xrightarrow{a} \surd$; intuitively, the first transition expresses that state s gets stuck after the execution of action a, while the second transition expresses that s terminates successfully after the execution of action a. To mark this distinction, we introduce a special predicate $\downarrow$ on states, to express successful termination; $\surd$ is the only state where $\downarrow$ holds (see also Definition 1). A state s is said to contain a deadlock if there is an execution sequence $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ where $s_n \not\downarrow$ and s_n cannot perform any actions. In general it is undesirable that a process contains a deadlock, because it represents that the system gets stuck without producing any output. Experience learns that specifications of distributed systems often contain a deadlock. For example, the third sliding window protocol in [102] contains a deadlock; see [51, *Stelling 7*]. It can, however, be very difficult to detect such a deadlock, even if one has a good insight into the design of such a system.

If one state contains a deadlock while another does not, then these states exhibit different behaviour. In Section 3.10 we will introduce an equivalence relation on states, called *bisimilarity*, that distinguishes such states. In particular, it distinguishes the process terms $a \cdot b + a \cdot c$ and $a \cdot (b + c)$. Namely, as $a \cdot \delta + a \cdot c$ contains a deadlock, it is not bisimilar to the deadlock-free process term $a \cdot c$. In other words, $\partial_{\{b\}}(a \cdot b + a \cdot c)$ is not bisimilar to $\partial_{\{b\}}(a \cdot (b + c))$. Since bisimilarity is a congruence, meaning that it is preserved under contexts (see Section 3.10), this implies that $a \cdot b + a \cdot c$ and $a \cdot (b + c)$ are not bisimilar.

Asynchronous communication between two system components A and B can be modelled by placing between them another component C, representing the channel between A and B. By letting both A and B communicate synchronously with C, and allowing C to store data elements in a buffer, one obtains asynchronous communication between A and B. In the protocol specifications that will be presented in Chapter 5, asynchronous communication will be modelled in this way.

3.5 Process Declarations

The heart of a μCRL specification is the *process declaration* section, marked with **proc**, where the behaviour of the system is declared. This section consists of recursive equations of the following form, for $n \geq 0$:

proc $\quad X(x_1{:}D_1, \ldots, x_n{:}D_n) = p$

Here X is a *recursion variable*, the x_i are data variables, and the D_i are sort names, expressing that the data parameters x_i are of sort D_i. Moreover, p is a process term possibly containing occurrences of expressions $Y(d_1, \ldots, d_m)$, where Y is a recursion variable and the d_i are data terms that may contain occurrences of the data variables $x_1, \ldots, x_n$. Intuitively, in this recursive equation, $X(x_1, \ldots, x_n)$ is declared to have the same (potential) behaviour as the process term p. For instance, given the recursive equation $X = a{\cdot}X$; the process that can only perform an infinite sequence of a-transitions is a *solution* for the recursion variable X.

The recursive equations are to be *guarded*, meaning that an occurrence of an expression $Y(d_1, \ldots, d_m)$ in the right-hand side of a recursive equation is always proceeded by an action. In other words, such an expressions always occurs in a context of the form $C_1[a(\mathbf{d}){\cdot}C_2[]]$. Unguarded recursive equations such as $Z = Z + a$ and $Z = Z{\cdot}a$ are considered meaningless, as they do not determine the entire initial behaviour of the recursion variable Z; as a result, such a process declaration does not specify a unique process.

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

init $\quad X(d_1, \ldots, d_n)$

$X(d_1, \ldots, d_n)$ represents the initial behaviour of the system that is being described. In general, in μCRL specifications the **init** section is used to instantiate the data parameters of a process declaration, meaning that the d_i are data terms that do not contain data variables.

Example 6. The process declaration below specifies a clock process, which repeatedly performs the action *tick* or displays the current time. (The specification of the data types is omitted.)

act $tick$
$display : Nat$
proc $Clock(n{:}Nat) = tick \cdot Clock(S(n)) + display(n) \cdot Clock(n)$
init $Clock(0)$

3.6 Conditionals

The process term $p \lhd b \rhd q$, where b is a data term of sort $Bool$, behaves as p if b is equal to true, and behaves as q if b is equal to false. This operator, called the *conditional* operator, binds stronger than $+$ and weaker than $\cdot$. Using the conditional operator, data can influence process behaviour. The conditional operator is characterised by axioms C1 and C2 in Table 3.4.

Table 3.4. Axioms for conditionals

C1	$x \lhd \mathtt{T} \rhd y = x$
C2	$x \lhd \mathtt{F} \rhd y = y$

Example 7. We specify a stopwatch that counts down, and issues a time-out message followed by a clock reset when the time has become zero.

act $tick$, $time\text{-}out$, $reset$
proc $Clock(n{:}Nat) = tick \cdot Clock(n \dot{-} S(0)) \lhd n > 0 \rhd time\text{-}out \cdot Reset$
$Reset = \sum_{m:Nat}(reset \cdot Clock(m) \lhd m > 0 \rhd \delta)$
init $Reset$

3.7 Summation over a Data Type

From now on, process terms are considered modulo associativity of the $+$, meaning that we do not care to write brackets for terms of the form $p_1{+}p_2{+}p_3$. This is allowed in view of axiom A2 in Table 3.1.

The *sum* operator $\sum_{d:D} P(d)$, with $P(d)$ a mapping from the data type D to process terms, and d a data parameter, behaves as $P(d_1) + P(d_2) + \cdots$, i.e., as the possibly infinite choice between $P(d)$ for any datum d of sort D.

Example 8. The following process declaration specifies a single-place buffer, repeatedly receiving a natural number n using action name r (shorthand for *read*), and then delivering that value via action name s (shorthand for *send*).

proc $\quad Buffer = \sum_{n:Nat} r(n)\cdot s(n)\cdot Buffer$

Table 3.5. Axioms for summation

SUM1	$\sum_{d:D} x = x$
SUM2	$\sum_{d:D} P(d) = \sum_{d:D} P(d) + P(d_0) \qquad (d_0 \in D)$
SUM3	$\sum_{d:D}(P(d) + Q(d)) = \sum_{d:D} P(d) + \sum_{d:D} Q(d)$
SUM4	$(\sum_{d:D} P(d))\cdot x = \sum_{d:D}(P(d)\cdot x)$
SUM5	$(\sum_{d:D} P(d)) \between x = \sum_{d:D}(P(d) \between x)$
SUM6	$(\sum_{d:D} P(d)) \mid x = \sum_{d:D}(P(d) \mid x)$
SUM6′	$x \mid (\sum_{d:D} P(d)) = \sum_{d:D}(x \mid P(d))$
SUM7	$\partial_H(\sum_{d:D} P(d)) = \sum_{d:D} \partial_H(P(d))$
SUM8	$(\forall e{\in}D\ P(e) = Q(e)) \Rightarrow \sum_{d:D} P(d) = \sum_{d:D} Q(d)$

In Table 3.5 the axioms for the sum operator are listed. As before the process variable x in the axioms may be instantiated with process terms; P and Q represent functions from some data type D to process terms.

The sum operator $\sum_{d:D} P(d)$ is a conceptually difficult operator, because it acts as a binder for the data parameter d. For example, a data variable d occurs *free* in the process term $a(d)$, while it occurs *bound* in the process term $\sum_{d:D} a(d)$. We allow α-*conversion* (i.e., renaming of bound occurrences of variables) in the sum operator. Hence, we consider the process terms $\sum_{d:D} P(d)$ and $\sum_{e:D} P(e)$ as equal (under the assumption that e and d do not occur free in $P(d)$ and $P(e)$, respectively). See [79] for a thorough treatment of this binding construct.

When substituting a process term p for a process variable x in the axioms, p is not allowed to contain free occurrences of data variables. For example, we cannot substitute the process term $a(d)$ for x in the left-hand side of SUM1 in Table 3.5. SUM1 says that since the data parameter d does not appear in p, we may omit the sum operator from $\sum_{d:D} p$. SUM2 allows one to split an instance of the summand from a sum. SUM3 says that one may distribute the sum operator over an alternative composition. SUM4 expresses that a process term without process variables can be moved in- and outside the scope of any sum. SUM5-7 deal with the interplay of the sum operator with the left merge, the communication merge, and the encapsulation operator. Finally, SUM8

expresses that two sums are equal if all instantiations of their arguments are equal.

The distinction between constructors and non-constructors in μCRL is essential for the implementation of summation in the μCRL toolset. Namely, $\sum_{d:D}$ sums over all data terms d that can be built from the functions that have been declared in the **func** section of the specification of sort D (see Section 2.1).

3.8 An Example: The Bag

We specify a process that can put elements of a data type D into a bag, and subsequently collect these data elements from the bag in arbitrary order. This example stems from [13]. The action $in(d)$ represents putting datum d into the bag, and the action $out(d)$ represents collecting the datum d from the bag. All communications between actions result in δ. Initially the bag is empty, so that one can only put a datum into the bag. The state space in Fig. 3.2 depicts the behaviour of the bag over $\{0,1\}$, with the initial state placed in the leftmost uppermost corner.

The bag over a data type D can be specified by the following recursive equation, using the merge $\|$:

act $in, out : D$
proc $Bag = \sum_{d:D} in(d) \cdot (Bag \parallel out(d))$

Note that in the case that D is $\{0,1\}$, the process term Bag represents the bag over $\{0,1\}$ as depicted in Fig. 3.2. Namely, Bag executes $in(d)$ for some $d \in \{0,1\}$. The subsequent process term $Bag \parallel out(d)$ can put elements 0 and 1 in the bag and take them out again (by means of the concurrent component Bag), or it can at any time take the initial element d out of the bag (by means of the parallel component $out(d)$).

3.9 Renaming

Sometimes it is efficient to reuse a given specification with different action names. The unary renaming operators ρ_f, with $f : \mathsf{Act} \to \mathsf{Act}$, are suited for this purpose. The subscript f signifies that the action name a is renamed to $f(a)$. Actions a and $f(a)$ must carry the same data types. The process term $\rho_f(p)$ behaves as p with its action names renamed according to f. An equational characterisation of the renaming operator can be found in Table 3.6.

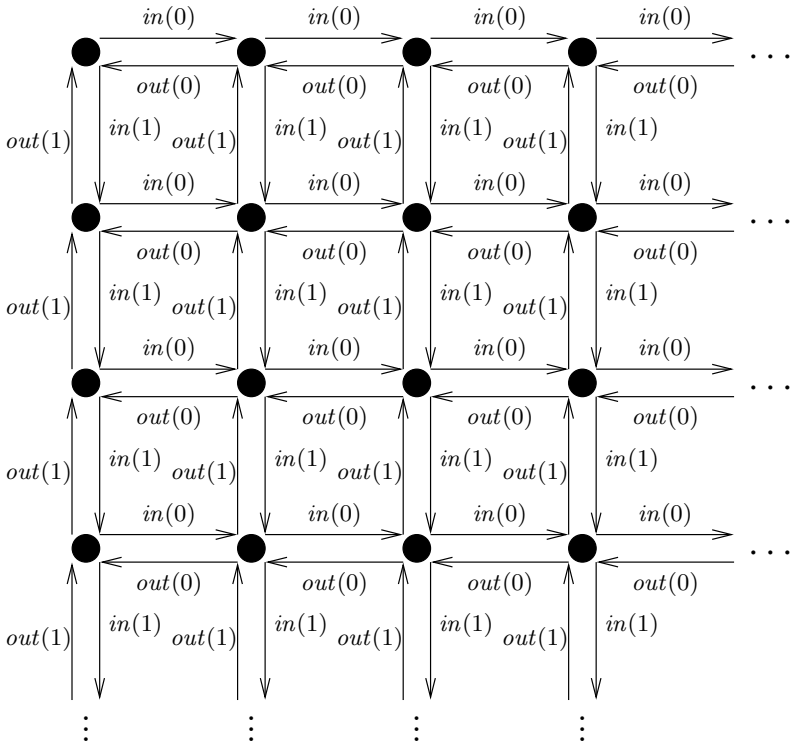


Fig. 3.2. State space of the bag over $\{0, 1\}$

Table 3.6. Axioms for renaming

R1	$\rho_f(\delta) = \delta$
R3	$\rho_f(a(\mathbf{d})) = f(a)(\mathbf{d})$
R4	$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$
R5	$\rho_f(x{\cdot}y) = \rho_f(x){\cdot}\rho_f(y)$
SUM9	$\rho_f(\sum_{d:D} P(d)) = \sum_{d:D} \rho_f(P(d))$

3.10 Bisimilarity

In the process algebraic framework defined in the previous sections, two levels can be distinguished. On the one hand there are the process terms, which can be manipulated by means of the axioms. Techniques from equational logic

and automated support from theorem provers can be used in the derivation of equations. On the other hand there are the state spaces that are attached to these process terms. Several techniques exist to minimise and analyse such state spaces. While on the level of process terms we have defined an equality relation, we have not yet introduced a way to relate the states in a state space.

Processes have been studied since the early 1960s, first to settle questions in natural languages, later on to study the semantics of programming languages. These studies were in general based on so-called trace equivalence, in which two states in a state space are said to be equivalent if they can execute exactly the same strings of actions. However, for distributed systems this equivalence is not satisfactory, which was shown in Example 3.

Bisimilarity [7, 85, 90] discriminates more states than trace equivalence. Namely, if two states are bisimilar, then not only they can execute exactly the same strings of actions, but also they have the same branching structure.

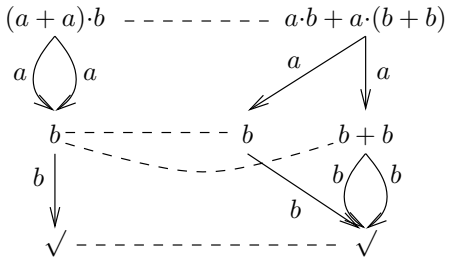
We recall that the predicate $\downarrow$ on states expresses successful termination; $\surd$ is the only state where $\downarrow$ holds (see Section 3.4).

Definition 1 (Bisimilarity). *Given a state space. A* bisimulation relation $\mathcal{B}$ *is a symmetric binary relation on states such that:*

1. *if* $s_1 \mathcal{B} s_2$ *and* $s_1 \stackrel{a(\mathbf{d})}{\rightarrow} s_1'$*, then there is a transition* $s_2 \stackrel{a(\mathbf{d})}{\rightarrow} s_2'$ *with* $s_1' \mathcal{B} s_2'$*; and*
2. *if* $s_1 \mathcal{B} s_2$ *and* $s_1 \downarrow$*, then* $s_2 \downarrow$*.*

Two states s_1 *and* s_2 *are* bisimilar*, denoted by* $s_1 \leftrightarrows s_2$*, if there is a bisimulation relation* $\mathcal{B}$ *such that* $s_1 \mathcal{B} s_2$*.*

Example 9. $(a + a)\cdot b \leftrightarrows a\cdot b + a\cdot(b + b)$.
A bisimulation relation that relates these two process terms is defined by $(a + a)\cdot b \,\mathcal{B}\, a\cdot b + a\cdot(b + b)$, $b \,\mathcal{B}\, b$, and $b \,\mathcal{B}\, b + b$. This bisimulation relation can be depicted as follows:



Without proof we note some important facts about bisimilarity. It is not hard to see that bisimilarity is an *equivalence* relation on states (meaning that it is reflexive, symmetric and transitive). Moreover, bisimilarity is a *congruence* with respect to the operators in μCRL, meaning that if $p \leftrightarrows q$, then $C[p] \leftrightarrows C[q]$ for all contexts $C[]$. Last but not least, bisimilarity is *sound* with respect to the equality relation on process terms, in the sense that if two process terms can be equated, then they are bisimilar.

Exercises

Exercise 12. Specify the process that first executes $a(d)$, and then $b(stop, \mathtt{F})$ or c. Also specify the process that executes either $a(d)$ followed by $b(stop, \mathtt{F})$, or $a(d)$ followed by c.

Exercise 13. Derive the following three equations from A1–5:

1. $((a + a)\cdot(b + b))\cdot(c + c) = a\cdot(b\cdot c)$
2. $(a + a)\cdot(b\cdot c) + (a\cdot b)\cdot(c + c) = (a\cdot(b + b))\cdot(c + c)$
3. $((a + b)\cdot c + a\cdot c)\cdot d = (b + a)\cdot(c\cdot d)$

Exercise 14. Prove for all process terms p and q: if $p \subseteq q$ and $q \subseteq p$, then $p = q$.

Exercise 15. Prove that the axioms A1–3 can be derived from A3 together with

$$\text{A2}' \quad (x + y) + z \;=\; y + (z + x).$$

Exercise 16. Suppose that a and b communicate to b', while a and c communicate to c'. Derive the equation $a \parallel (b + c) = (b + c) \parallel a$ from the axioms.

Exercise 17. Suppose action a cannot communicate with itself. Derive the equation $(b\cdot a) \parallel a = (b \parallel a)\cdot a$ from the axioms.

Exercise 18. Derive from the axioms, using induction, that the parallel operator is commutative and associative: $x \parallel y = y \parallel x$ and $(x \parallel y) \parallel z = x \parallel (y \parallel z)$.

Exercise 19. Let the communication of two actions from $\{a, b, c\}$ always result in c. Derive the equation $\partial_{\{a,b\}}((a\cdot b) \parallel (b\cdot a)) = c\cdot c$ from the axioms. (Cf. Example 4).

Exercise 20. Use the axioms to equate the process term

$$\partial_{\{send,\, read\}}((send(0) + send(1)) \parallel (read(0) + read(1)))$$

from Example 5 to $comm(0) + comm(1)$.

Exercise 21. Give an example to show that process terms $\partial_H(p \parallel q)$ and $\partial_H(p) \parallel \partial_H(q)$ can display distinct behaviour.

Exercise 22. Suppose $p + q = \delta$ can be derived from the axioms for certain process terms p and q. Derive $p = \delta$ from the axioms.

Exercise 23. Let the communication of b and c result in a, while a and c do not communicate. Say for each of the following process terms whether it contains a deadlock:

1. $\partial_{\{b\}}(a\cdot b + c)$

2. $\partial_{\{b\}}(a{\cdot}(b+c))$
3. $\partial_{\{b,c\}}(a{\cdot}(b+c))$
4. $\partial_{\{b\}}((a{\cdot}b) \parallel c)$
5. $\partial_{\{b,c\}}((a{\cdot}b) \parallel c)$

Exercise 24. Consider the *Clock* process from Example 6. Use the axioms to derive the equation

$$\partial_{\{tick\}}(Clock(0)) = display(0){\cdot}\partial_{\{tick\}}(Clock(0))$$

Exercise 25. Specify a process that adds the elements of a list of natural numbers and prints the final result.

Exercise 26. Derive the following three equations from the axioms, using induction.

1. $x \lhd b \rhd y = x \lhd b \rhd \delta + y \lhd \neg b \rhd \delta$
2. $x \lhd b_1 \vee b_2 \rhd \delta = x \lhd b_1 \rhd \delta + x \lhd b_2 \rhd \delta$
3. if $(b = \mathtt{T} \Rightarrow x = y)$, then $x \lhd b \rhd z = y \lhd b \rhd z$

Exercise 27. Specify a stack and a queue process. A stack (resp. queue) process is similar to the buffer in Example 8, but can read an unbounded number of elements of some sort D via action name r and deliver them in the reverse (resp. same) order via action name s.

Exercise 28. Derive from the axioms, using induction,

$$\sum_{b:Bool} x \lhd b \rhd y = x + y.$$

Exercise 29. Let $b : D \to Bool$ with $b(e) = \mathtt{T}$ for some $e \in D$. Derive from the axioms, using induction,

$$x = \sum_{d:D} x \lhd b(d) \rhd \delta.$$

Exercise 30. Give a process declaration of the bag over $\{d_1, d_2\}$ that does not include the three parallel operators.

Exercise 31. Say for each of the following pairs of process terms whether they are bisimilar:

1. $(b+c){\cdot}a + b{\cdot}a + c{\cdot}a$ and $b{\cdot}a + c{\cdot}a$
2. $a{\cdot}(b+c) + a{\cdot}b + a{\cdot}c$ and $a{\cdot}b + a{\cdot}c$
3. $(a+a){\cdot}(b{\cdot}c) + (a{\cdot}b)(c+c)$ and $(a{\cdot}(b+b))(c+c)$

For each pair of bisimilar terms, give a bisimulation relation that relates them.

Exercise 32. Show that the process terms $read(d){\cdot}(write_1(d) + write_2(d))$ and $read(d){\cdot}write_1(d) + read(d){\cdot}write_2(d)$ are not bisimilar.

Exercise 33. Let a^1 denote a, and let a^{k+1} denote $a{\cdot}a^k$ for $k > 0$. Prove that $a^k \not\leftrightarrows a^{k+1}$ for $k > 0$.

Exercise 34. Let $H \subseteq \mathsf{Act}$. Prove that $\partial_H(\rho_f(p)) \leftrightarrows \rho_f(\partial_{f^{-1}(H)}(p))$ for all process terms p (with $f^{-1}(H) = \{a \in \mathsf{Act} \mid f(a) \in H\}$).

4

Hiding Internal Transitions

In this chapter it is explained how one can abstract away from the internal transitions of a process, so that only its external, visible transitions remain.

4.1 Hiding of Actions

If a customer asks a programmer to implement a system, ideally this customer is able to provide the external (often called functional) behaviour of the desired program. That is, he or she should be able to tell what is the output of the program for each possible input. The programmer then comes up with an implementation. The question is, does this implementation really display the desired external behaviour? To answer this question, we need to abstract away from the internal transitions of the program.

Hiding is an important means in the analysis of distributed systems. Action names of internal events of a system can be hidden, so that the relationship between the external events becomes more clear. The hidden action is denoted by τ. This reserved action name is not in Act, it does not carry any data parameters, and it cannot communicate with any action names. Intuitively, a τ-transition in a system cannot be observed directly. The τ is meant for analysis purposes, and hardly ever used in system specifications, as it is uncommon to specify that something unobservable must happen.

A typical equation characterising τ is $a{\cdot}\tau{\cdot}p = a{\cdot}p$. It says that it is by observation impossible to tell whether or not hidden actions happen immediately after an action a. Sometimes, the presence of hidden actions can be observed, due to the context in which they appear. For example, $a + \tau{\cdot}b \neq a + b$, as the left-hand side can execute the τ, after which it only offers a b-transition, whereas the right-hand side cannot reach such a state. We will say that such a τ is *non-inert*.

Axioms B1,2 in Table 4.1 are the characterising equations for the hidden action. They express that a τ-transition is *inert* if it does not lose any possible behaviours (this will be explained in detail in Section 4.4).

Table 4.1. Axioms for hidden actions and hiding

B1	$x{\cdot}\tau = x$	
B2	$x{\cdot}(\tau{\cdot}(y+z)+y) = x{\cdot}(y+z)$	
TID	$\tau_I(\delta) = \delta$	
TI1	$\tau_I(a(\mathbf{d})) = a(\mathbf{d})$	if $a \notin I$
TI2	$\tau_I(a(\mathbf{d})) = \tau$	if $a \in I$
TI3	$\tau_I(x+y) = \tau_I(x) + \tau_I(y)$	
TI4	$\tau_I(x{\cdot}y) = \tau_I(x){\cdot}\tau_I(y)$	
SUM10	$\tau_I(\sum_{d:D} P(d)) = \sum_{d:D} \tau_I(P(d))$	
R2	$\rho_f(\tau) = \tau$	

In order to make actions hidden, the unary *hiding* operators τ_I, for subsets I of Act, are introduced. The process term $\tau_I(p)$ behaves as its argument p, except that all actions with a name from I are renamed to τ. This is characterised by axioms TID and TI1–4.

A recursive equation such as $X = \tau{\cdot}X$ does not specify a unique process; each process term $\tau{\cdot}p$ constitutes a solution for X. The recursive equations in a process declaration are *guarded* if each expression $Y(d_1, \ldots, d_m)$ in the right-hand side of a recursive equation occurs in a context of the form $C_1[a(\mathbf{d}){\cdot}C_2[]]$ with $a \in \mathsf{Act}$ (i.e., $a \neq \tau$). Typically, process declarations such as $X = X$ and $X = \tau{\cdot}X$ are not guarded.

4.2 Summary

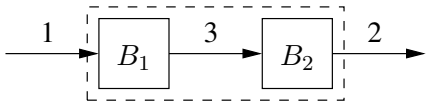
So far we have presented a standard framework for the specification and manipulation of concurrent processes. Summarising, it consists of basic operators (Act, $+$, $\cdot$) to define finite processes, parallel operators ($\|$, $\lfloor\!\lfloor$, $|$) to express parallelism and communication, deadlock and encapsulation to force actions into communication, the hidden action and hiding to make internal transitions invisible, process declarations to express recursion, the conditional operator and summation to mix data with processes, and action renaming to support the reuse of specifications.

In particular, the framework is suitable for the specification and verification of communication protocols. For such a verification, the desired external

behaviour of the protocol is represented in the form of a process declaration involving only the basic operators. Moreover, the implementation of the protocol is represented in the form of a process declaration that involves the basic operators and the three parallel operators. Next, the internal send and read transitions of the implementation are forced into communication using an encapsulation operator, and these internal communication transitions are made invisible using a hiding operator, so that only the input/output relation of the implementation remains. Finally, the protocol can be proved correct by equating the process term representing the implementation of the protocol to the process term representing the desired external behaviour, by means of the axioms.

4.3 An Example: Two One-Bit Buffers in Sequence

To give an example of the use of the framework described in the previous sections, we consider two buffers of capacity one that are put in sequence: buffer B_1 reads a datum from a channel 1 and sends this datum into channel 3, while buffer B_2 reads a datum from a channel 3 and sends this datum into channel 2. This system can be depicted as follows:



Let Δ denote a data domain. Action $r_i(d)$ represents reading datum d from channel i, while action $s_i(d)$ represents sending datum d into channel i. Moreover, action $c_3(d)$ denotes communication of datum d through channel 3. Similar to Example 5, $s_3 \mid r_3 = c_3$, while all other communications between actions result in δ. The buffers B_1 and B_2 are defined by the process declaration

$$\begin{array}{ll} \textbf{act} & r_1, r_2, r_3, s_1, s_2, s_3, c_3 : \ \Delta \\ \textbf{comm} & s_3 \,|\, r_3 = c_3 \\ \textbf{proc} & B_1 = \sum_{d:\Delta} r_1(d) \cdot s_3(d) \cdot B_1 \\ & B_2 = \sum_{d:\Delta} r_3(d) \cdot s_2(d) \cdot B_2 \end{array}$$

The system consists of buffers B_1 and B_2 in sequence, which is described by the initial declaration

$$\textbf{init} \quad \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_2 \parallel B_1))$$

The encapsulation operator enforces send and read actions over channel 3 into communication, while the hiding operator makes communication actions over channel 3 invisible.

The two buffers B_1 and B_2 of capacity one in sequence behave as a queue of capacity two, which can read two data elements from channel 1 before

sending them in the same order into channel 2. The queue of capacity two over Δ is described by the process declaration

$$\begin{array}{ll}\textbf{proc} & X = \sum_{d:\Delta} r_1(d)\cdot Y(d)\\ & Y(d{:}\Delta) = \sum_{e:\Delta} r_1(e)\cdot Z(d,e) + s_2(d)\cdot X\\ & Z(d{:}\Delta, e{:}\Delta) = s_2(d)\cdot Y(e)\end{array}$$

In state X, the queue of capacity two is empty, so that it can only read a datum d from channel 1 and proceed to the state $Y(d)$ where the queue contains d. In $Y(d)$, the queue can either read a second datum e from channel 1 and proceed to the state $Z(d,e)$ where the queue contains d and e, or send datum d into channel 2 and proceed to the state X where the queue is empty. Finally, in the state $Z(d,e)$ the queue is full, so that it can only send datum d into channel 2 and proceed to the state $Y(e)$ where it contains e.

We show algebraically that $\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_2 \parallel B_1))$ behaves as a queue of capacity two. In order to simplify the presentation, we assume that the data set Δ consists of the single element 0, and actions are abbreviated by omitting the suffix (0). First we expand $\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)$; in derivation steps, each subterm to which an axiom or process declaration is applied is underlined. Moreover, the axioms that are used in such a derivation step are mentioned above the corresponding equality sign.

Since the actions r_3 and r_1 do not communicate, the axioms for the parallel operators together with the recursive equations yield:

$$\begin{array}{rl}& \underline{B_2 \parallel B_1}\\ \overset{\text{CM1}}{=} & \underline{B_2 \mathbin{\lfloor\!\lfloor} B_1} + \underline{B_1 \mathbin{\lfloor\!\lfloor} B_2} + \underline{B_2 \mid B_1}\\ = & \underline{(r_3\cdot s_2\cdot B_2) \mathbin{\lfloor\!\lfloor} B_1} + \underline{(r_1\cdot s_3\cdot B_1) \mathbin{\lfloor\!\lfloor} B_2} + \underline{(r_3\cdot s_2\cdot B_2) \mid (r_1\cdot s_3\cdot B_1)}\\ \overset{\text{CM3,CM7}}{=} & r_3\cdot((s_2\cdot B_2) \parallel B_1) + r_1\cdot((s_3\cdot B_1) \parallel B_2) + \underline{\delta\cdot((s_2\cdot B_2) \parallel (s_3\cdot B_1))}\\ \overset{\text{A7}}{=} & r_3\cdot((s_2\cdot B_2) \parallel B_1) + \underline{r_1\cdot((s_3\cdot B_1) \parallel B_2) + \delta}\\ \overset{\text{A6}}{=} & r_3\cdot((s_2\cdot B_2) \parallel B_1) + r_1\cdot((s_3\cdot B_1) \parallel B_2).\end{array}$$

So the axioms for deadlock and encapsulation yield:

$$\begin{array}{rl}& \partial_{\{s_3,r_3\}}(\underline{B_2 \parallel B_1})\\ = & \underline{\partial_{\{s_3,r_3\}}(r_3\cdot((s_2\cdot B_2) \parallel B_1) + r_1\cdot((s_3\cdot B_1) \parallel B_2))}\\ \overset{\text{D3}}{=} & \underline{\partial_{\{s_3,r_3\}}(r_3\cdot((s_2\cdot B_2) \parallel B_1))} + \underline{\partial_{\{s_3,r_3\}}(r_1\cdot((s_3\cdot B_1) \parallel B_2))}\\ \overset{\text{D4}}{=} & \underline{\partial_{\{s_3,r_3\}}(r_3)}\cdot\partial_{\{s_3,r_3\}}((s_2\cdot B_2) \parallel B_1) + \underline{\partial_{\{s_3,r_3\}}(r_1)}\cdot\partial_{\{s_3,r_3\}}((s_3\cdot B_1) \parallel B_2)\\ \overset{\text{D1,2}}{=} & \underline{\delta\cdot\partial_{\{s_3,r_3\}}((s_2\cdot B_2) \parallel B_1)} + r_1\cdot\partial_{\{s_3,r_3\}}((s_3\cdot B_1) \parallel B_2)\\ \overset{\text{A7}}{=} & \underline{\delta + r_1\cdot\partial_{\{s_3,r_3\}}((s_3\cdot B_1) \parallel B_2)}\\ \overset{\text{A6}}{=} & r_1\cdot\partial_{\{s_3,r_3\}}((s_3\cdot B_1) \parallel B_2).\end{array}$$

Summarising, we have derived

$$\partial_{\{s_3,r_3\}}(B_2 \parallel B_1) = r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2). \tag{4.1}$$

We proceed to expand $\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2)$. As above, it can be derived from the axioms for the parallel operators together with the recursive equations that

$$(s_3 \cdot B_1) \parallel B_2 = s_3 \cdot (B_1 \parallel B_2) + r_3 \cdot ((s_2 \cdot B_2) \parallel (s_3 \cdot B_1)) + c_3 \cdot (B_1 \parallel (s_2 \cdot B_2)).$$

Using the equation above, it can be derived from the axioms for deadlock and encapsulation that

$$\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2) = c_3 \cdot \partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2)). \tag{4.2}$$

We proceed to expand $\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2))$. By the axioms for the parallel operators together with the recursive equations,

$$B_1 \parallel (s_2 \cdot B_2) = r_1 \cdot ((s_3 \cdot B_1) \parallel (s_2 \cdot B_2)) + s_2 \cdot (B_2 \parallel B_1).$$

So by the axioms for encapsulation,

$$\begin{aligned} &\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2)) \\ &= r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2)) + s_2 \cdot \partial_{\{s_3,r_3\}}(B_2 \parallel B_1). \end{aligned} \tag{4.3}$$

We proceed to expand $\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2))$. By the axioms for the parallel operators together with the recursive equations,

$$(s_3 \cdot B_1) \parallel (s_2 \cdot B_2) = s_3 \cdot (B_1 \parallel (s_2 \cdot B_2)) + s_2 \cdot (B_2 \parallel (s_3 \cdot B_1)).$$

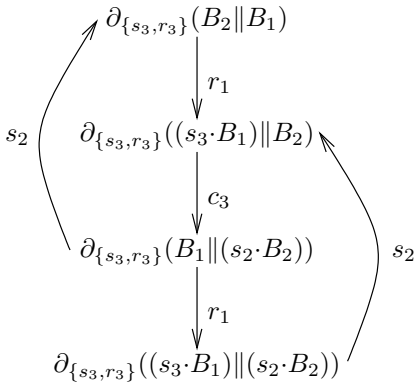
So by the axioms for deadlock and encapsulation,

$$\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2)) = s_2 \cdot \partial_{\{s_3,r_3\}}(B_2 \parallel (s_3 \cdot B_1)).$$

Commutativity of the merge (see Exercise 18) yields $B_2 \parallel (s_3 \cdot B_1) = (s_3 \cdot B_1) \parallel B_2$, so

$$\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2)) = s_2 \cdot \partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2). \tag{4.4}$$

Summarising, we have algebraically derived the following relations:



Equations (4.1) and (4.2) together with the axioms for τ and hiding yield:

$$\begin{aligned}
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)) &\overset{(4.1)}{=} \tau_{\{c_3\}}(r_1{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2)) \\
&\overset{\text{TI1,4}}{=} r_1{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2)) \\
&\overset{(4.2)}{=} r_1{\cdot}\tau_{\{c_3\}}(c_3{\cdot}\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
&\overset{\text{TI2,4}}{=} r_1{\cdot}\tau{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
&\overset{\text{B1}}{=} r_1{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))).
\end{aligned}$$

Moreover, equation (4.3) together with the axioms for hiding yield:

$$\begin{aligned}
&\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
\overset{(4.3)}{=}\ &\tau_{\{c_3\}}(r_1{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2)) + s_2{\cdot}\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)) \\
\overset{\text{TI1,3,4}}{=}\ &r_1{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2))) + s_2{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)).
\end{aligned}$$

Finally, equations (4.2) and (4.4) together with the axioms for τ and hiding yield:

$$\begin{aligned}
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2))) &\overset{(4.4)}{=} \tau_{\{c_3\}}(s_2{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2)) \\
&\overset{\text{TI1,4}}{=} s_2{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2)) \\
&\overset{(4.2)}{=} s_2{\cdot}\tau_{\{c_3\}}(c_3{\cdot}\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
&\overset{\text{TI2,4}}{=} s_2{\cdot}\tau{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
&\overset{\text{B1}}{=} s_2{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))).
\end{aligned}$$

The last three derivations together show that

$$\begin{aligned}
X &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)) \\
Y &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
Z &:= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2)))
\end{aligned}$$

is a solution for the process declaration of the queue of capacity two over $\{0\}$:

$$\begin{aligned}
X &= r_1{\cdot}Y \\
Y &= r_1{\cdot}Z + s_2{\cdot}X \\
Z &= s_2{\cdot}Y.
\end{aligned}$$

4.4 Branching Bisimilarity

While on the level of process terms with hidden actions we have defined an equality relation, we have not yet introduced a corresponding equivalence

relation on the states in a state space. (Cf. Section 3.10, where bisimilarity was defined on states, in the absence of τ). In this section we define the notion of *rooted branching bisimilarity* on states. This equivalence relation is sound with respect to the equality relation on process terms, in the sense that if two process terms can be equated then they are rooted branching bisimilar. Furthermore, in case the hidden action is absent, rooted branching bisimilarity agrees with bisimilarity.

Each equality relation on terms is by default closed under contexts; i.e., $p = q$ implies $C[p] = C[q]$ for all contexts $C[]$. Therefore, in order to capture an equivalence on process terms in an equational setting, it is imperative that the equivalence is a *congruence*, meaning that if process terms p and q are equivalent, then $C[p]$ and $C[q]$ are equivalent for all contexts $C[]$ (see also Section 3.10).

The intuition for the hidden action, that it represents an internal event of the system, in which we are not really interested, requires that two states may be equivalent even if one state can perform a τ while the other cannot. The question that we must pose ourselves is:

which τ-transitions are inert?

The obvious answer to this question, 'all τ-transitions are inert', turns out to be incorrect. Namely, this answer would produce an equivalence relation that, on the level of process terms, is not a congruence.

As an example of a τ-transition that is not inert, consider the process terms $a + \tau \cdot \delta$ and a. If the τ-transition in the first term were inert, then these two terms would be equivalent. However, the state space of the first term contains a deadlock, $a + \tau \cdot \delta \xrightarrow{\tau} \delta$, while the state space of the second term does not. Hence, the τ-transition in the first term is not inert. In order to describe this case more vividly, we give an example.

Example 10. Consider a protocol that first receives a datum d via channel 1, and then communicates this datum via channel 2 or via channel 3. If the datum is communicated through channel 2, then it is sent into channel 4. If the datum is communicated through channel 3, then it gets stuck, as the subsequent channel 5 is broken. So the system gets into a deadlock if the datum d is transferred via channel 3. This deadlock should not disappear if we abstract away from the internal communication transitions via channels 2 and 3, because this would cover up an important problem of the protocol.

The system, which is depicted in Fig. 4.1, is described by the process term

$$\begin{aligned} & \partial_{\{s_5\}}(r_1(d) \cdot (c_2(d) \cdot s_4(d) + c_3(d) \cdot s_5(d))) \\ \overset{\text{D1,2,4,5}}{=} \; & r_1(d) \cdot (c_2(d) \cdot s_4(d) + c_3(d) \cdot \delta) \end{aligned}$$

where $s_i(d)$, $r_i(d)$, and $c_i(d)$ represent a send, read, and communication action of the datum d via channel i, respectively. Hiding the communication actions $c_2(d)$ and $c_3(d)$ in this process term yields $r_1(d) \cdot (\tau \cdot s_4(d) + \tau \cdot \delta)$. The

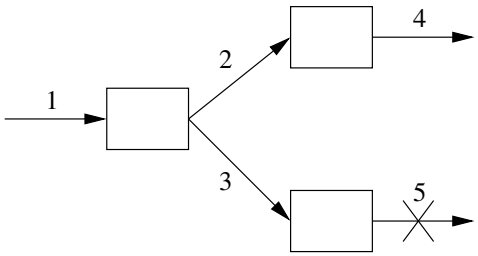


Fig. 4.1. Protocol with a malfunctioning channel

τ-transitions in this process term are not inert, because executing either of them constitutes a decision whether or not the process gets into a deadlock.

As a further example of a τ-transition that is not inert, consider the process terms $a+\tau\cdot b$ and $a+b$. We argued previously that the process terms $\partial_{\{b\}}(a+\tau\cdot b) = a+\tau\cdot\delta$ and $\partial_{\{b\}}(a+b) = a$ are not equivalent, because the first term contains a deadlock while the second term does not. Hence, $a+\tau\cdot b$ and $a+b$ cannot be equivalent, for else the envisioned equivalence relation would not be a congruence.

Problems with congruence can be avoided by taking a more restrictive view on abstracting away from hidden actions. A correct answer to the question

which τ-transitions are inert?

turns out to be

those τ-transitions that do not lose possible behaviours!

For example, the process terms $a+\tau\cdot(a+b)$ and $a+b$ are equivalent, because the τ-transition in the first process term is inert: after execution of this τ it is still possible to execute a. In general, process terms $p+\tau\cdot(p+q)$ and $p+q$ are equivalent for all process terms p and q. By contrast, in a process term such as $a+\tau\cdot b$ the τ-transition is not inert, since execution of this τ means losing the option to execute a.

The intuition above is formalised in the notion of *branching bisimilarity* [50]. Let the states s_1 and s_2 be branching bisimilar. If $s_1 \stackrel{\tau}{\to} s_1'$, then s_2 does not have to simulate this τ-transition if it is inert, meaning that s_1' and s_2 are branching bisimilar. Moreover, a non-inert transition $s_1 \stackrel{a(\mathbf{d})}{\to} s_1'$ need not be simulated by s_2 immediately, but only after a number of inert τ-transitions: $s_2 \stackrel{\tau}{\to} \cdots \stackrel{\tau}{\to} \hat{s} \stackrel{a(\mathbf{d})}{\to} s_2'$, where s_1 and $\hat{s}$ are branching bisimilar (to ensure that the τ-transitions are inert) and s_1' and s_2' are branching bisimilar (so that $s_1 \stackrel{a(\mathbf{d})}{\to} s_1'$ is simulated by $\hat{s} \stackrel{a(\mathbf{d})}{\to} s_2'$).

We recall that the predicate $\downarrow$ on states expresses successful termination; $\surd$ is the only state where $\downarrow$ holds (see Section 3.4).

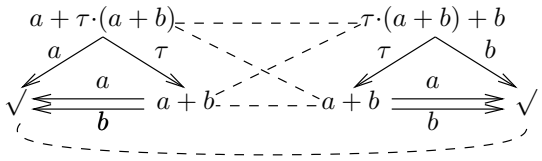
Definition 2 (Branching bisimilarity). *Assume a state space. A* branching bisimulation relation $\mathcal{B}$ *is a symmetric binary relation on states such that*

1. *if* $s_1 \mathcal{B} s_2$ *and* $s_1 \stackrel{a(\mathbf{d})}{\rightarrow} s_1'$, *then*
   - *either* $a = \tau$ *and* $s_1' \mathcal{B} s_2$;
   - *or there is a sequence of (zero or more)* τ*-transitions* $s_2 \stackrel{\tau}{\rightarrow} \cdots \stackrel{\tau}{\rightarrow} \hat{s}$ *such that* $s_1 \mathcal{B} \hat{s}$ *and* $\hat{s} \stackrel{a(\mathbf{d})}{\rightarrow} s_2'$ *with* $s_1' \mathcal{B} s_2'$; *and*
2. *if* $s_1 \mathcal{B} s_2$ *and* $s_1 \downarrow$, *then there is a sequence of (zero or more)* τ*-transitions* $s_2 \stackrel{\tau}{\rightarrow} \cdots \stackrel{\tau}{\rightarrow} \hat{s}$ *such that* $s_1 \mathcal{B} \hat{s}$ *and* $\hat{s} \downarrow$.

Two states s_1 *and* s_2 *are* branching bisimilar, *denoted by* $s_1 \underline{\leftrightarrow}_b s_2$, *if there is a branching bisimulation relation* $\mathcal{B}$ *such that* $s_1 \mathcal{B} s_2$.

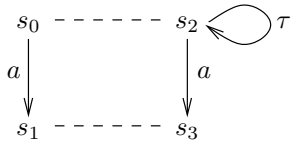
Example 11. $a + \tau{\cdot}(a+b) \underline{\leftrightarrow}_b \tau{\cdot}(a+b) + b$.

A branching bisimulation relation that relates these two process terms is defined by $a + \tau{\cdot}(a+b) \mathcal{B} \tau{\cdot}(a+b) + b$, $a + b \mathcal{B} \tau{\cdot}(a+b) + b$, $a + \tau{\cdot}(a+b) \mathcal{B} a + b$, $a + b \mathcal{B} a + b$, and $\surd \mathcal{B} \surd$. This relation can be depicted as follows:



It is left to the reader to verify that this relation satisfies the requirements of a branching bisimulation.

Branching bisimilarity satisfies a notion of *fairness*. That is, if an exit from a τ-loop exists, then no infinite execution sequence will remain in this τ-loop forever. The intuition is that there is zero chance that no exit from the τ-loop will ever be chosen. For example, it is not hard to see that the states s_0 and s_2 in the two state spaces below are branching bisimilar.



Groote and Vaandrager [64] presented an algorithm to decide which states in a finite state space are branching bisimilar. The worst-case time complexity of this algorithm is $O(mn)$, where n is the number of states and m the number of transitions in the state space; see Section 6.4.

Without proof we state that branching bisimilarity is an equivalence relation; see [10]. However, it is not yet a congruence. For example, $\tau{\cdot}a$ and a are branching bisimilar (see Exercise 38), but $\tau{\cdot}a+b$ and $a+b$ are not branching bisimilar. Namely, if $\tau{\cdot}a+b$ executes τ then it loses the option to execute b, so this τ-transition is not inert.

Milner [86] showed that this problem can be overcome by adding a rootedness condition: initial τ-transitions are never inert. In other words, two states are considered equivalent if they can simulate each other's initial transitions, such that the resulting states are branching bisimilar. This leads to the notion of *rooted branching bisimilarity*.

Definition 3 (Rooted branching bisimilarity). *Assume a state space. A* rooted branching bisimulation relation $\mathcal{B}$ *is a symmetric binary relation on states such that:*

1. *if* $s_1 \mathcal{B} s_2$ *and* $s_1 \stackrel{a(\mathbf{d})}{\rightarrow} s_1'$, *then* $s_2 \stackrel{a(\mathbf{d})}{\rightarrow} s_2'$ *with* $s_1' \underline{\leftrightarrow}_b s_2'$; *and*
2. *if* $s_1 \mathcal{B} s_2$ *and* $s_1 \downarrow$, *then* $s_2 \downarrow$.

Two states s_1 *and* s_2 *are* rooted branching bisimilar, *denoted by* $s_1 \underline{\leftrightarrow}_{rb} s_2$, *if there is a rooted branching bisimulation relation* $\mathcal{B}$ *such that* $s_1 \mathcal{B} s_2$.

Since branching bisimilarity is an equivalence relation, it is not hard to see that rooted branching bisimilarity is also an equivalence relation.

Branching bisimilarity strictly includes rooted branching bisimilarity, which in turn strictly includes bisimilarity:

$$\underline{\leftrightarrow} \subset \underline{\leftrightarrow}_{rb} \subset \underline{\leftrightarrow}_b \ .$$

In the absence of τ, bisimilarity and branching bisimilarity induce exactly the same equivalence classes.

Without proof we note that rooted branching bisimilarity is a *congruence* with respect to the operators in μCRL, meaning that if $p \underline{\leftrightarrow}_{rb} q$, then $C[p] \underline{\leftrightarrow}_{rb} C[q]$ for all contexts $C[]$. Moreover, rooted branching bisimilarity is *sound* with respect to the equality relation on process terms, in the sense that if two process terms can be equated, then they are rooted branching bisimilar.

Exercises

Exercise 35. Derive the following equations from the axioms.

1. $a{\cdot}(\tau{\cdot}b+b) = a{\cdot}b$;
2. $a{\cdot}(\tau{\cdot}(b+c)+b) = a{\cdot}(\tau{\cdot}(b+c)+c)$;
3. $\tau_{\{a\}}(a{\cdot}(a{\cdot}(b+c)+b)) = \tau_{\{d\}}(d{\cdot}(d{\cdot}(b+c)+c))$;
4. If $y \subseteq x$, then $\tau{\cdot}(\tau{\cdot}x+y) = \tau{\cdot}x$.

Exercise 36. Give a μCRL specification of the two buffers of capacity one in sequence together with their data types. Use the μCRL toolset to analyse this specification.

Exercise 37. Use the renaming operator (see Section 3.9) to extract a process declaration of the buffer B_2 in Exercise 36 from the process declaration of B_1.

Exercise 38. Give branching bisimulation relations to prove that the process terms a, $a{\cdot}\tau$, and $\tau{\cdot}a$ are branching bisimilar.

Exercise 39. Give a branching bisimulation relation to prove that the process terms $\tau{\cdot}(\tau{\cdot}(a+b)+b)+a$ and $a+b$ are branching bisimilar.

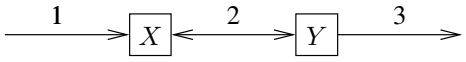
Exercise 40. Assume a state space, and let the states s and s' in this state space be on a τ-loop; that is, there exist sequences of τ-transitions $s \xrightarrow{\tau} \cdots \xrightarrow{\tau} s'$ and $s' \xrightarrow{\tau} \cdots \xrightarrow{\tau} s$. Prove that s and s' are branching bisimilar.

Exercise 41. Say for the following five pairs of process terms whether or not they are bisimilar, rooted branching bisimilar, or branching bisimilar:

1. $(a+b){\cdot}(c+d)$ and $a{\cdot}c+a{\cdot}d+b{\cdot}c+b{\cdot}d$
2. $(a+b){\cdot}(c+d)$ and $(b+a){\cdot}(d+c)+a{\cdot}(c+d)$
3. $\tau{\cdot}(b+a)+\tau{\cdot}(a+b)$ and $a+b$
4. $c{\cdot}(\tau{\cdot}(b+a)+\tau{\cdot}(a+b))$ and $c{\cdot}(a+b)$
5. $a{\cdot}(\tau{\cdot}b+c)$ and $a{\cdot}(b+\tau{\cdot}c)$

In each case, give explicit relations, or explain why such relations do not exist.

Exercise 42. Data elements (from a collection Δ) can be received by a one-place buffer X via channel 1, in which case they are sent on to one-place buffer Y via channel 2. Y either sends on an incoming datum via channel 2, or it sends back this datum to X via channel 2. In the latter case, X returns the datum to Y via channel 2.



X and Y are defined by the following μCRL specification:

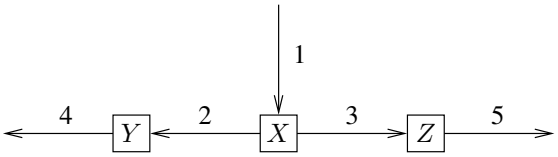
act $r_1, s_2, r_2, c_2, s_3 : \Delta$
comm $s_2 \,|\, r_2 = c_2$
proc $X = \sum_{d:\Delta}(r_1(d)+r_2(d)){\cdot}s_2(d){\cdot}X$
$Y = \sum_{d:\Delta} r_2(d){\cdot}(s_2(d)+s_3(d)){\cdot}Y$

Let p denote $\partial_{\{s_2,r_2\}}(X \| Y)$, and let Δ consist of $\{d_1, d_2\}$.

1. Draw the state space of p.
2. Are data elements read via channel 1 and sent via channel 3 in the same order?

3. Does $\partial_{\{s_3\}}(p)$ contain a deadlock? If yes, then give an execution trace to a deadlock state.
4. Draw the state space of $\tau_{\{c_2\}}(p)$ after minimisation modulo branching bisimilarity.

Exercise 43. Data elements (from a collection Δ) can be received by a one-bit buffer X via channel 1, in which case they are sent on in an alternating fashion to one-bit buffers Y and Z via channels 2 and 3, respectively. So the first received datum is sent to Y, the second to Z, the third to Y, etc. Y and Z send on incoming data elements via channels 4 and 5, respectively.



1. Specify the independent processes X, Y and Z in μCRL, including the action declaration **act**, the communication declaration **comm**, the process declaration **proc**, and the initial declaration **init**.
2. Let Δ consist of a single element. Draw the state space that belongs to the initial declaration.

Exercise 44. Five philosophers are sitting around a table, each with a plate of noodles. Between each pair of adjacent philosophers there is a single chopstick (so there are five chopsticks in total). A philosopher can only eat if he holds the chopsticks at his left and at his right.

Give a μCRL specification of the dining philosophers in which no philosopher is ever permanently excluded from the meal. Use actions $u(i)$ and $d(i)$ to represent that a philosopher takes up or puts down chopstick i, respectively. Draw the state space belonging to your μCRL specification.

5

Protocol Specifications

This chapter contains detailed descriptions of μCRL specifications of a number of communication protocols. In the exercises, it is asked to provide complete μCRL specifications of these protocols, so that the state spaces of these protocols can be generated. Next the μCRL and CADP toolsets can be used to minimise and analyse these state spaces, using techniques that will be presented in Chapter 6. In Section 7.4, the correctness of one of these protocols, namely the synchronous version of the Tree Identify Protocol, will be proved using a symbolic verification technique, which will be explained in Section 7.3.

5.1 Alternating Bit Protocol

Suppose two armies have agreed to attack a city at the same time. The two armies reside on different hills, while the city lies in between these two hills. The only way for the armies to communicate with each other is by sending messengers through the hostile city. This communication is inherently unsafe; if a messenger is caught inside the city, then the message does not reach its destination. The paradox is that in such a situation, the two armies are never able to be 100% sure that they have agreed on a time to attack the city. Namely, if one army sends the message that it will attack at say 11am, then the other army has to acknowledge reception of this message before this time; army one has to acknowledge the reception of this acknowledgement, etc.

The alternating bit protocol (ABP) [9, 81] ensures successful transmission of data through a corrupted channel (such as messengers through a hostile city). This success is based on the assumption that data can be resent an unlimited number of times, i.e., there is no time limit on when a datum must have been transmitted. Then the fairness assumption, which underlies branching bisimilarity (see Section 4.4), guarantees that eventually the datum will be transmitted successfully.

The ABP is depicted in Fig. 5.1. Data elements $d_1, d_2, d_3, \ldots$ from a non-empty set Δ are communicated between a Sender and a Receiver. If the

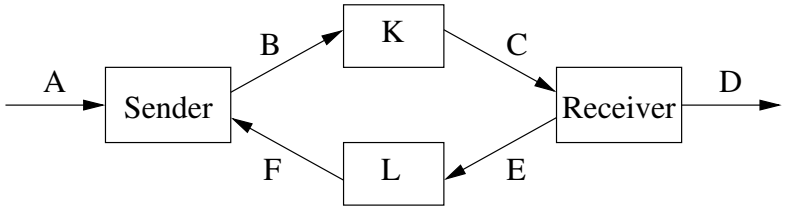


Fig. 5.1. Alternating bit protocol

Sender reads a datum from channel A, then this datum is communicated asynchronously to the Receiver, which sends the datum into channel D. However, the channels between the Sender and the Receiver are corrupted, so that a message that is communicated through these channels can be turned into an error message $\bot$. Therefore, every time the Receiver receives a message, it sends an acknowledgement to the Sender, which can also be corrupted.

Note that messages cannot get lost; they can only get corrupted. The Bounded Retransmission Protocol, which will be discussed in the next section, is an adaptation of the ABP in which messages can get lost completely. There a time-out mechanism will be used to ensure progress of that protocol. For the ABP, however, the corrupted messages $\bot$ guarantee progress without such a time-out mechanism.

In the ABP, the Sender attaches a bit 0 to data elements d_{2k-1} and a bit 1 to data elements d_{2k}. As soon as the Receiver reads a datum, it sends back the attached bit, to acknowledge reception. If the Receiver receives a corrupted message, then it sends the previous acknowledgement to the Sender once more. The Sender keeps on sending a pair (d_i, b) as long as it receives the acknowledgement $1 - b$ or $\bot$. When the Sender receives the acknowledgement b, it starts sending out the next datum d_{i+1} with attached bit $1 - b$, until it receives the acknowledgement $1 - b$, etc. Alternation of the attached bit enables the Receiver to determine whether a received datum is really new, and alternation of the acknowledgement enables the Sender to determine whether an acknowledgement really concerns the reception of a datum, and not of an error message.

We first give a μCRL specification of the desired external behaviour of the ABP: The data elements that are read from channel A by the Sender are sent into channel D by the Receiver in the same order, and no data elements are lost. In other words, the desired external behaviour of the ABP is specified by the process declaration

$$X \;=\; \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot s_{\mathrm{D}}(d)\cdot X$$

where action $r_{\mathrm{A}}(d)$ represents 'read datum d from channel A', and action $s_{\mathrm{D}}(d)$ represents 'send datum d into channel D'.

Next, we give a μCRL specification of the ABP itself. First, we specify the Sender in the state that it sends out a datum with the bit b attached to it, represented by the process term $S(b)$ for $b \in Bit$, where Bit consists of $\{0, 1\}$:

$$\begin{aligned} S(b{:}Bit) &= \sum_{d:\Delta} r_{\mathrm{A}}(d) \cdot s_{\mathrm{B}}(d,b) \cdot T(d,b) \\ T(d{:}\Delta, b{:}Bit) &= r_{\mathrm{F}}(b) \cdot S(1-b) \\ &\quad + (r_{\mathrm{F}}(1-b) + r_{\mathrm{F}}(\bot)) \cdot s_{\mathrm{B}}(d,b) \cdot T(d,b) \end{aligned}$$

In state $S(b)$, the Sender reads a datum d from channel A, and sends this datum into channel B, with the bit b attached to it. Next, the system proceeds to state $T(d, b)$, in which it expects to receive the acknowledgement b through channel F, ensuring that the pair (d, b) has reached the Receiver unscathed. If the correct acknowledgement b is received, then the system proceeds to state $S(1 - b)$, in which it is going to send out a datum with the bit $1 - b$ attached to it. If the acknowledgement is either the wrong bit $1-b$ or the error message $\bot$, then the system sends the pair (d, b) into channel B once more.

Next, we specify the Receiver in the state that it is expecting to receive a datum with the bit b attached to it, represented by the process term $R(b)$:

$$\begin{aligned} R(b{:}Bit) &= \sum_{d:\Delta} r_{\mathrm{C}}(d,b) \cdot s_{\mathrm{D}}(d) \cdot s_{\mathrm{E}}(b) \cdot R(1-b) \\ &\quad + \sum_{d:\Delta} (r_{\mathrm{C}}(d,1{-}b) + r_{\mathrm{C}}(\bot)) \cdot s_{\mathrm{E}}(1-b) \cdot R(b) \end{aligned}$$

In state $R(b)$ there are two possibilities.

1. If in $R(b)$ the Receiver reads a pair (d, b) from channel C, then this constitutes new information, so the datum d is sent into channel D, after which acknowledgement b is sent to the Sender via channel E. Next, the Receiver proceeds to state $R(1 - b)$, in which it is expecting to receive a datum with the bit $1 - b$ attached to it.
2. If in $R(b)$ the Receiver reads a pair $(d, 1{-}b)$ or an error message $\bot$ from channel C, then this does not constitute new information. So then the Receiver sends acknowledgement $1 - b$ to the Sender via channel E, and remains in state $R(b)$.

To model asynchronous communication, the channels between the Sender and the Receiver are specified as separate concurrent components. The recursive equations of these components K and L express that messages between the Sender and the Receiver, and vice versa, may become corrupted.

$$\begin{aligned} K &= \sum_{d:\Delta} \sum_{b:Bit} r_{\mathrm{B}}(d,b) \cdot (j \cdot s_{\mathrm{C}}(d,b) + j \cdot s_{\mathrm{C}}(\bot)) \cdot K \\ L &= \sum_{b:Bit} r_{\mathrm{E}}(b) \cdot (j \cdot s_{\mathrm{F}}(b) + j \cdot s_{\mathrm{F}}(\bot)) \cdot L \end{aligned}$$

The action j expresses a non-deterministic choice whether or not a message is corrupted. This action is included because K determines whether (d, b) or $\bot$ is communicated, and not the Receiver; likewise, L determines whether b or $\bot$ is communicated, and not the Sender. Without the action j, deadlocks could disappear in the abstract μCRL specification. For instance, if the j's were omitted from K and L, and T_{db} and R_b would only consist of their first summand, then the protocol specification would work correctly, because the actions $s_{\mathrm{C}}(\bot)$ and $s_{\mathrm{F}}(\bot)$ offered by K and L would simply be ignored by the Receiver and the Sender, respectively.

A send and a read action of the same message ((d, b), b, or $\bot$) over the same internal channel (B, C, E or F) communicate with each other:

$$\begin{array}{ll} s_{\mathrm{B}} \,|\, r_{\mathrm{B}} = c_{\mathrm{B}} & s_{\mathrm{E}} \,|\, r_{\mathrm{E}} = c_{\mathrm{E}} \\ s_{\mathrm{C}} \,|\, r_{\mathrm{C}} = c_{\mathrm{C}} & s_{\mathrm{F}} \,|\, r_{\mathrm{F}} = c_{\mathrm{F}} \end{array}$$

All other communications between action names result in δ.

The initial state of the ABP is obtained by putting $S(0)$, $R(0)$, K and L in parallel, encapsulating send and read actions over internal channels, and hiding communication actions over these channels and the action j. That is, the ABP is expressed by the process term

$$\partial_H(S(0) \parallel R(0) \parallel K \parallel L)$$

where the set H consists of all send and read actions over B, C, E and F. The state space of $\partial_H(S(0) \parallel R(0) \parallel K \parallel L)$ is depicted in Fig. 5.2, where d and e range over Δ.

Note that the state space is symmetric; in one half any $d \in \Delta$ is being communicated, while in the other half any $e \in \Delta$ is being communicated. The only important distinction between the two halves is that at the left-hand side a bit 0 is attached to messages, while in the right-hand side a bit 1 is being attached.

To abstract away from internal transitions in the ABP, one can study the process term

$$\tau_I(\partial_H(S(0) \parallel R(0) \parallel K \parallel L))$$

where the set I consists of all communication actions over B, C, E and F together with j.

In the state space belonging to this process term, all τ-transitions are inert, i.e., do not lose possible behaviours (cf. Section 4.4). Moreover, corresponding states in the two symmetric halves of the state space (e.g., $\tau_I(\partial_H(S(0) \parallel R(0) \parallel K \parallel L))$ in the left half and $\tau_I(\partial_H(S(1) \parallel R(1) \parallel K \parallel L))$ in the right half) are branching bisimilar.

In Exercise 45 it is asked to specify the ABP in the μCRL toolset, including all data types. Then one can instantiate Δ with a finite data set, and generate the state space. After minimisation modulo branching bisimilarity (see Section 6.4), the resulting state space should correspond with the desired external behaviour of the ABP, which was described at the start of this section.

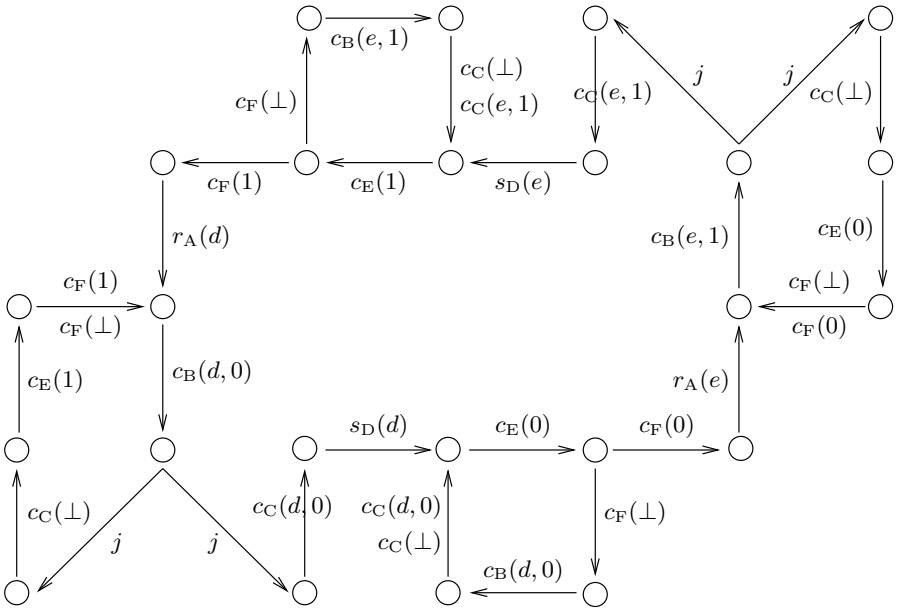


Fig. 5.2. State space of $\partial_H(S(0) \parallel R(0) \parallel K \parallel L)$

Alternatively, one could equate the process term above to a process term representing the desired external behaviour. Such a symbolic correctness proof of the ABP can be found in, e.g., [46]. It resembles the correctness proof of two one-bit buffers in sequence that was presented in Section 4.3; but of course the correctness proof for the ABP is considerably more complicated than the one for the two buffers in sequence.

5.2 Bounded Retransmission Protocol

Philips formulated a bounded retransmission protocol (BRP) for the implementation of a remote control (RC). Data elements that are sent from the RC to their destination, say a TV, may get lost. For example, the user may point the RC in the wrong direction. Therefore, if the TV receives a datum, it sends back a message to the RC, to acknowledge reception; this acknowledgement may also get lost. As in the ABP, the RC attaches an alternating bit to each datum that it sends to the TV, so that the TV can recognise whether it received a datum before.

In general, the data packets that are sent from the RC to the TV are large, so that they cannot be sent in one go. This means that each data packet is chopped into little pieces, and the RC sends these pieces one by one. The RC attaches a special label to the last element of a data packet, so that

at reception of this datum, the TV recognises that this completes the data packet.

A datum can only be resent a limited number of times. This means that the correctness criterion cannot be that each datum that is sent by the RC will eventually reach the TV. Instead, it is required that either the complete data packet is communicated between the RC and the TV, or the RC sends an appropriate message to the outside world to inform its corresponding partner that this communication has (or may have) failed.

In the ABP, the unrealistic assumption was made that messages can get corrupted but never get lost completely. In the BRP, however, in the asynchronous communication between the RC and the TV, data elements may get lost. In order to ensure that the BRP progresses, we need to incorporate some notion of time. Namely, if the RC sends a datum to the TV and does not receive an acknowledgement within a certain period of time, then it is certain that the datum or its acknowledgement was lost, so that the datum has to be resent. Furthermore, if the TV does not receive a next datum within a certain period of time, then it can be sure that the RC has given up transmission of a data packet.

Two timer processes T_1 and T_2 send time-out messages to the RC and the TV, respectively. If the RC sends a datum to the TV, then it sets timer T_1; if the RC receives an acknowledgement, then it resets T_1. Alternatively, T_1 sends a time-out to the RC, to signal that the acknowledgement has been delayed for too long; in that case, the RC resends the datum. Likewise, timer T_2 can send a time-out to the TV, to signal that the next datum has been delayed so long that the RC must have given up transmission of the data packet.

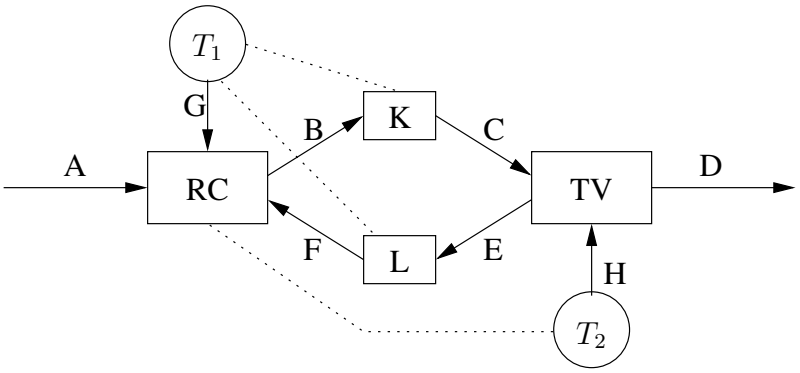


Fig. 5.3. Bounded retransmission protocol

The BRP is depicted in Fig. 5.3. The medium between the RC and the TV is represented by two separate components K and L, which can pass on a datum or lose it at random. The dotted lines between these components and timer T_1 designate that losing a datum or an acknowledgement triggers T_1 to

send a time-out to the RC via channel G. Similarly, the dotted line between the RC and the timer T_2 designates that if the RC gives up transmitting a data packet, then this is followed by a delay that is sufficiently long for T_2 to send a time-out to the TV via channel H.

First, we give an informal description of the BRP, and explain its required external behaviour. Next, we present the formal specification, which is a simplification of the specification in [55], where setting and resetting the timers is performed by explicit actions, error messages are more sophisticated, and special actions are needed in order to enforce synchronisation of the RC and the TV.

Suppose the RC receives a data packet $(d_1, \ldots, d_N)$ via channel A. Then the RC transmits the data elements $d_1, \ldots, d_N$ separately, where the last datum d_N is supplied with a special label *last*. Furthermore, each datum is supplied with an alternating bit 0 or 1: data elements d_{2k-1} are supplied with bit 0 while data elements d_{2k} are supplied with bit 1. If the RC sends a pair (d_i, b) into channel B for the first time, then it sets timer T_1, and moreover it sets a counter at one to keep track of the number of attempts to send datum d_i. Now there are two possibilities:

1. The RC receives an acknowledgement *ack* via channel F. Then it sends out the next pair $(d_{i+1}, 1-b)$, sets timer T_1 again, and gives the counter the value zero.
2. The RC receives a time-out from timer T_1 via channel G. Then it sends out the pair (d_i, b) again, sets timer T_1 again, and increases the value of the counter by one.

Transmission of the data packet is either completed successfully, if the RC receives an acknowledgement from the TV that it received the last datum d_N of the packet, or broken off unsuccessfully, if at some point the counter reaches its preset maximum value *max*. In the first case, the RC sends the message I_{OK} into channel A, to inform the outside world that transmission of the data packet $(d_1, \ldots, d_N)$ was concluded successfully. In the second case, the RC sends the message I_{NOK} into channel A, to inform the outside world that transmission of the data packet failed.

If the TV receives a pair (d_i, b) via channel C for the first time (which can be judged from the attached bit), then it sends d_i into channel D if $i > 1$, or the pair (d_i, first) if $i = 1$, to inform its corresponding partner in the outside world that this is the first datum of a new data package. Next, it sends an acknowledgement *ack* into channel E. Now there are three possibilities:

1. The TV receives the next pair $(d_{i+1}, 1-b)$ via channel C. Then it sends d_{i+1} into channel D and *ack* into channel E.
2. The TV receives the pair (d_i, b) again. Then it only sends *ack* into channel E.
3. The TV receives a time-out from timer T_2 via channel H, which signals that the RC has given up transmission of the data packet.

This procedure is repeated until the TV may receive a message (d, b, last), in which case it sends the pair (d, last) into channel D, informing its corresponding partner in the outside world that this successfully concludes transmission of the data packet. (We assume that all data packets have length at least two, so that there is no collision between the labels *first* and *last*.)

K and L represent the non-deterministic behaviour of the medium between the RC and the TV. If K reads a message via channel B, then it may or may not pass on this message to the TV via channel C. In the latter case, timer T_1 will eventually send a time-out to the RC. Similarly, if L reads a message via channel E, then it may or may not pass on this message to the RC via channel F. In the latter case, timer T_1 will eventually send a time-out to the RC.

This almost finishes the informal description of the BRP. However, there is one aspect of this protocol that has not yet been discussed, concerning error messages. This characteristic is explained using the specification of the required external behaviour, which is depicted in Fig. 5.4. The clockwise circle in this picture represents successful transfers of data elements (starting at the leftmost node), while the transitions that digress from this circle are error messages that are sent into channel A.

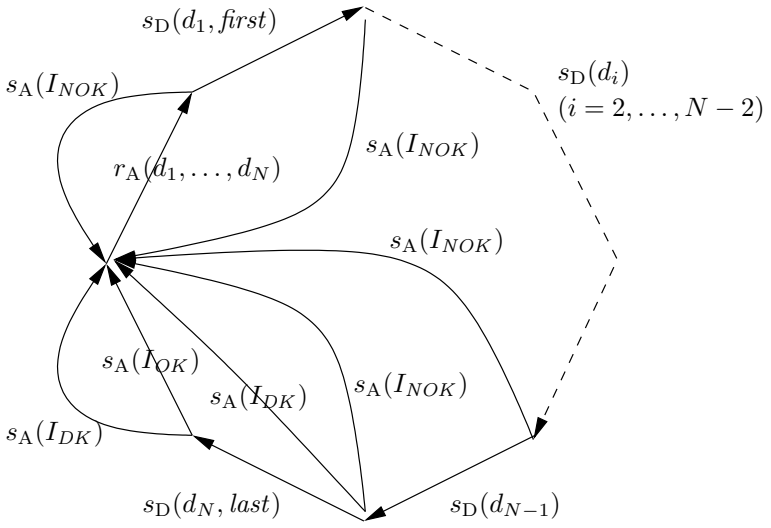


Fig. 5.4. External behaviour of the BRP

There is one special case with respect to the messages that are sent into channel A, at the end of transmission of a data packet. Suppose the RC attempted to send the final triple (d_N, b, last) to the TV, but that it did not receive an acknowledgement, even after the maximum number of tries. Then the RC does not know whether the TV received the datum d_N, so it cannot

be certain that transmission of the data packet was concluded successfully. In this case the RC sends a special error message I_{DK} into channel A.

We proceed to present a process declaration that formally specifies the BRP. This process declaration exhibits the external behaviour depicted in Fig. 5.4, intertwined with non-inert τ-transitions. The process declaration uses the following data parameters and functions.

- d ranges over a non-empty data set Δ, and λ ranges over the set Λ of non-empty lists of data elements. $head(\lambda)$ represents the first element of the list λ, and $tail(\lambda)$ the remaining list; moreover, $length(\lambda)$ represents the length of λ (see Exercise 4).
- b ranges over Bit, while n ranges over $\{1, \ldots, max\}$, where $max \geq 1$ is the maximum number of attempts that the RC is allowed to undertake to transmit a datum to the TV. In the μCRL specification of the BRP, one can declare $max :\rightarrow Nat$, and this function symbol of arity zero can be instantiated with the desired value in the algebraic specification of the natural numbers.
- Finally, we have the acknowledgement ack, the time-out to, the appendices $first$ and $last$ for the first and last datum of a data packet, and the messages I_{OK}, I_{NOK}, and I_{DK} for the outside world.

We start with the specification of the RC; its initial state is represented by the recursion variable X.

$$X \;=\; \textstyle\sum_{\lambda:\Lambda} r_{\mathrm{A}}(\lambda)\cdot Y(\lambda, 0, S(0)) \lhd \mathit{length}(\lambda) > S(0) \rhd \delta$$

$$Y(\lambda{:}\Lambda, b{:}\mathit{Bit}, n{:}\mathit{Nat})$$

$$= \; (s_{\mathrm{B}}(\mathit{head}(\lambda), b) \lhd \mathit{length}(\lambda) > S(0) \rhd s_{\mathrm{B}}(\mathit{head}(\lambda), b, \mathit{last}))\cdot Z(\lambda, b, n)$$

$$Z(\lambda{:}\Lambda, b{:}\mathit{Bit}, n{:}\mathit{Nat})$$

$$= \; r_{\mathrm{F}}(\mathit{ack})\cdot(Y(\mathit{tail}(\lambda), 1{-}b, S(0)) \lhd \mathit{length}(\lambda) > S(0) \rhd s_{\mathrm{A}}(I_{OK})\cdot X)$$

$$+ \; r_{\mathrm{G}}(\mathit{to})\cdot(Y(\lambda, b, S(n)) \lhd n < \mathit{max} \rhd$$

$$(s_{\mathrm{A}}(I_{NOK}) \lhd \mathit{length}(\lambda) > S(0) \rhd s_{\mathrm{A}}(I_{DK}))\cdot s_{\mathrm{H}}(\mathit{to})\cdot X)$$

The intuition behind this process declaration is as follows.

- In state X, the RC waits until it receives a data packet λ via channel A, after which it proceeds to $Y(\lambda, 0, S(0))$. The second argument 0 represents the bit that is going to be attached to $head(\lambda)$, while the third argument $S(0)$ represents a counter that registers the number of attempts to send the head of λ to the TV.
- In state $Y(\lambda, b, n)$, the RC attempts for the nth time to send the head of list λ to the TV via channel B, with bit b attached to it. At that moment the RC resets the timer T_1; this is left implicit in the specification. If λ consists of a single datum, then moreover a label $last$ is attached to this message.

- In state $Z(\lambda, b, n)$, the RC waits for either an acknowledgement via channel F or a time-out via channel G.
  - Suppose the RC receives an acknowledgement from the TV. If λ contains two or more data elements, then it proceeds to send the head of $\mathit{tail}(\lambda)$ to the TV, with bit $1 - b$ attached to it, and the counter starting at zero. If λ consists of a single datum, then it concludes successful transmission of the data packet by sending I_{OK} into channel A, and proceeds to state X.
  - Suppose the RC receives a time-out from timer T_1. If $n < \mathit{max}$, then it sends the pair $(\mathit{head}(\lambda), b)$ to the TV again, with the counter n increased by one. If $n = \mathit{max}$, then it concludes that transmission of the data packet was unsuccessful (if λ consists of two or more elements) or may have been unsuccessful (if λ consists of a single element), by sending I_{NOK} or I_{DK} into channel A, respectively. This message is followed by a delay, sufficiently long to let timer T_2 send a time-out to the TV via channel H, after which the RC proceeds to state X.

Next, we specify the TV; its initial state is represented by the recursion variable V:

$$\begin{aligned} V &= \textstyle\sum_{d:\Delta} r_{\mathrm{C}}(d,0)\cdot s_{\mathrm{D}}(d,\mathit{first})\cdot s_{\mathrm{E}}(\mathit{ack})\cdot W(1) \\ &+ \textstyle\sum_{d:\Delta}(r_{\mathrm{C}}(d,0,\mathit{last}) + r_{\mathrm{C}}(d,1,\mathit{last}))\cdot s_{\mathrm{E}}(\mathit{ack})\cdot V \\ &+ r_{\mathrm{H}}(\mathit{to})\cdot V \end{aligned}$$

$$\begin{aligned} W(b{:}\mathit{Bit}) &= \textstyle\sum_{d:\Delta} r_{\mathrm{C}}(d,b)\cdot s_{\mathrm{D}}(d)\cdot s_{\mathrm{E}}(\mathit{ack})\cdot W(1-b) \\ &+ \textstyle\sum_{d:\Delta} r_{\mathrm{C}}(d,b,\mathit{last})\cdot s_{\mathrm{D}}(d,\mathit{last})\cdot s_{\mathrm{E}}(\mathit{ack})\cdot V \\ &+ \textstyle\sum_{d:\Delta} r_{\mathrm{C}}(d,1{-}b)\cdot s_{\mathrm{E}}(\mathit{ack})\cdot W(b) \\ &+ r_{\mathrm{H}}(\mathit{to})\cdot V \end{aligned}$$

The intuition behind these recursive equations is as follows.

- In state V, the TV is waiting for the first element of a new data packet, with the bit 0 attached to it. If it receives a message $(d, 0)$, then it sends the pair (d, first) into channel D, sends an acknowledgement into channel E, and proceeds to state $W(1)$.
  If the TV receives a message with *last* attached to it, then it recognises that it already received this datum before: it is the last datum of the data packet that it received previously. Hence, the TV only sends an acknowledgement into channel E, and remains in state V.
  Every time the TV receives a receives a datum, it resets the timer T_2; this is left implicit in the specification.
  Finally, the TV may receive a time-out from timer T_2 via channel H, which signals that the RC never received an acknowledgement for the last datum of the previous data packet, or that the RC failed to transfer a single datum of some new data packet. Then the TV remains in state V.

- In state $W(b)$, the TV has received some but not all data of a packet from the RC, and is waiting for a datum with the bit b attached to it. If it receives such a message, then it sends the datum into channel D, sends an acknowledgement into channel E, and proceeds to state $W(1-b)$ to wait for a message with the bit $1-b$ attached to it. If the TV receives a message with not only b but also *last* attached to it, then it concludes that the data packet has been transferred successfully. In that case it sends the pair $(d, last)$ into channel D, sends an acknowledgement into channel E, and proceeds to state V.
  If the TV receives a message with the bit $1-b$ attached to it, then it already received this datum before. Hence, it only sends an acknowledgement into channel E, and remains in state $W(b)$.
  Finally, the TV may receive a time-out from timer T_2 via channel H, which signals that the RC has given up transmission of the data packet. Then the TV proceeds to state V.

  Finally, we specify the mediums K and L:

$$K = \sum_{d:\Delta} \sum_{b:Bit} \{r_{\mathrm{B}}(d,b)\cdot(j\cdot s_{\mathrm{C}}(d,b) + j\cdot s_{\mathrm{G}}(to))\cdot K \\ + r_{\mathrm{B}}(d,b,last)\cdot(j\cdot s_{\mathrm{C}}(d,b,last) + j\cdot s_{\mathrm{G}}(to))\cdot K\}$$

$$L = r_{\mathrm{E}}(ack)\cdot(j\cdot s_{\mathrm{F}}(ack) + j\cdot s_{\mathrm{G}}(to))\cdot L$$

As in the ABP, the action j expresses the non-deterministic choice whether or not a message is lost.

The intuition behind these recursive equations is as follows.

- If K receives a message from the RC via channel B, then either it passes on this message to the TV via channel C, or it loses the message. In the latter case, the subsequent delay triggers timer T_1 to send a time-out to the RC via channel G.
- If L receives an acknowledgement from the TV via channel E, then either it passes on this acknowledgement to the RC via channel F, or it loses the acknowledgement. In the latter case, the subsequent delay triggers timer T_1 to send a time-out to the RC via channel G.

  The initial state of the BRP is expressed by

$$\tau_I(\partial_H(V \parallel X \parallel K \parallel L))$$

where the set H consists of the read and send actions over the internal channels B, C, E, F, G, and H, while the set I consists of the communication actions over these internal channels together with j.

In Exercise 47 it is asked to specify the BRP in the μCRL toolset, including all data types. Then one can provide concrete, finite instantiations for Δ, Λ and max, and generate the state space. After minimisation modulo branching bisimilarity, the resulting state space should correspond with the desired external behaviour (see Fig. 5.4), intertwined with non-inert τ-transitions.

Alternatively, one could equate the process term above to a process term representing the desired external behaviour. Such a symbolic correctness proof of the BRP can be found in [46]. The process algebraic verification of the BRP in [55] was checked using the theorem prover Coq. Alternative specifications and verifications of the BRP can be found in [1, 39, 55, 67].

5.3 Sliding Window Protocol

In the ABP and the BRP, the Sender sends out a datum and then waits for an acknowledgement or a time-out before it sends the next datum. In situations where transmission of data is relatively time consuming, this procedure tends to be unacceptably slow. In sliding window protocols [32] (see also [102]), a Sender can send out data elements without waiting for acknowledgements.

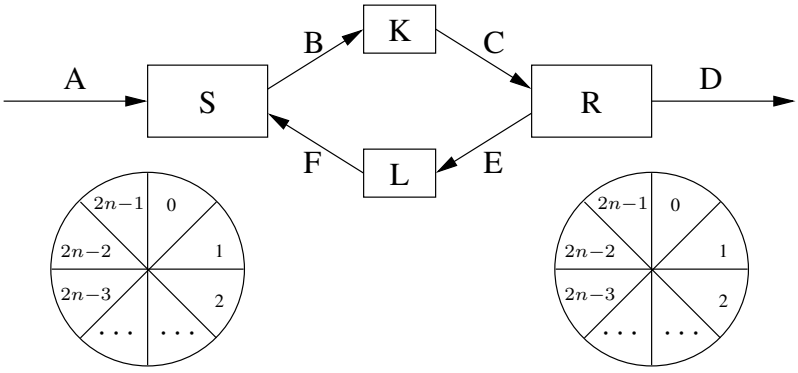


Fig. 5.5. Sliding window protocol

As in the BRP, we assume that the elements from a non-empty data domain Δ that are communicated asynchronously between a Sender to a Receiver may get lost. We specify a sliding window protocol (SWP) in which the Sender and the Receiver store incoming data elements in buffers of the same size $2n > 0$; see Fig. 5.5. At any time, each buffer is divided into one half that may contain data elements, and one half that must be empty. The part of the buffer that may contain data elements is called its *window*.

In the SWP, the Sender reads data elements from channel A and stores them in its buffer. Each incoming datum is stored at the next free position in the window; in other words, if the previous incoming datum was stored at position k, then the current one is stored at position $k + 1$, modulo $2n$. In case the window is full, the Sender cannot read data elements from channel A. At any time, the Sender can send a datum from its window into channel B, paired with its position number in the buffer; data elements can be selected for transmission at random from (the filled part of) the sending window. The

Receiver stores the data elements that it reads via channel C in the window of its buffer; if it receives a pair (d, k), then datum d is stored at position k. If however k is outside the receiving window, then the Receiver must purge the pair (d, k). If the first position of its window contains a datum e, then the Receiver can send e into channel D, remove this datum from its buffer, and slide its window forward by one position. The Receiver can also at any time send as an acknowledgement a number k into channel E, to inform the Sender that it received all data elements up to (but not including) position k. Upon reception of such an acknowledgement via channel F, the Sender eliminates all pairs up to position k from its buffer, and slides forward its window accordingly.

In the ABP and the BRP we used an alternating bit to make it possible for the Receiver to distinguish old from new data elements. In the SWP, we use the restriction that the sending and receiving window may not extend beyond half the size of the sending and receiving buffer, respectively. If the Receiver reads a pair (d, k) from channel C where k is within its window, then the Receiver can be certain that it did not yet send datum d (at position k) into channel D. (Actually, in practice the sliding window is simulated using an alternating bit on top of position numbers, so that one does not have to waste half the buffer space.)

We proceed to specify the SWP in μCRL. As said before, we assume that the sending and receiving buffers have predefined size $2n$ for some $n > 0$, and that the sliding windows in these buffers are restricted to n positions. As can be seen in Fig. 5.5, the nature of position numbers in buffers is firmly linked with modulo arithmetic (see, e.g., [40]). Two natural numbers are considered equal modulo $2n$ if their difference is divisible by $2n$. It is not hard to see that this defines an equivalence relation on natural numbers, with equivalence classes $0, \ldots, 2n-1$.

The buffer is modelled as a list of pairs (d, k) with $d \in \Delta$ and $k \in Nat$, representing that position k of the buffer is occupied by datum d. The data type *Buffer* is specified by the following two constructors, where [] denotes the empty buffer:

$$[] :\rightarrow Buffer$$
$$in : \Delta \times Nat \times Buffer \rightarrow Buffer$$

For $k \in \{0, \ldots, 2n-1\}$, let $succmod(k)$ denote the equivalence class of the successor of k modulo $2n$; in particular, $succmod(2n-1) = 0$. The operator $succmod : Nat \rightarrow Nat$ is defined by the following equation:

$$succmod(k) \;=\; if(eq(S(k), 2n), 0, S(k))$$

Here, $if : Bool \times Nat \times Nat \rightarrow Nat$ is an if-then-else function (i.e., $if(\mathtt{T}, k, \ell) = k$ and $if(\mathtt{F}, k, \ell) = \ell$), and $eq : Nat \times Nat \rightarrow Bool$ is the equality function on natural numbers from Section 2.3.

The specification of the SWP uses some auxiliary functions on buffers. $remove(k, t)$ is obtained by emptying position k in buffer t. $add(d, k, t)$ is

obtained by placing datum d at position k in buffer t, at the same time emptying position k in t. $retrieve(k,t)$ produces the datum that resides at position k in buffer t (if this position is occupied). And $test(k,t)$ produces $\mathtt{T}$ if and only if position k in t is occupied. These four functions are defined by:

$$\begin{aligned}
&remove(k,[]) &&= []\\
&remove(k,in(d,\ell,t)) &&= if(eq(k,\ell),t,in(d,\ell,remove(k,t)))\\
&add(d,k,t) &&= in(d,k,remove(k,t))\\
&retrieve(k,in(d,\ell,t)) &&= if(eq(k,\ell),d,retrieve(k,t))\\
&test(k,[]) &&= \mathtt{F}\\
&test(k,in(d,\ell,t)) &&= if(eq(k,\ell),\mathtt{T},test(k,t))
\end{aligned}$$

The second equation of *remove* implicitly assumes that there is at most one datum at each position in a buffer. For example, a buffer should not be of the form $in(d,k,in(e,k,t))$, because $remove(k,in(d,k,in(e,k,t)))$ would produce the erroneous result $in(e,k,t)$. In our specification of the SWP there is no such overloading of positions in buffers, because the function *add* is used instead of *in*, any time when a datum is placed into the receiving buffer. This is essential to obtain a finite state space, as otherwise some pair (d,k) could be added to the receiving buffer an unbounded number of times.

For $k,\ell \in \{0,\ldots,2n-1\}$, $release(k,\ell,t)$ is obtained by emptying positions k up to ℓ in t, modulo $2n$. That is, if $k \leq \ell$ then positions $k,\ldots,\ell-1$ are emptied, while if $\ell < k$ then positions $k,\ldots,2n-1$ and $0,\ldots,\ell-1$ are emptied. *release* is defined by:

$$release(k,\ell,t) = if(eq(k,\ell),t,release(succmod(k),\ell,remove(k,t)))$$

Actually, in the case of innermost rewriting (see Section 2.2), the algebraic specification above does not terminate. This is due to the fact that its left-hand side can be applied to the subterm $release(succmod(k),\ell,remove(k,t))$ in its right-hand side. This problem can be solved by splitting the equation for *release* into two cases, depending on whether the buffer in its third argument is empty or not:

$$\begin{aligned}
&release(k,\ell,[]) &&= []\\
&release(k,\ell,in(d,m,t)) &&=\\
&\quad if(eq(k,\ell),in(d,m,t),release(succmod(k),\ell,remove(k,in(d,m,t))))
\end{aligned}$$

In the specification of the SWP below, for clarity of presentation, two of the conditions within the conditional operator $\triangleleft\_\triangleright$ are formulated in natural language. For an algebraic formulations of these conditions, see Exercise 50. For $k,\ell \in \{0,\ldots,2n-1\}$, with the *range* from k to ℓ (modulo $2n$) we mean either $(\ell-k)+1$ (i.e., the number of elements in $\{k,\ldots,\ell\}$) if $k \leq \ell$, or $(2n-k)+\ell+1$ (i.e., the number of elements in $\{k,\ldots,2n-1\} \cup \{0,\ldots,\ell\}$) if $\ell < k$.

The Sender is modelled by the process term $X(\mathit{first\text{-}in}, \mathit{first\text{-}empty}, t)$, where t represents the sending buffer of size $2n$, *first-in* the first position in the window of t, and *first-empty* the first empty position in (or just outside) the window of t.

$$X(\mathit{first\text{-}in}{:}\mathit{Nat}, \mathit{first\text{-}empty}{:}\mathit{Nat}, t{:}\mathit{Buffer})$$

$$= \sum_{d:\Delta} r_{\mathrm{A}}(d) \cdot X(\mathit{first\text{-}in}, \mathit{succmod}(\mathit{first\text{-}empty}), \mathit{in}(d, \mathit{first\text{-}empty}, t))$$
$$\triangleleft \text{ 'the range from } \mathit{first\text{-}in} \text{ to } \mathit{first\text{-}empty} \text{ does not exceed } n\text{'} \triangleright \delta$$

$$+ \sum_{k:Nat} s_{\mathrm{B}}(\mathit{retrieve}(k, t), k) \cdot X(\mathit{first\text{-}in}, \mathit{first\text{-}empty}, t)$$
$$\triangleleft \mathit{test}(k, t) \triangleright \delta$$

$$+ \sum_{k:Nat} r_{\mathrm{F}}(k) \cdot X(k, \mathit{first\text{-}empty}, \mathit{release}(\mathit{first\text{-}in}, k, t))$$

The specification of the Receiver uses a function $\mathit{next\text{-}empty}(k, t)$, producing the first empty position in t starting from k, modulo $2n$:

$$\mathit{next\text{-}empty}(k, t) = \mathit{if}(\mathit{test}(k, t), \mathit{next\text{-}empty}(\mathit{succmod}(k), t), k)$$

Again, in the case of innermost rewriting, the algebraic specification above does not terminate. This is due to the fact that its left-hand side can be applied to the subterm $\mathit{next\text{-}empty}(\mathit{succmod}(k), t)$ in its right-hand side. This problem can be solved by adapting the equation for *next-empty* to:

$$\begin{aligned}
&\mathit{next\text{-}empty}(k, []) &&= k \\
&\mathit{next\text{-}empty}(k, \mathit{in}(d, \ell, t)) &&= \\
&\quad \mathit{if}(\mathit{test}(k, \mathit{in}(d, \ell, t)), \mathit{next\text{-}empty}(\mathit{succmod}(k), \mathit{remove}(k, \mathit{in}(d, \ell, t))), k)
\end{aligned}$$

We proceed to specify the Receiver, which is modelled by the process term $Y(\mathit{first\text{-}in}, t)$, where t represents the receiving buffer of size $2n$, while *first-in* represents the first position in the window of t.

$$Y(\mathit{first\text{-}in}{:}\mathit{Nat}, t{:}\mathit{Buffer})$$

$$= \sum_{d:\Delta} \sum_{k:Nat} r_{\mathrm{C}}(d, k) \cdot (Y(\mathit{first\text{-}in}, \mathit{add}(d, k, t))$$
$$\triangleleft \text{ 'the range from } \mathit{first\text{-}in} \text{ to } k \text{ does not exceed } n\text{'} \triangleright Y(\mathit{first\text{-}in}, t))$$

$$+ s_{\mathrm{D}}(\mathit{retrieve}(\mathit{first\text{-}in}, t)) \cdot Y(\mathit{succmod}(\mathit{first\text{-}in}), \mathit{remove}(\mathit{first\text{-}in}, t))$$
$$\triangleleft \mathit{test}(\mathit{first\text{-}in}, t) \triangleright \delta$$

$$+ s_{\mathrm{E}}(\mathit{next\text{-}empty}(\mathit{first\text{-}in}, t)) \cdot Y(\mathit{first\text{-}in}, t)$$

Finally, we specify the mediums K and L, which may lose messages between the Sender and the Receiver, and vice versa.

$$K = \sum_{d:\Delta} \sum_{k:Nat} r_{\mathrm{B}}(d,k)\cdot(j\cdot s_{\mathrm{C}}(d,k) + j)\cdot K$$

$$L = \sum_{k:Nat} r_{\mathrm{E}}(k)\cdot(j\cdot s_{\mathrm{F}}(k) + j)\cdot L$$

The initial state of the SWP is expressed by

$$\tau_I(\partial_H(X(0,0,[]) \parallel Y(0,[]) \parallel K \parallel L))$$

where the set H consists of the read and send actions over the internal channels B, C, E, and F, while the set I consists of the communication actions over these internal channels together with j.

The desired external behaviour of the SWP is that data elements that are read from channel A by the Sender are sent into channel D by the Receiver in the same order. In other words, the process term above is intended to be a solution for $Z([])$ in the process declaration

$$\begin{aligned} Z(\lambda{:}List) = {} & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot Z(append(d,\lambda)) \lhd length(\lambda) < 2n \rhd \delta \\ & + s_{\mathrm{D}}(head(\lambda))\cdot Z(tail(\lambda)) \lhd nonempty(\lambda) \rhd \delta \end{aligned}$$

See Exercise 4 for specifications of the data type *List* and for the functions *append*, *length*, *head*, *tail* and *nonempty*. Note that the action $r_{\mathrm{A}}(d)$ can be performed until the list λ contains $2n$ elements, because in that situation the windows of the sending and of the receiving buffer must both contain the maximum number of n elements.

In Exercise 51 it is asked to specify the SWP in the μCRL toolset, including all data types. Then one can provide concrete, finite instantiations for Δ and n, and generate the state space. After minimisation modulo branching bisimilarity, the resulting state space should correspond with the desired external behaviour. A symbolic correctness proof of the SWP is given in [5]. That proof is based on the cones and foci method [47], which will be explained in Section 7.3.

In the *two-way* SWP [25, 30], not only the Sender reads data elements from channel A and passes them on to the Receiver, but also the Receiver reads data elements from channel D and passes them on to the Sender. In the two-way SWP, the Sender has two buffers, one to store incoming data elements from channel A, and one to store incoming data elements from channel F; likewise for the Receiver. Note that in the two-way SWP, the Sender and the Receiver are symmetric identities, and likewise for the mediums K and L.

In the two-way SWP, acknowledgements that are sent from the Sender to the Receiver, and vice versa, can get a free ride by attaching them to data elements. This technique, which is commonly known as *piggybacking*, promotes a better use of available bandwidth. A symbolic correctness proof of the two-way SWP with piggybacking is given in [4].

Piggybacking can slow down the two-way SWP, since an acknowledgement may have to wait for a long time before it can be attached to a data element. Therefore, the Sender and the Receiver ought to be supplied with a timer (cf.

the BRP), which sends a time-out message if an acknowledgement must be sent out without further delay; see [102] for more details.

5.4 Tree Identify Protocol

The IEEE 1394 (FireWire) standard [72] contains protocols for connecting devices, in order to carry all forms of digital video and audio quickly, reliably, and inexpensively. Its architecture is scalable, and it is 'hot-pluggable', meaning that devices can be added or removed easily at any time. The topology of the network should be a connected, acyclic graph with bidirectional channels.

Much effort has been spent on the description and verification of various parts of this standard, using different formalisms and proof techniques. For example, the operation of sending packets of information across the network is described using μCRL in [78], and using E-LOTOS in [100]. The former is essentially a description only, with five correctness properties stated informally, but not formalised or proved. The latter is based on the μCRL description, adding another layer of the protocol and carrying out a verification using the toolset CADP.

In this section we concentrate on the tree identify phase of the physical layer, which occurs after a bus reset in the system, which can for instance happen when a node is added to or removed from the network. The purpose of the tree identify protocol (TIP) is to assign a (new) *root*, or leader, to the network. Essentially, this protocol consists of a set of negotiations between nodes to establish the direction of the parent–child relationship. Thus, from a connected, bidirectional, acyclic graph it creates a connected, unidirectional spanning tree towards a root node. Potentially, a node can be a parent to many nodes, but a child of at most one node. The node with no parent (after the negotiations are complete) is the root. The TIP must ensure that exactly one root is chosen.

We present two specifications of the TIP in μCRL from [99]: one with synchronous and one with asynchronous communication. They were derived with reference to the transition diagram in Section 4.4.2.2 of the standard [72]. If the network is connected, and there are no cycles, then the specifications of both the synchronous and the asynchronous version of the TIP ultimately produce one root. A symbolic verification of the synchronous version of the protocol, based on the cones and foci method, will be presented in Section 7.4. See [99] for a symbolic verification of the asynchronous version of the protocol.

The two μCRL specifications that we present for the TIP do not take into account timing aspects and probabilities. For a formal specification of the TIP using I/O automata, and for an analysis of timing aspects and probabilities, see [43, 97, 101].

Implementation A: Synchronous Communication

We assume a network, consisting of a collection of nodes and bidirectional channels between nodes. The aim of the TIP is to establish parent–child relations between neighbouring nodes. The final structure should be a tree, directed towards the root of the network.

In order to establish parent–child relations, a node can send a parent request to a neighbouring node, asking that node to become its parent; a parent request from node i to node j is represented by the action $s(i,j)$, which communicates with the read action $r(i,j)$ to $c(i,j)$. We first assume that communication is *synchronous*, so that a parent request from the sending node is instantly read by the receiving node; in other words, $c(i,j)$ establishes a child–parent relation between the nodes i and j. As communication between nodes is synchronous, there is no need for acknowledgements.

Each node keeps track of the neighbours from which it has not yet received a parent request; of course, initially this list consists of all neighbours. If a node i is the parent of all its neighbours except for one node j, then i is allowed to send a parent request to j. In the case that a node received parent requests from all its neighbours, this node declares itself the root of the network, by performing the action $leader(i)$.

Apart from the standard data types of booleans and of natural numbers, the μCRL specification of Implementation A, which is given below, includes data types for nodes, for lists of nodes and for states. The latter data type consists of elements 0 and 1, where a node is in state 0 if it is looking for a parent, and in state 1 if it has a parent or is the root. The process term $X(i,p,s)$ represents node with identity i, in state s, with p as list of possible parents.

$$\begin{aligned}
&X(i{:}Node, p{:}Nodelist, s{:}State)\\
&= \textstyle\sum_{j:Node} r(j,i)\cdot X(i,p\backslash\{j\},s) \lhd \mathtt{T} \rhd \delta\\
&+ \textstyle\sum_{j:Node} s(i,j)\cdot X(i,p,1) \lhd p = \{j\} \wedge s = 0 \rhd \delta\\
&+ leader(i)\cdot X(i,p,1) \lhd p = [] \wedge s = 0 \rhd \delta
\end{aligned}$$

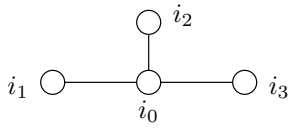
The initial state of Implementation A consists of the merge of the node processes for the nodes $i_1,\ldots,i_k$ in state 0, where for each node i_m the possible parents list p_m consists of all its neighbours:

$$\tau_I(\partial_H(X(i_1,p_1,0) \parallel \cdots \parallel X(i_k,p_k,0)))$$

Here, H consists of all read and send actions between neighbours, while I consists of all communication actions between neighbours. Note that the topology of the network is recorded by the lists $p_1,\ldots,p_k$.

If a network is connected and free of cycles, then the μCRL specification of the synchronous version of the TIP always produces one root; see Section 7.4 for a formal proof of this fact.

Example 12. The network



is captured by

$$\tau_I(\partial_H(X(i_0,\{i_1,i_2,i_3\},0) \,\|\, X(i_1,\{i_0\},0) \,\|\, X(i_2,\{i_0\},0) \,\|\, X(i_3,\{i_0\},0)))$$

where H consists of all read and send actions between i_0 and the other three nodes, while I consists of all communication actions between i_0 and the other three nodes. The state space of this network, generated using the μCRL and CADP toolsets, is depicted in Fig. 5.6.

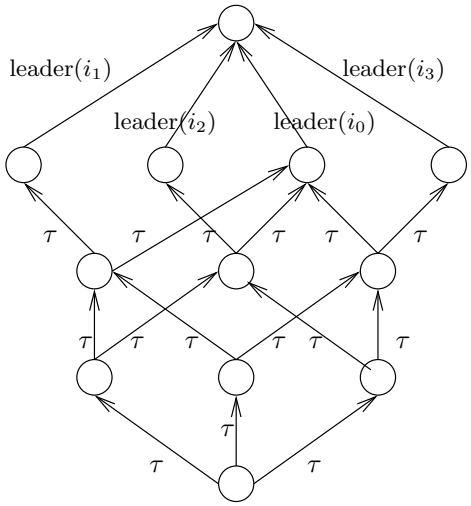


Fig. 5.6. External behaviour of the TIP

Implementation B: Asynchronous Communication

Implementation A assumes synchronous communication between nodes. Now we move to a setting where communication is *asynchronous*. Then two neighbouring nodes can simultaneously send parent requests to each other. In the IEEE 1394 standard, such a situation, called *root contention*, is resolved using a probabilistic approach. If two nodes are in root contention, then with probability 0.5 a node resends a parent request, or with probability 0.5 a node waits for a certain period of time whether it receives a parent request. Root

contention is resolved if one node resends a parent request while the other node waits to receive a parent request.

In Implementation B, one-place buffers are introduced to model asynchronous communication between nodes. There are two buffers for each pair of neighbours, one buffer for each direction. Since communication is asynchronous, it is no longer guaranteed that a parent request will be accepted, as opposite parent requests may cross each other. Therefore, parent requests, which carry the label *req*, are acknowledged by an opposite message carrying the label *ack*. There are send, read and communication messages from nodes to buffers, denoted by s', r' and c', and from buffers to nodes, denoted by s, r and c.

Again, individual nodes of the network are specified as separate processes, which are put in parallel with the one-place buffers, and the topology of the network is captured in the initial possible parents lists of the nodes. In Implementation B, a node can be in five different states, which can roughly be described as follows:

0: receiving parent requests;

1: sending acknowledgements, followed by sending a parent request or performing a *leader* action;

2: waiting for an acknowledgement;

3: root contention;

4: finished.

The relations between the five states of a node are depicted in Fig. 5.7. In state 0 the node is receiving parent requests. If all but one of its neighbours have become its children, the node can move to state 1 by sending an acknowledgement. Alternatively, if all of its neighbours have become its children, the node moves to state $1'$. In the special case that the node has only one neighbour, it sends a parent request to this neighbour straight away and moves to state 2. In states 1 and $1'$ the node sends all outstanding acknowledgements; in state 1 the node can at any time receive a parent request from the remaining neighbour and move to state $1'$. As states 1 and $1'$ are very similar, in the forthcoming μCRL specification they will be collapsed. When in state $1'$ all acknowledgements have been sent, the node emits a *leader* action to move to the final state 4. By contrast, if in state 1 all acknowledgements have been sent, the node sends a parent request to the remaining neighbour and moves to state 2. If in state 2 an acknowledgement is received, then the node moves to the final state 4. Alternatively, if in state 2 a parent request is received, then the node moves to state 3 to resolve the root contention with its remaining neighbour. Each of the two nodes in contention throws up a virtual coin, and with probability 0.5 either it resends a parent request or it waits for a fixed amount of time whether it receives a parent request. If within this period no parent request is received, then the two nodes throw up their coins once more. As we do not model timing or probability aspects, this means that in state 3

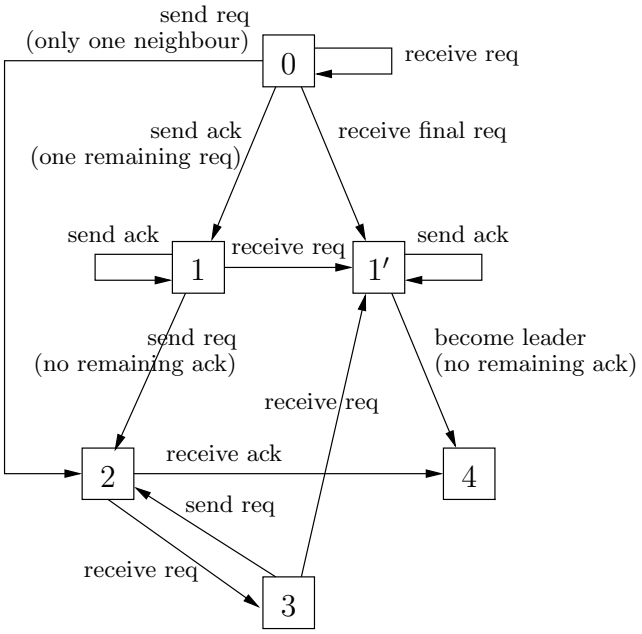


Fig. 5.7. Relations between the five states

the node either resends a parent request and returns to state 2, or receives a parent request and moves to state $1'$ to send an acknowledgement followed by a *leader* action. Owing to fairness (see Section 4.4), ultimately root contention will be resolved.

In the μCRL specification of Implementation B, the node with identity i, in state s, is represented by the process term $X(i, p, q, s)$, with p as list of possible parents, and with q as list of neighbours to which it must still send an acknowledgement. Lists of node identities are built from the standard constructors [] and *in* (see Exercise 4). Moreover, the specification includes the following functions:

- $remove(j, p)$ removes node j from list p;
- $single(p)$ tests whether list p contains exactly one element, while $single(p, j)$ tests whether list p consists of a single node j;
- $test(j, p)$ tests whether node j occurs in list p;
- $empty(p)$ tests whether list p is empty.

(In Exercise 56 you are asked to specify these functions.)

$$
\begin{aligned}
&X(i{:}Node, p{:}Nodelist, q{:}Nodelist, s{:}State)\\
&= \sum_{j:N} r(j,i,req)\cdot X(i, remove(j,p), in(j,q), if(single(p),1,0)) \\
&\qquad\qquad \lhd\ s = 0 \ \rhd\ \delta \\
&+ \sum_{j:N} s'(i,j,ack)\cdot X(i,p,remove(j,q),1) \\
&\qquad\qquad \lhd\ single(p) \wedge test(j,q) \wedge s = 0 \ \rhd\ \delta \\
&+ \sum_{j:N} s'(i,j,req)\cdot X(i,p,q,2) \lhd\ single(p,j) \wedge empty(q) \wedge s = 0 \rhd \delta \\
&+ \sum_{j:N} s'(i,j,ack)\cdot X(i,p,remove(j,q),1) \lhd\ test(j,q) \wedge s = 1 \rhd \delta \\
&+ \sum_{j:N} s'(i,j,req)\cdot X(i,p,q,2) \lhd\ single(p,j) \wedge empty(q) \wedge s = 1 \rhd \delta \\
&+ leader(i)\cdot X(i,p,q,4) \lhd\ empty(p) \wedge empty(q) \wedge s = 1 \rhd \delta \\
&+ \sum_{j:N} r(j,i,req)\cdot X(i,[],in(j,q),1) \lhd\ s = 1 \rhd \delta \\
&+ \sum_{j:N} r(j,i,ack)\cdot X(i,p,q,4) \lhd\ s = 2 \rhd \delta \\
&+ \sum_{j:N} r(j,i,req)\cdot X(i,p,q,3) \lhd\ s = 2 \rhd \delta \\
&+ \sum_{j:N} r(j,i,req)\cdot X(i,[],p,1) \lhd\ s = 3 \rhd \delta \\
&+ \sum_{j:N} s'(i,j,req)\cdot X(i,p,q,2) \lhd\ single(p,j) \wedge s = 3 \rhd \delta
\end{aligned}
$$

The (unidirectional) channel from node i to node j is specified as a one-place buffer:

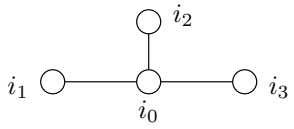
$$
\begin{aligned}
B(i{:}Node, j{:}Node) = {}& r'(i,j,req)\cdot s(i,j,req)\cdot B(i,j) \\
&+ r'(i,j,ack)\cdot s(i,j,ack)\cdot B(i,j)
\end{aligned}
$$

The initial state of Implementation B consists of the merge of the channel processes together with the node processes for the nodes $i_1, \ldots, i_k$ in state 0, where for each node i_m the possible parents list p_m consists of all its neighbours, and the list of neighbours that need to be acknowledged is empty:

$$\tau_I(\partial_H(X(i_1,p_1,[],0) \parallel \cdots \parallel X(i_k,p_k,[],0) \parallel B(i_\ell,i_{\ell'}) \parallel \cdots \parallel B(i_m,i_{m'})))$$

Here, H consists of all read and send actions between nodes and buffers, I consists of all communication actions between nodes and buffers, and the $(i_\ell, i_{\ell'}), \ldots, (i_m, i_{m'})$ are pairs of neighbours.

Example 13. The network



is captured by

$$\tau_I(\partial_H(\\ X(i_0,\{i_1,i_2,i_3\},[],0) \parallel X(i_1,\{i_0\},[],0) \parallel X(i_2,\{i_0\},[],0) \parallel X(i_3,\{i_0\},[],0)\\ \parallel B(i_0,i_1) \parallel B(i_0,i_2) \parallel B(i_0,i_3) \parallel B(i_1,i_0) \parallel B(i_2,i_0) \parallel B(i_3,i_0)))$$

where H consists of all read and send actions between nodes and buffers, while I consists of all communication actions between nodes and buffers. The external behaviour of this network, i.e., its state space after minimisation modulo branching bisimilarity, is depicted in Fig. 5.6.

5.5 Movable Patient Support for an MRI Scanner

The best way to get well-acquainted with formal specification and analysis, is by applying it from scratch in the design of a distributed system. In this section a Movable Patient Support Platform is sketched, MPSP for short, which is a trolley bed on which a patient can lie inside a medical scanning machine for magnetic resonance imaging. This sketch is based on such a system that has been developed at Philips Medical Systems. The intention is that the MPSP can serve as a student assignment for applying formal verification during system design (see Exercise 58).

The MPSP can be either disconnected from the scanner or *docked*, i.e., connected to the scanner. The MPSP can be disconnected from the scanner, to make it possible that a patient is prepared in a separate room, while another patient is being scanned. Currently patients are prepared while in or near the scanner, and during this time the scanner is idle.

There are two motors in the MPSP. Motor M1 controls the vertical and motor M2 controls the horizontal movement. Both motors have brakes that can be turned on and off separately from the motors. Horizontal movement is only allowed when the MPSP is docked. When the MPSP is disconnected from the scanner, the bed must always be in the rightmost position (which is detected by a sensor), as otherwise the MPSP might tumble over. When disconnected from the scanner, the horizontal brake must always be applied. The vertical brake must always be applied while the vertical motor is off. When a motor is on, the corresponding brake must not be used, as otherwise the motor could overheat.

The movements are controlled via a console on the MPSP. The Stop button puts the MPSP in emergency mode, in which the brakes must be released. This allows medical staff to manually drag the patient outside the scanner, in case an emergency occurs (e.g., a heart attack while scanning, or a system malfunction). The Resume button puts the MPSP back to normal operating mode.

The Undock button can be used to disconnect the MPSP from the scanning device. When the Undock button is pressed, a message is sent to the scanner,

which will undock the MPSP. For this a gentle spring mechanism is used that pushes the MPSP away from the scanner. The undock message should never be sent to the scanner if the bed is not in the rightmost position, as otherwise the MPSP might tumble over.

The reset button is used for calibration. Every scanner can have a different height, generally dependent on how it is installed in the hospital. The MPSP can only be moved inside the scanner if scanner and bed are at the same height, which is called the standard height. Before use, the MPSP must be calibrated by setting the standard height. This is done as follows. The MPSP is docked, which is detected via a sensor in the docking unit D. Then, using the Up and Down buttons, the bed is moved to the correct height. By pressing the reset button once, this height is set to be the standard height. If the reset button is pressed while the MPSP is not docked, the standard height is forgotten and the MPSP goes into uncalibrated mode.

When the MPSP is docked and calibrated, the bed halts when it has reached the standard height. If the up button is pressed at the standard height, the MPSP moves into the scanner. In order to avoid unexpected movements it is important that the up button is released before the inward movement is commenced. This means that releasing the Up and Down buttons are important interface actions.

When the MPSP is docked and calibrated and the Down button is pressed, the bed moves outward, until an outward horizontal sensor indicates that the bed is completely outside the scanner. By releasing and pressing the Down button again, the bed subsequently moves downwards. While the MPSP is docked and calibrated, the bed cannot be moved above the standard height.

When the MPSP is disconnected or uncalibrated, the Up and Down buttons can only be used to move the bed up and down, respectively. The bed is not allowed to move above some uppermost and below some lowermost position. There are two sensors, to detect when the bed is in the uppermost or lowermost position.

In Exercise 58 it is asked to design a set of controllers that must operate the MPSP in such a way that no harm can ever be done to a patient or to the equipment. Also it is asked to formulate requirements for the MPSP, and to verify those requirements using the μCRL and CADP toolsets. Beware that requirements should not be formulated too general. For example, consider a requirement 'the bed can move up, down, left or right'. Here the question remains in which states of the system such movements are possible, and under which inputs from the environment. Furthermore, a requirement like 'the controllers must communicate asynchronously' cannot be formulated in temporal logic, as it does not involve inputs and outputs of the system.

The actual platform is much more complex than the one described here. For instance, there are at least three emergency modes, the platform can also be controlled via a console on the scanner, or via an operator on a host computer. In all cases the platform can respond differently.

Exercises

Exercise 45. Give a μCRL specification of the ABP together with its data types. Use the μCRL toolset to generate a `.tbf` file, for the case that $\Delta = \{d_1, d_2\}$. Generate a state space from this `.tbf` file, using the μCRL and CADP toolsets. Minimise this state space modulo branching bisimilarity, using the CADP toolset. (See Appendix A for information on how to use the μCRL and CADP toolsets.)

Exercise 46. Suppose that in the ABP the Sender would not attach an alternating bit to data elements, and that the Receiver would only send one kind of acknowledgement. Show with the help of the μCRL toolset that in that case data elements could get lost.

Exercise 47. Give a μCRL specification of the BRP together with its data types. Generate a `.tbf` file, for the case that data packets have length three, and the maximum number of retries is four. Generate a state space from this `.tbf` file. Minimise this state space modulo branching bisimilarity.

Exercise 48. Suppose that the timer T_1 is absent. Verify using the CADP toolset that this version of the BRP contains a deadlock. Give an execution trace to a deadlock state (in absence of the hiding operator).

Exercise 49. What would go wrong if the timer T_2 were absent from the BRP? Generate the state space of this protocol. Does the resulting protocol contain a deadlock?

Exercise 50. Define a function $\mathit{plusmod} : \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$, where for $k, \ell \in \{0, \ldots, 2n-1\}$ $\mathit{plusmod}(k, \ell)$ produces the equivalence class of $k + \ell$ modulo $2n$ in $\{0, \ldots, 2n-1\}$.

Also define a function $\mathit{ordered} : \mathit{Nat} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$, where $\mathit{ordered}(k, \ell, m)$ produces `T` if and only if ℓ lies in the range from k to $m-1$, modulo $2n$; that is, if $k \leq m < 2n$ then $\ell \in \{k, \ldots, m-1\}$, and if $m < k < 2n$ then $\ell \in \{k, \ldots, 2n-1\} \cup \{0, \ldots, m-1\}$.

Use the functions *plusmod* and *ordered* to give an algebraic formulation of the condition 'the range from k to ℓ does not exceed m', which appears in the μCRL specification of the SWP.

Exercise 51. Give a μCRL specification of the SWP and its data types. Use the μCRL and CADP toolsets to analyse this specification, for the case that $\Delta = \{d_1, d_2\}$ and $n = 2$. In particular, use the minimisation and model checking techniques that are explained in Chapter 6.

Exercise 52. Consider the SWP with buffer size is three and window size is two. Give an execution trace of $\partial_H(X(0, 0, []) \parallel Y(0, []) \parallel K \parallel L)$ in which a datum is erroneously sent out via channel D more than once.

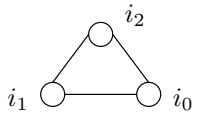
Exercise 53. Give a μCRL specification of the two-way SWP. Use renaming (see Section 3.9) to extract the Receiver and L from the recursive equations of the Sender and K, respectively.

Exercise 54. Give a μCRL specification of the Sender in the two-way SWP with piggybacking.

Exercise 55. Explain why there are non-inert τ-transitions in the external behaviour depicted in Fig. 5.6.

Exercise 56. Give a μCRL specification of the asynchronous version of the TIP and its data types. Use the μCRL and CADP toolsets to analyse this specification, for the network depicted in Example 13. In particular, use the minimisation and model checking techniques that are explained in Chapter 6.

Exercise 57. Specify the initial state of the synchronous version of the TIP, for the network



Explain why for this network, the TIP does not exhibit any behaviour.

Exercise 58. Designing a set of controllers for the patient support system described in Section 5.5. Carry out the following steps:

1. List the interactions of the control system with the outside world. These are for instance 'turning on a motor', 'reading that the Up button is released' and 'applying a brake'. Describe clearly but compactly the meaning of each interaction in words.
2. Formulate the system requirements for the patient support system in terms of these interactions. Identify both *safety* requirements to express that something bad will never happen (e.g., when motor M1 is on, its brake must not be applied), and *liveness* requirements to express that something good will eventually happen (e.g., when the system is in uncalibrated mode, the bed is not in the uppermost position, and the Up button is pressed, then the bed must go up).
3. Depict an architecture for the control system.
4. Specify the concurrent components in the architecture in μCRL.
5. Verify, using the μCRL and CADP toolsets, whether all your requirements are valid with respect to your μCRL specification. Apply the model checking techniques that are explained in Section 6.6.
6. If not all requirements are satisfied, then modify the μCRL specification, and verify the requirements again.

The description in Section 5.5 is underspecified. This means that in certain cases you have the freedom to make your own design decisions.

Write a report that covers all items above. This report should be a concise technical account of the system, from which the requirements, architecture and design, system behaviour, and action interface can be easily understood. It should also be made clear how the requirements have been verified.

6

Verification Algorithms on State Spaces

In this chapter explain how a μCRL specification can be turned into a state space. Moreover, we present some of the ideas and algorithms that underlie the transformation and analysis of the resulting state spaces. These algorithms are supported by the μCRL and CADP toolsets.

6.1 Linear Process Equations

A *linear process equation* (LPE) [16] is a one-line process declaration that consists of actions, summations, sequential compositions and conditionals. In particular, an LPE does not contain any parallel operators, encapsulations or hidings. In essence an LPE is a vector of data parameters together with a list of summands consisting of a condition, action and effect triple, describing when an action may happen and what is its effect on the vector of data parameters. This format resembles I/O automata [80], extended finite state machines [73], Unity processes [33] and STGA [76].

μCRL specifications can be transformed into an LPE (see Section 6.2). Since LPEs do not contain parallel operators, there is a strong relation between an LPE and its corresponding state space. A μCRL specification consisting of about ten concurrent components typically gives rise to an LPE of only several hundreds of summands. And the state space that is generated from the LPE (see Section 6.3) may consist of billions of states. Even in cases where the state space is too large to allow generation, or even infinite, the LPE can generally be obtained without much effort.

LPEs will play a central role. We will explain proof techniques for LPEs (Sections 7.1, 7.2 and 7.3), and transformations of LPEs to arrive at a smaller state space or at a simplified LPE (Sections 7.5 and 7.6).

Definition 4 (Linear process equation). *A* linear process equation *(LPE) is a process declaration of the form*

$$X(d{:}D) \;=\; \sum_{i:I} \sum_{e:E} a_i(f_i(d,e)) \cdot X(g_i(d,e)) \lhd h_i(d,e) \rhd \delta$$

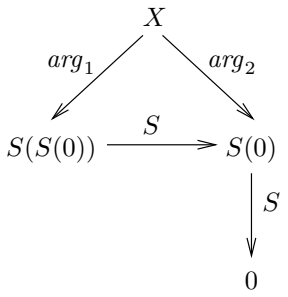
where $a_i \in \mathsf{Act} \cup \{\tau\}$, $f_i : D \times E \to D_i$, $g_i : D \times E \to D$ *and* $h_i : D \times E \to \mathit{Bool}$.

Intuitively, in an LPE written as in Definition 4, the different states of the corresponding state space are represented by the data parameter $d{:}D$. Note that D may be a Cartesian product of n data types, meaning that d may actually be a tuple $(d_1, \ldots, d_n)$ (so it would actually be more precise to write $\mathbf{d}$ instead of d). The LPE expresses that in state d one can perform actions a_i, carrying a data parameter $f_i(d,e)$, under the condition that $h_i(d,e)$ is true; in this case the resulting state is $g_i(d,e)$.

The data type E helps to give LPEs a more general form, as not only the data parameter $d{:}D$, but also the data parameter $e{:}E$ can influence the parameter of the action a_i, the condition h_i and the resulting state g_i. Typically, a data parameter e in an LPE is used to let a send or read action range over a data domain. See for example the process declaration of a queue of capacity two in Section 4.3. Again, E can be a Cartesian product of data types.

LPEs are stored using the maximal sharing method offered by ATerms, as explained at the end of Section 2.1.

Example 14. The process term $X(S(S(0)), S(0))$ is stored as follows.



6.2 Linearisation

The μCRL toolset offers two linearisation algorithms to transform a μCRL specification into an LPE: the default method and a so-called regular method. A formal treatment of the default method of linearisation is given in [60]. The linearisation algorithm from [60] underlies the `mcrl` command of the μCRL toolset, while the regular method underlies `mcrl -regular` (see Appendix A).

We start with discussing the required input for these linearisation algorithms, called *parallel pCRL*, which is a subset of μCRL. (The p in pCRL stands for 'pico', as the μ in μCRL stands for 'micro'.) Basically, in parallel pCRL, actions, summations, sequential compositions and conditionals are

used to build basic processes terms, to which the merge, encapsulation and hiding can be applied. Two types of recursion variables X are distinguished, on the basis of their recursive equations $X(d_1{:}D_1, \ldots, d_n{:}D_n) = p$:

type I: p contains only $\cdot$, $+$, $\lhd b \rhd$, $\sum_{d:D}$;
type II: p also contains $\|$, ∂_H, τ_I, ρ_f.

In parallel pCRL, recursion variables of type II must not be used recursively, meaning that they can always be eliminated from right-hand sides of recursive equations.

Example 15. Consider the two two-bit buffers in sequence from Exercise 37:

$$\begin{aligned} B_1 &= \textstyle\sum_{d:D} r_1(d)\cdot s_3(d)\cdot B_1 \\ B_2 &= \textstyle\sum_{d:D} r_3(d)\cdot s_2(d)\cdot B_2 \\ C &= \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel B_2)) \\ D &= \tau_{\{c_4\}}(\partial_{\{s_4,r_4\}}(\rho_{s_2\to s_4}(C) \parallel \rho_{r_1\to r_4}(C))) \end{aligned}$$

B_1 and B_2 are of type I, while C and D are of type II. This process declaration is within parallel pCRL. Namely, the occurrences of the type II recursion variable C in the right-hand side of the recursive equation of D can be eliminated by replacing them by the right-hand side of the recursive equation of C.

We first explain, by means of examples, how a process declaration with only type I recursion variables is linearised, according to the default method and according to the regular method. In the default method, process terms that remain to be executed are pushed on a stack.

Example 16. The recursion variable Y in the process declaration below is of type I:

$$Y = a\cdot Y\cdot b + c$$

Y performs k a's, then a c, and then k b's, for any $k \geq 0$. The default method linearises this process declaration as follows.

Step 1: Make a so-called Greibach Normal Form, by replacing non-initial occurrences of actions by fresh recursion variables.

$$\begin{aligned} Y &= a\cdot Y\cdot Z + c \\ Z &= b \end{aligned}$$

Step 2: Linearisation.

List is a data type of stacks that can contain recursion variables and their data parameters (see Exercise 4). The empty list [] and $in : D \times List \to List$ are the constructors of *List*. Furthermore, $nonempty : List \to Bool$ checks

whether a list is non-empty, while $head : List \to D$ and $tail : List \to List$ produce the head and tail of a (non-empty) list. In this example, $D = \{Y, Z\}$.

The two recursive equations above are joined in a single recursive equation, using a list over $\{Y, Z\}$, which acts as a stack on which the recursion variables are pushed that remain to be executed. Moreover, the non-linear subterm $Y \cdot Z$ in the right-hand side of the first recursive equation is linearised by pushing Y and Z onto the stack. When the stack becomes empty, the process terminates successfully.

$$\begin{aligned} X(\lambda{:}List) = {} & a \cdot X(in(Y, in(Z, tail(\lambda)))) \lhd eq(head(\lambda), Y) \rhd \delta \\ & + (c \cdot X(tail(\lambda)) \lhd nonempty(tail(\lambda)) \rhd c) \lhd eq(head(\lambda), Y) \rhd \delta \\ & + (b \cdot X(tail(\lambda)) \lhd nonempty(tail(\lambda)) \rhd b) \lhd eq(head(\lambda), Z) \rhd \delta \end{aligned}$$

The initial state Y of the original process declaration is represented by $X(in(Y, []))$.

In Example 16, Y and Z do not carry data parameters, so that D consists only of $\{Y, Z\}$. In case recursion variables carry data parameters, their arguments must also be pushed onto the stack (see Exercise 60).

The list data type that is used in the default linearisation method gives a lot of overhead. The *regular* linearisation method follows a different approach: a non-linear process term in the right-hand side of a recursive equation is replaced by a fresh recursion variable. By recording this replacement, it is avoided that the same non-linear process term is replaced multiple times by different fresh recursion variables.

Example 17. The recursion variables Y and Z in the process declaration below are of type I:

$$\begin{aligned} Y &= a \cdot Z \cdot Y \\ Z &= b \cdot Z + b \end{aligned}$$

Y repeatedly performs an a followed by one or more b's. The regular method linearises this process declaration as follows.

Replace $Z \cdot Y$ by a fresh recursion variable X.

$$\begin{aligned} Y &= a \cdot X \\ Z &= b \cdot Z + b \\ X &= Z \cdot Y \end{aligned}$$

Expand Z in the right-hand side of X. (Store that $X = Z \cdot Y$.)

$$\begin{aligned} Y &= a \cdot X \\ Z &= b \cdot Z + b \\ X &= b \cdot Z \cdot Y + b \cdot Y \end{aligned}$$

Replace $Z \cdot Y$ by X in the right-hand side of X.

$$\begin{aligned} Y &= a \cdot X \\ Z &= b \cdot Z + b \\ X &= b \cdot X + b \cdot Y \end{aligned}$$

We give a second example with the regular method, that involves data types.

Example 18.

$$X(n{:}Nat) = a(n) \cdot b(S(n)) \cdot c(S(S(n))) \cdot X(S(S(S(n))))$$

$$\begin{aligned} X(n{:}Nat) &= a(n) \cdot Y(n) \\ Y(n{:}Nat) &= b(S(n)) \cdot c(S(S(n))) \cdot X(S(S(S(n)))) \end{aligned}$$

$$\begin{aligned} X(n{:}Nat) &= a(n) \cdot Y(n) \\ Y(n{:}Nat) &= b(S(n)) \cdot Z(n) \\ Z(n{:}Nat) &= c(S(S(n))) \cdot X(S(S(S(n)))) \end{aligned}$$

The following example shows that the regular method does not always terminate. It takes as starting point the process declaration from Example 16.

Example 19.

$$
\begin{array}{l}
Y = a{\cdot}Y{\cdot}b + c \\
\hline
Y = a{\cdot}X_1 + c \\
X_1 = Y{\cdot}b \\
\hline
Y = a{\cdot}X_1 + c \\
X_1 = a{\cdot}X_1{\cdot}b + c{\cdot}b \\
\hline
Y = a{\cdot}X_1 + c \\
X_1 = a{\cdot}X_2 + c{\cdot}Z_1 \\
X_2 = X_1{\cdot}b \\
Z_1 = b \\
\hline
Y = a{\cdot}X_1 + c \\
X_1 = a{\cdot}X_2 + c{\cdot}Z_1 \\
X_2 = a{\cdot}X_2{\cdot}b + c{\cdot}Z_1{\cdot}b \\
Z_1 = b \\
\hline
\vdots
\end{array}
$$

We continue to show how recursive equations containing type II recursion variables can be linearised. Since we consider parallel pCRL, type II recursion variables can by definition be eliminated from right-hand sides of recursive equations. Next, recursive equations of type I recursion variables can be linearised using one of the previously described methods. It remains to linearise right-hand sides of recursive equations of type II recursion variables, containing parallelism, encapsulation, hiding and renaming. This can be done in a straightforward fashion. We give an example that features the merge of two LPEs.

Example 20. Let $a \,|\, b = c$, and consider the process declaration

$$
\begin{aligned}
X(n{:}Nat) \quad &= a(n){\cdot}X(S(n)) \lhd n < 10 \rhd \delta \\
&\quad + b(n){\cdot}X(S(S(n))) \lhd n > 5 \rhd \delta \\
Y(m{:}Nat, n{:}Nat) &= X(m) \parallel X(n)
\end{aligned}
$$

The recursive equation of the type II recursion variable Y can be linearised as follows:

$$
\begin{aligned}
Y(m{:}Nat, n{:}Nat) = {} & a(m){\cdot}Y(S(m), n) \lhd m < 10 \rhd \delta \\
& + a(n){\cdot}Y(m, S(n)) \lhd n < 10 \rhd \delta \\
& + b(m){\cdot}Y(S(S(m)), n) \lhd m > 5 \rhd \delta \\
& + b(n){\cdot}Y(m, S(S(n))) \lhd n > 5 \rhd \delta \\
& + c(m){\cdot}Y(S(m), S(S(n))) \lhd m < 10 \wedge n > 5 \wedge eq(m, n) \rhd \delta \\
& + c(n){\cdot}Y(S(S(m)), S(n)) \lhd m > 5 \wedge n < 10 \wedge eq(m, n) \rhd \delta
\end{aligned}
$$

6.3 State Space Generation and Storage

From an LPE $X(d{:}D)$ and initial state $d_0 \in D$, the state space can be generated with the μCRL toolset using the `instantiator` command (see Appendix A). The state space generation algorithm in Table 6.1 focuses on finding reachable states, i.e., transitions are ignored. When a state is discovered, storing its outgoing transitions is a trivial matter. In the algorithm, the set *Closed* contains the generated states of which the outgoing transitions have already been explored, while the set *Open* contains the generated states of which the outgoing transitions still need to be explored.

Table 6.1. State space generation algorithm

Initially, $Open = \{d_0\}$ and $Closed = \emptyset$.

while $Open \neq \emptyset$ **do**

select $d \in Open$; $Open := Open \backslash \{d\}$; $Closed := Closed \cup \{d\}$;

from LPE X, compute each transition $d \xrightarrow{a} d'$;

if $d' \notin Open \cup Closed$ **then** $Open := Open \cup \{d'\}$

The state space generation algorithm itself is straightforward. However, there are two bottlenecks in its execution. (1) State spaces tend to become very large, typically billions of states, and these somehow have to be stored in memory. (2) Since state spaces are so large, the check whether $d' \notin Open \cup Closed$ can become very expensive.

Hashing (see, e.g., [38]) is a good method for storing state spaces. Hashing is generally used when the number of slots in a table is relatively large compared with the number of items that are stored in the table. A *hash function* h maps data elements to a relatively small set of indices, which guarantees a better use of memory, and speeds up the average time to search for an element

in the table. If two data elements d and d' in the table happen to have the same hash value, meaning that $h(d) = h(d')$, then the two items are kept as a *linked list* at position $h(d)$ in the resulting hash table. A hash table is depicted in Fig. 6.1.

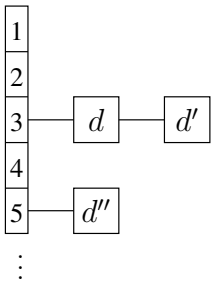


Fig. 6.1. Example of a hash table

When the hash table gets full, blocks of states from the hash table can be swapped to *disk* (e.g., based on 'age'). A disk lookup is much more expensive than an operation on the hash table, but the hope is that only relatively few disk lookups are needed.

When a generated state d' is not in the hash table, in principle the check $d' \notin \mathit{Open} \cup \mathit{Closed}$ in the state space generation algorithm requires an expensive disk lookup, to see whether maybe d' was swapped to disk. A *Bloom filter* [24] allows a relatively inexpensive check whether $d' \notin \mathit{Open} \cup \mathit{Closed}$, which however can give rise to false positives (i.e., a result $d' \notin \mathit{Open} \cup \mathit{Closed}$ is always correct, but a result $d' \in \mathit{Open} \cup \mathit{Closed}$ may be incorrect).

A Bloom filter has as parameters some natural numbers k and m, which must be chosen smartly to guarantee that (1) the Bloom filter is a relatively inexpensive data structure (compared to the disk space), and (2) the number of false positives is minimised. Fix some random hash functions $h_1, \ldots, h_k : D \to \{1, \ldots, m\}$. A Bloom filter is a bit array of length m, with initially all bits set to 0. For each generated state d, the bits in the Bloom filter at positions $h_1(d), \ldots, h_k(d)$ are set to 1. If a state d' is generated, and does not occur at entry $h(d')$ in the hash table, then it is checked whether positions $h_i(d')$ for $i = 1, \ldots, k$ in the Bloom filter all contain 1. If not, then $d' \notin \mathit{Open} \cup \mathit{Closed}$. Else, still an expensive disk lookup is required, since the result $d' \in \mathit{Open} \cup \mathit{Closed}$ might be a false positive (see Exercise 63).

An important question is what are optimal values for k and m. Of course this depends on the size of the state space that is being generated. When n elements have been inserted in $\mathit{Open} \cup \mathit{Closed}$, kn times a bit in the Bloom filter has been set to 1. So the probability that, after the n insertions, a certain position in the Bloom filter still contains 0 is

$$\left(\frac{m-1}{m}\right)^{kn}$$

So the probability that k positions in the Bloom filter all contain 1 is

$$\left(1-\left(\frac{m-1}{m}\right)^{kn}\right)^{k}$$

For given m and n, the number of false positives are minimal for k approximately $0.7 \cdot \frac{m}{n}$. In a typical case study, $k = 4$ and 256 MB is given to the Bloom filter; see [66].

The motivation for *bitstate hashing* [71] is that, due to the state explosion problem, often in the end the size of a generated state space turns out to be much too large to be stored in memory. In bitstate hashing, unlike standard hash tables, no linked lists are maintained; that is, if two generated states happen to have the same hash value, then only one of them may be explored. The downside of this approach is that it may give rise to only a partial search of the state space. The upside is that no extra disk space is needed. In practice, bitstate hashing has shown to be a healthy pragmatic approach to meet the state explosion problem. On the one hand, if the number of states in a generated state space is considerably smaller than the number of slots in the hash table, then only few states will be overwritten. On the other hand, if the number of states in a generated state space is considerably larger than the number of slots in the hash table, then ultimately only few slots in the hash table will be empty. Bitstate hashing therefore approximates an exhaustive search for small protocols, and slowly changes into a controlled partial search for large protocols. Bitstate hashing is prominent in the model checker SPIN [70], and is available for the μCRL toolset via a connection to OPEN/CÆSAR [49].

6.4 Minimisation Modulo Branching Bisimilarity

We sketch an algorithm by Groote and Vaandrager [64] to decide which states in a finite state space are branching bisimilar (see Definition 2). The basic idea of this algorithm is to partition the set of states in the state space under consideration into subsets of states that may be branching bisimilar; if two states are in distinct subsets, then it is guaranteed that they are not branching bisimilar. Initially, all states are in the same set. In each processing step, one of the sets in the partition is divided into two disjoint subsets. This is repeated until none of the sets in the partition can be divided any further.

We take as input a finite state space. In order to explain the minimisation algorithm, we introduce some notation. If P and P' are two sets of states and $a \in \mathsf{Act} \cup \{\tau\}$, then $s_0 \in \mathit{split}_a(P, P')$ if there exists an execution sequence

$s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_{n-1} \xrightarrow{a} s_n$ for some $n > 0$ such that $s_i \in P$ for $i = 0, \ldots, n-1$ and $s_n \in P'$. Note that $split_a(P, P') \subseteq P$.

The crux of the minimisation algorithm is that, since it is guaranteed that branching bisimilar states reside in the same set of the partition, a state $s_0 \in split_a(P, P')$ and a state $\hat{s}_0 \in P \backslash split_a(P, P')$ cannot be branching bisimilar if $a \neq \tau$ or $P \neq P'$. Namely, $s_0 \in split_a(P, P')$ implies that there exists an execution sequence $s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_{n-1} \xrightarrow{a} s_n$ with $s_i \in P$ for $i = 0, \ldots, n-1$ and $s_n \in P'$. Since $\hat{s}_0 \notin split_a(P, P')$, this execution sequence cannot be mimicked (modulo branching bisimilarity) by $\hat{s}_0$.

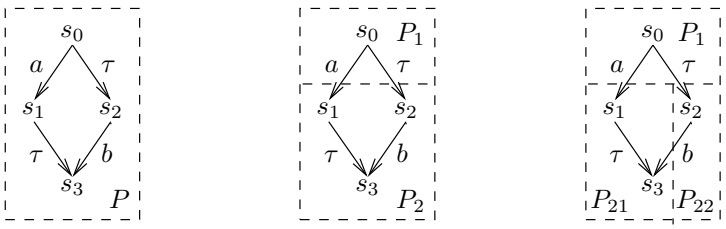
The minimisation algorithm works as follows. Suppose that at some point we have constructed a partition $P_1, \ldots, P_k$ of disjoint sets of states, where $P_1 \cup \cdots \cup P_k$ is the set of states in the state space under consideration. (Remember that initially this partition consists of a single set.) Suppose that for some $i, j \in \{1, \ldots, k\}$ and $b \in \mathsf{Act} \cup \{\tau\}$ with $i \neq j$ or $b \neq \tau$ we have $\emptyset \subset split_b(P_i, P_j) \subset P_i$ (where $\subset$ denotes strict set inclusion). Then P_i can be replaced by the two disjoint sets $split_b(P_i, P_j)$ and $P_i \backslash split_b(P_i, P_j)$, which by the previous condition are guaranteed to be strict subsets of P_i. This procedure is repeated until no set in the partition can be split in this fashion any further.

Groote and Vaandrager proved that if this procedure outputs the partition $Q_1, \ldots, Q_\ell$, then two states are in the same set Q_i if and only if they are branching bisimilar in the state space. Thus, the states in each set Q_i can be collapsed, and τ-transitions within such a set can be eliminated, producing the desired minimised version of the state space modulo branching bisimilarity.

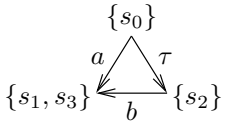
Example 21. We show how the minimisation algorithm minimises the state space that belongs to the process term $(a \cdot \tau + \tau \cdot b) \cdot \delta$. Initially, the set P contains all four states in the corresponding state space.

$split_a(P, P)$ consists of the initial state only, so that the minimisation algorithm separates the initial state from the other states.

Next, $split_b(P_2, P_2)$ only contains the state belonging to the subterm $b \cdot \delta$, so that the minimisation algorithm separates this state from the other two states in P_2.



Finally, none of the sets P_1, P_{21} and P_{22} can be split any further, so that we obtain the following minimised state space:

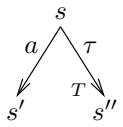


Groote and Vaandrager showed that their algorithm can be performed in worst-case time complexity $O(mn)$, where n is the number of states and m the number of transitions in the state space under consideration. The crux of the minimisation algorithm is an ingenious method to decide, for a given partition $P_1, \ldots, P_k$, whether there exist i, j and b such that $\emptyset \subset \mathit{split}_b(P_i, P_j) \subset P_i$; this method, which is omitted here, requires $O(m+n)$. Since there can be no more than $n-1$ subsequent splits of sets in the partition, the worst-case time complexity is $O(mn)$.

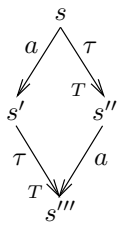
6.5 Confluence

In distributed systems, two different concurrent components can in general perform transitions that are not causally related, meaning that executing these two transitions consecutively leads to the same state, irrespective of the fact which of the two transitions is executed first. This phenomenon is referred to as *confluence* (cf. Section 2.2). Confluence can be exploited when analysing processes that involve the hidden action τ. Roughly, a τ-transition $s \xrightarrow{\tau} s'$ is confluent if it commutes with any other transition from s. Different notions of confluence, and of detecting confluent τ-transitions, were proposed in [23, 56, 62, 86, 107].

Here we use the notion of confluence originating from [56]. Assume a state space, together with a set T of τ-transitions in this state space; let $s \xrightarrow{\tau}_T s'$ denote that $s \xrightarrow{\tau} s' \in T$. We say that T is *confluent* if for each pair of different transitions $s \xrightarrow{a} s'$ (with $a \in \mathsf{Act} \cup \{\tau\}$) and $s \xrightarrow{\tau}_T s''$, the picture



can be completed in the following fashion:



Definition 5 (Confluence). *Assume a state space G. A collection T of τ-transitions in G is* confluent *if for all transitions $s \xrightarrow{a} s' \in G$ and $s \xrightarrow{\tau} s'' \in T$:*

(1) either $s' \xrightarrow{\tau} s''' \in T$ and $s'' \xrightarrow{a} s''' \in G$, for some state s''';
(2) or $a = \tau$ and $s' = s''$.

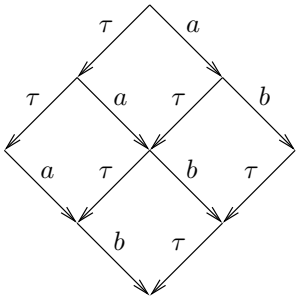
The union of confluent sets of τ-transitions is again confluent, so for each state space there is a maximal confluent set of τ-transitions.

A confluent set T of τ-transitions can be used to trim the corresponding state space G. Namely, if $s \xrightarrow{\tau} s' \in T$, then s and s' are branching bisimilar states in G, so that they can be identified. Even more so, if G does not contain τ-loops, then $s \xrightarrow{\tau} s'$ can be given priority over all other outgoing transitions of s (meaning that all transitions of the form $s \xrightarrow{a} s''$, except $s \xrightarrow{\tau} s'$ itself, are eliminated from G) without influencing the branching bisimulation class of s. Priorisation with respect to confluent transitions $s \xrightarrow{\tau} s'$ and collapsing the states s and s' may lead to a substantial reduction of G. The next example shows that the absence of τ-loops in G is essential for the soundness, modulo branching bisimilarity, of prioritisation of confluent τ-transitions.

Example 22. Consider the finite state space defined by the process declaration $X = (\tau + a)\cdot X$; it contains a τ-loop, so it is not convergent. The τ-transition in this state space is confluent. If the a-transition is eliminated from this state space, then the resulting state space is defined by the recursive equation $Y = \tau\cdot Y$. Clearly, X and Y are not branching bisimilar.

We give an example of the use of confluence for state space reduction.

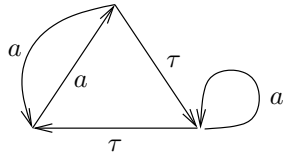
Example 23. Consider the state space



The maximal confluent set of τ-transitions contains all six τ-transitions. Priorisation of confluent τ's produces the state space belonging to $\tau\cdot\tau\cdot a\cdot b\cdot\delta$.

The next example shows that the maximal confluent set of τ-transitions may be a proper subset of the collection of inert τ-transitions of a state space.

Example 24. Consider the state space



The maximal confluent set of τ-transitions is empty. However, the two τ-transitions are both inert.

Groote and van de Pol [56] presented an efficient algorithm to compute, for a given finite state space, the maximal confluent set T of τ-transitions. As a preprocessing step, first all states that are on a τ-loop are collapsed to a single state. That is, if there are execution sequences $s \xrightarrow{\tau} \cdots \xrightarrow{\tau} s'$ and $s' \xrightarrow{\tau} \cdots \xrightarrow{\tau} s$, then s and s' are identified. If two states are on a τ-loop, then they are branching bisimilar (see Exercise 40). The τ-loops in a state space can for instance be detected using Kosaraju's algorithm (see, e.g., [38]) for finding the *strongly connected components* in a directed graph. Nodes i and j are in the same strongly connected component of a directed graph if there exists a path from i to j and vice versa.

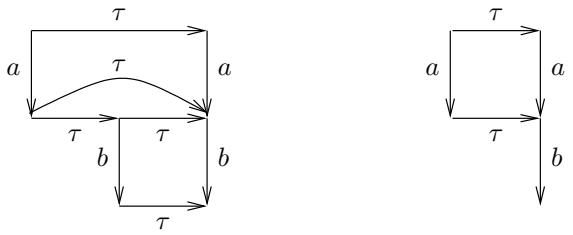
In the algorithm from [56], initially all τ-transitions of the state space are in T, and subsequently τ-transitions are eliminated from T if they are found to obstruct confluence. That is, all transitions of the state space are placed on a stack. In each processing step, a transition $s \xrightarrow{a} s'$ is taken from the stack, and for each $s \xrightarrow{\tau} s'' \in T$ it is verified whether either property (1) or (2) in Definition 5 holds. If this is not the case, then $s \xrightarrow{\tau} s''$ is eliminated from T, and all transitions $s''' \xrightarrow{b} s$ that were previously eliminated from the stack are placed back on the stack. Namely, $s''' \xrightarrow{b} s$ and some transition $s''' \xrightarrow{\tau} s'''' \in T$ may previously have been found not to obstruct confluence due to the fact that $s \xrightarrow{\tau} s''$ was erroneously present in T. This procedure is repeated until the stack is empty, after which the constructed set T is delivered as output. This is guaranteed to be the maximal confluent set of τ-transitions.

Assume a finite state space with m transitions. The algorithm to detect τ-loops (or better, to find strongly connected components) has worst-case time complexity $O(m+n)$. Groote and van de Pol [56] showed that the worst-case time complexity of their algorithm to compute the maximal confluent set of τ-transitions is also $O(m+n)$ (under the assumption that there is a fixed bound on the number of outgoing τ-transitions from each state). Thus their algorithm performs better than the minimisation algorithm modulo branching bisimilarity; see Section 6.4. Moreover, Groote and van de Pol showed by means of a number of benchmarks that in practice confluence can be considerably more efficient than minimisation, especially if the state space under consideration is large and the number of τ-transitions is relatively low. Hence, computing the maximal confluent set of τ-transitions can be a sensible preprocessing step before applying minimisation modulo branching bisimilarity.

Computation of the maximal confluent set of τ-transitions is supported by the μCRL toolset.

After compression of a state space on the basis of its maximal confluent set of τ-transitions, the resulting state space may again contain confluent τ-transitions. Hence, it makes sense to iterate the algorithm of Groote and van de Pol until the maximal confluent set of τ-transitions in the resulting state space has become empty. We present an example of this phenomenon.

Example 25. Below are depicted a state space before and after compression with respect to the confluent τ-transitions, respectively:



Compression with respect to the confluent τ-transitions in the latter state space produces the state space belonging to $a{\cdot}b{\cdot}\delta$.

6.6 Model Checking

Model checking is an automated method for verifying properties of states in a finite state space. These properties are expressed as temporal logic formulas. Efficient algorithms exist to traverse the state space and check in which states a certain temporal logic formula is satisfied. We focus on two temporal logics: computation tree logic and the μ-calculus.

Computation Tree Logic

CTL [34] is a temporal logic to express properties of state spaces that do not carry actions on their transitions. *Action-based* CTL [42], written ACTL, is an extension of CTL to state spaces with actions. ACTL consists of formulas on states, defined by the following BNF grammar:

$$\phi \quad ::= \quad \mathtt{T} \mid \neg\phi \mid \phi \wedge \phi' \mid \langle a \rangle \phi \mid \mathtt{E}\,\phi\,\mathtt{U}\,\phi' \mid \mathtt{EG}\,\phi$$

where a ranges over $\mathsf{Act} \cup \{\tau\}$. Here, $\mathtt{T}$ is the universal predicate that holds in all states. (Actually, CTL assumes a collection of predicates, each of which holds in part of the states.) As usual, $\neg$ denotes negation and $\wedge$ conjunction. All other operators from boolean logic, such as disjunction and implication, can be expressed by means of negation and conjunction. The '$\mathtt{E}$' in the last two operators stands for existential quantification. The intuitions behind the three temporal constructs are as follows.

- $\langle a \rangle \phi$ holds in a state s if there is a transition $s \xrightarrow{a} s'$ where formula ϕ holds in state s'.
- $\mathtt{E}\,\phi\,\mathtt{U}\,\phi'$ holds in a state s if there is an execution sequence, starting in s, that only visits states in which ϕ holds, until it visits a state in which ϕ' holds.
- $\mathtt{EG}\,\phi$ holds in a state s if there is an execution sequence, starting in s, which cannot be extended to a longer execution sequence, such that it only visits states in which ϕ holds.

These intuitions can be formalised as follows. Assume a state space. A *full path* is either an infinite execution sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$, or a finite execution sequence $s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{\ell-1}} s_\ell$ where there is no transition $s_\ell \xrightarrow{b} s$ (i.e., s_ℓ is a deadlock state). The states s_0 that satisfy an ACTL formula ϕ, denoted by $s_0 \models \phi$, are defined inductively as follows:

$$
\begin{array}{ll}
s_0 \models \mathtt{T} & \\
s_0 \models \neg\phi & \text{if } s_0 \not\models \phi \\
s_0 \models \phi \wedge \phi' & \text{if } s_0 \models \phi \text{ and } s_0 \models \phi' \\
s_0 \models \langle a \rangle \phi & \text{if there is a state } s_1 \text{ with } s_0 \xrightarrow{a} s_1 \text{ and } s_1 \models \phi \\
s_0 \models \mathtt{E}\,\phi\,\mathtt{U}\,\phi' & \text{if there is an execution sequence } s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{\ell-1}} s_\ell \\
 & \text{with } s_k \models \phi \text{ for } k \in \{0, \ldots, \ell-1\} \text{ and } s_\ell \models \phi' \\
s_0 \models \mathtt{EG}\,\phi & \text{if there is a full path, starting in } s_0, \text{ such that } s \models \phi \\
 & \text{for all states } s \text{ on this full path}
\end{array}
$$

ACTL contains some more temporal operators, which can all be expressed in terms of the aforementioned constructs. In the operators defined below, '$\mathtt{A}$' refers to universal quantification.

- $[a]\,\phi$ holds in a state s if for each transition $s \xrightarrow{a} s'$, formula ϕ holds in state s'. It is equivalent to $\neg\,\langle a \rangle\,\neg\phi$.
- $\mathtt{A}\,\phi\,\mathtt{U}\,\phi'$ holds in a state s if each execution sequence starting in s only visits states in which ϕ holds, until it visits a state in which ϕ' holds. It is equivalent to $(\neg\,\mathtt{E}\,\neg\phi'\,\mathtt{U}\,(\neg\phi \wedge \neg\phi')) \wedge (\neg\,\mathtt{EG}\,\neg\phi')$.
- $\mathtt{AG}\,\phi$ holds in a state s if each execution sequence, starting in s only visits states in which ϕ holds. It is equivalent to $\neg\,\mathtt{E}\,\mathtt{T}\,\mathtt{U}\,\neg\phi$.

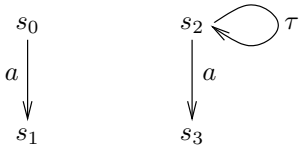
Assume a finite state space. The algorithmic verification to determine in which states a given temporal logic formula holds is known as *model checking*. We present an efficient model checking algorithm for CTL formulas from [35], which extends to ACTL without any complications [42]. The algorithm works by induction on the structure of the given formula, so one can assume that it is known for each proper subformula of the given ACTL formula in which states of the state space it is satisfied.

- The three boolean operators are straightforward: $\mathtt{T}$ holds in all states, $\neg\phi$ holds in a state if and only if ϕ does not hold in this state, and $\phi \wedge \phi'$ holds in a state if and only if both ϕ and ϕ' hold in this state.
- $\langle a \rangle \phi$ holds in each state that can perform an a-transition resulting in a state where ϕ holds.
- To compute the states in which $\mathtt{E}\,\phi\,\mathtt{U}\,\phi'$ holds, we start with the states where ϕ' holds, and then work backwards, using the reverse of the transition relation, visiting only states where ϕ holds.
- To compute the states in which $\mathtt{EG}\,\phi$ holds, first we eliminate all states from the state space where ϕ does not hold. A state satisfies $\mathtt{EG}\,\phi$ if and only if in the resulting state space there is an execution sequence either to a state that is a deadlock state in the original state space, or to a strongly connected component in the resulting state space that includes at least one transition.

This algorithm has worst-case time complexity $O(|\phi| \cdot m)$, where $|\phi|$ is the size of the given ACTL formula and m is the number of transitions of the original state space.

Fairness

Branching bisimilarity satisfies a notion of *fairness*; see Section 4.4. That is, if an exit from a τ-loop exists, then no infinite execution sequence will remain in this τ-loop forever. In the setting of model checking it is often also desirable to have a notion of fairness. For example, consider the following two state spaces.



Due to the τ-loop, state s_2 satisfies $\mathtt{EG}\,\langle a \rangle \mathtt{T}$. However, according to the fairness paradigm, each full path will at some point execute the exit action a and end up in state state s_3, where $\langle a \rangle \mathtt{T}$ is not satisfied. In general, fairness dictates that each full path ends up in a strongly connected component from which no escape is possible. Under the assumption of fairness, the semantics of $\langle a \rangle \phi$ and $\mathtt{E}\,\phi\,\mathtt{U}\,\phi'$ remains unchanged, because these formulas deal with the existence of some finite execution sequence. However, the semantics of $\mathtt{EG}\,\phi$ needs to be adapted. Let us say that a full path is *fair* if either it is finite, or it ends up in a strongly connected component without exits, and all states in this strongly connected component are visited infinitely often. The interpretation of $\mathtt{EG}\,\phi$ becomes:

- $\texttt{EG}\,\phi$ holds in a state s if there is a *fair* full path, starting in s, that only visits states in which ϕ holds.

The model checking algorithm for CTL formulas is then adapted as follows.

- To compute the states in which $\texttt{EG}\,\phi$ holds, first we eliminate all states from the state space where ϕ does not hold. A state satisfies $\texttt{EG}\,\phi$ if and only if there is an execution sequence in the resulting state space to a strongly connected component from which no escape is possible in the original state space.

Linear Temporal Logic

LTL [93] is a temporal logic to express properties of state spaces that do not carry actions on their transitions. Although the class of LTL formulas has a large overlap with the class of CTL formulas, the model checking algorithm for LTL is very different from that of CTL. The model checking algorithm for LTL [75] is linear in the number of transitions, but exponential in the size of the formula. From a practical point of view this exponential complexity is not so problematic, because in general the size of a formula is small with respect to the size of the state space against which it is checked. CTL is usually referred to as a *branching-time* temporal logic, because it quantifies over the paths that are possible from a state. LTL is referred to as a *linear-time* temporal logic, because formulas are interpreted over linear sequences of states. See [45] for a comparison of branching-time and linear-time temporal logics.

μ-Calculus

The *μ-calculus* [74] is based on fixpoint computations [103]. Let D be a finite set with a partial ordering $\leq$ (meaning that this binary relation is reflexive, anti-symmetric and transitive). Given a mapping $\mathcal{S} : D \to D$, an element $d \in D$ is a *fixpoint* of $\mathcal{S}$ if $\mathcal{S}(d) = d$. Moreover, d is a *least fixpoint* or *greatest fixpoint* if $d \leq e$ or $e \leq d$, respectively, for all fixpoints e of $\mathcal{S}$. The least and the greatest fixpoint of $\mathcal{S}$ (if they exist) are denoted by $\mu X.\mathcal{S}(X)$ and $\nu X.\mathcal{S}(X)$, respectively.

A mapping $\mathcal{S} : D \to D$ is called *monotonic* if $d \leq e$ implies $\mathcal{S}(d) \leq \mathcal{S}(e)$. Let $\mathcal{S} : D \to D$ be monotonic, and suppose that D has a least element d_0 and a greatest element e_0 (i.e., $d_0 \leq d$ and $d \leq e_0$ for all $d \in D$). Then $\mathcal{S}$ has a least and a greatest fixpoint, which can be computed in an iterative way. Put $d_{i+1} = \mathcal{S}(d_i)$ and $e_{i+1} = \mathcal{S}(e_i)$ for $i \geq 0$. Since $d_0 \leq d_1$ and $e_1 \leq e_0$, by the monotonicity of $\mathcal{S}$ it follows that $d_i \leq d_{i+1}$ and $e_{i+1} \leq e_i$ for all $i \geq 0$. As D is finite, $d_j = d_{j+1}$ and $e_{k+1} = e_k$ for some $j, k \geq 0$. It is not hard to see that $\mu X.\mathcal{S}(X) = d_j$ and $\nu X.\mathcal{S}(X) = e_k$, respectively.

We proceed to define the μ-calculus. Its temporal logic formulas, which express properties of states, are defined by the following BNF grammar:

$$\phi \quad ::= \quad \mathtt{T} \mid \mathtt{F} \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \langle a \rangle \phi \mid [a]\,\phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

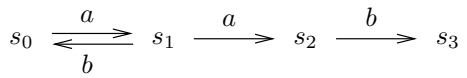
where a ranges over $\mathsf{Act} \cup \{\tau\}$ and X ranges over some collection of recursion variables. We restrict to *closed* μ-calculus formulas, meaning that each occurrence of a recursion variable X is within the scope of a minimal fixpoint μX or a maximal fixpoint νX. For example, $\mu X.X$ is closed, while $\mu X.Y$ is not; the recursion variable Y in the latter formula is not within the scope of a minimal or maximal fixpoint.

Assume a finite state space. The formulas $\mu X.\phi$ and $\nu X.\phi$ represent minimal and maximal fixpoints, respectively. Here, the formula ϕ represents a mapping from sets of states to sets of states: a set S of states is mapped to those states where ϕ holds, under the assumption that the recursion variable X evaluates to $\mathtt{T}$ for states in S, and to $\mathtt{F}$ for states outside S. The image of S, under the mapping ϕ, is denoted by $FIX(\phi, X{=}S)$. As partial ordering on sets of states we take set inclusion. So the least and the greatest element are the empty set and the set of all states, respectively.

In order to guarantee the existence of a minimal and a maximal fixpoint, the mappings represented by μ-calculus formulas ϕ must be monotonic. It is not hard to see that the operations conjunction, disjunction, $\langle a \rangle \phi$ and $[a]\,\phi$ are indeed monotonic; for instance, $\phi \Rightarrow \phi'$ yields $\langle a \rangle \phi \Rightarrow \langle a \rangle \phi'$, and similar facts hold for the other operations. So $\mu X.\phi$ and $\nu X.\phi$ are well-defined. By definition, the formulas $\mu X.\phi$ and $\nu X.\phi$ are satisfied only by the states in the minimal and maximal fixpoint of the mapping ϕ, respectively.

$\mu X.\phi$ and $\nu X.\phi$ can be computed as explained before. For the case $\mu X.\phi$, initially we take as solution S_0 for X the empty set of states. Next, we repeatedly compute $S_{i+1} = FIX(\phi, X{=}S_i)$ for $i \geq 0$, meaning the set of states that satisfy ϕ, under the assumption that S_i is the set of states where the subformula X is satisfied. By monotonicity, $S_i \subseteq S_{i+1}$. Since there are only finitely many states, $S_i = S_{i+1}$ for some i. This fixpoint is the actual solution for X in $\mu X.\phi$. For the case $\nu X.\phi$, initially we take as solution S_0 for X the set of all states. Again, we repeatedly compute $S_{i+1} = FIX(\phi, X{=}S_i)$ for $i \geq 0$. By monotonicity, $S_i \supseteq S_{i+1}$. Since there are only finitely many states, $S_i = S_{i+1}$ for some i. This fixpoint is the actual solution for X in $\nu X.\phi$.

Example 26. Consider the state space



First, we compute the solution for X in the formula

$$\nu X.(\langle a \rangle X \vee \langle b \rangle X)$$

Initially, the maximal fixpoint X contains all states. In the first iteration, s_3 is the only state from which there is no a- or b-transition to a state in X, so the next value for X is $\{s_0, s_1, s_2\}$. In the second iteration, s_2 is the only state

from which there is no a- or b-transition to a state in X, so the next value for X is $\{s_0, s_1\}$. Since $s_0 \xrightarrow{a} s_1$ and $s_1 \xrightarrow{b} s_0$, $\{s_0, s_1\}$ is the fixpoint solution for X. So the formula is satisfied in s_0 and s_1, and not satisfied in s_2 and s_3.

Next, we compute the solution for Y in the formula

$$\mu Y.(\langle a\rangle Y \vee \langle b\rangle Y)$$

Initially, the minimal fixpoint Y is empty. Since clearly no state can perform an a- or b-transition to a state in the empty set, $\emptyset$ is the fixpoint solution for Y. So the formula is satisfied in none of the states.

Note that, in contrast to ACTL, negation $\neg$ is absent from the μ-calculus. This is because negation does not induce a monotonic mapping on sets of states.

Example 27. Let us try to compute which states in the state space below satisfy the formula $\mu X.\neg\langle a\rangle X$.

$$s_0 \underset{a}{\overset{a}{\rightleftarrows}} s_1$$

Since μX is a minimal fixpoint, initially X is taken to be $\emptyset$. From s_0 and s_1, no a-transition is possible to a state in $X = \emptyset$, so s_0 and s_1 both satisfy $\neg\langle a\rangle X$. This means that in the first iteration, X is taken to be $\{s_0, s_1\}$. Now both from s_0 and s_1, an a-transition is possible to a state in $X = \{s_0, s_1\}$, so neither s_0 nor s_1 satisfies $\neg\langle a\rangle X$. This means that in the second iteration, X is taken to be $\emptyset$, etc. This computation does not reach a fixpoint.

Consider a formula $\mu X.\phi$. By monotonicity, computing $FIX(\phi, X{=}\emptyset)$ takes no more than n iterations, where n is the total number of states. However, in the case of nested fixpoints, for example when ϕ is of the form $\nu Y.\phi'$ where X occurs in ϕ', an adaptation of the solution for X may lead to an adaptation of the solution for Y. Recomputing the value Y again takes up to n iterations, etc. Hence, the worst-case time complexity of the model checking algorithm for the μ-calculus described above is $O(|\phi|{\cdot}m{\cdot}n^{N(\phi)})$, where $N(\phi)$ is the length of the longest chain of nested fixpoints in ϕ.

Example 28. Let the state space consist of $2n+1$ states $\{s_0, \ldots, s_{2n}\}$, and of transitions $s_{2i} \xrightarrow{a} s_{2i+1}$ and $s_{2i+1} \xrightarrow{b} s_{2i+2}$ for $i = 0, \ldots, n-1$. We compute the solutions for X and Y in the formula

$$\nu X.\nu Y.(\langle a\rangle X \vee \langle b\rangle Y)$$

Initially, the maximal fixpoints X and Y contain all states. The subsequent intermediate solutions for X and Y are:

Y	X
$\{s_0, \ldots, s_{2n}\}$	$\{s_0, \ldots, s_{2n}\}$
$\{s_0, \ldots, s_{2n-1}\}$	$\{s_0, \ldots, s_{2n-2}\}$
$\{s_0, \ldots, s_{2n-3}\}$	$\{s_0, \ldots, s_{2n-4}\}$
$\vdots$	$\vdots$
$\emptyset$	$\emptyset$

The empty set is the actual solution for both X and Y.

It is not a coincidence that in Example 28, X and Y have the same solution. One can prove that if two fixpoints are direct neighbours in a formula, then their solutions always coincide.

If in a chain of nested fixpoints there are two subsequent minimal fixpoints, say μX and μY, then after each adaptation of the solution for X, the newly computed solution for Y is guaranteed to include the previous solution for Y [44]. Likewise, if in a chain of nested fixpoints there are two subsequent maximal fixpoints, say νX and νY, then after each adaptation of the solution for X, the new solution for Y is guaranteed to be included in the previous solution for Y (cf. Example 28). Hence, the worst-case time-complexity for model checking μ-calculus formulas reduces to $O(|\phi| \cdot m \cdot n^{A(\phi)})$, where $A(\phi)$ is the length of the longest chain of nested *alternating* fixpoints in ϕ.

The following example shows that if in a chain of nested fixpoints a maximal fixpoint νX is followed by a minimal fixpoint μY (or vice versa), then after an adaptation of the solution for X, the new solution for Y is not guaranteed to include the previous solution for Y.

Example 29. Consider the state space

$$s_0 \underset{a}{\overset{a}{\rightleftarrows}} s_1$$

We compute the solutions for X and Y in the formula

$$\nu X.\langle b \rangle \, (\mu Y.(\langle a \rangle \, X \vee \langle a \rangle \, Y))$$

Initially, the maximal fixpoint X contains all states, while the minimal fixpoint Y is empty. The subsequent intermediate solutions for X and Y are:

Y	X
$\emptyset$	$\{s_0, s_1\}$
$\{s_0, s_1\}$	$\emptyset$
$\emptyset$	$\emptyset$

The empty set is the actual solution for both X and Y. In the second iteration it is essential that recomputation of Y starts at $\emptyset$ (instead of the intermediate solution $\{s_0, s_1\}$).

Regular μ-Calculus

In the *regular μ-calculus* [83] one is allowed to use expressions $\langle\beta\rangle\,\phi$ and $[\beta]\,\phi$, where β is a so-called *regular expression*, representing a set of traces. The formula $\langle\beta\rangle\,\phi$ means that ϕ holds after some trace from β, and $[\beta]\,\phi$ that ϕ holds after all traces from β.

The regular expressions β are defined by the following BNF grammar:

$$\alpha ::= \mathtt{T} \mid a \mid \neg\alpha \mid \alpha \wedge \alpha'$$

$$\beta ::= \alpha \mid \beta{\cdot}\beta' \mid \beta|\beta' \mid \beta^*$$

As before, a ranges over $\mathsf{Act} \cup \{\tau\}$. Expressions α represent a set of actions: $\mathtt{T}$ denotes the set of all actions, a the set $\{a\}$, $\neg\alpha$ the complement of α, and $\alpha \wedge \alpha'$ the intersection of α and α'. Regular expressions β represent a set of traces: $\beta{\cdot}\beta'$ denotes the traces that can be obtained by concatenating a trace from β and a trace from β', $\beta|\beta'$ the union of β and β', and β^* the transitive-reflexive closure of β (i.e., the traces that can be obtained by concatenating finitely many traces from β).

Regular μ-calculus formulas are transformed into μ-calculus formulas:

$$\langle\alpha\rangle\,\phi = \bigvee\nolimits_{a\in\alpha} \langle a\rangle\,\phi \qquad [\alpha]\,\phi = \bigwedge\nolimits_{a\in\alpha} [a]\,\phi$$

$$\langle\beta{\cdot}\beta'\rangle\,\phi = \langle\beta\rangle\,\langle\beta'\rangle\,\phi \qquad [\beta{\cdot}\beta']\,\phi = [\beta]\,[\beta']\,\phi$$

$$\langle\beta|\beta'\rangle\,\phi = \langle\beta\rangle\,\phi \vee \langle\beta'\rangle\,\phi \qquad [\beta|\beta']\,\phi = [\beta]\,\phi \wedge [\beta']\,\phi$$

$$\langle\beta^*\rangle\,\phi = \mu X.(\phi \vee \langle\beta\rangle\,X) \qquad [\beta^*]\,\phi = \nu X.(\phi \wedge [\beta]\,X)$$

Example 30. We give six examples of regular μ-calculus formulas.

$$[\mathtt{T}^*]\,\phi \qquad \phi \text{ holds always}$$

$\mathtt{T}^*$ contains all traces. So the formula above expresses that each execution sequence ends up in a state where ϕ holds.

$$[\mathtt{T}^*]\,\langle\mathtt{T}\rangle\,\mathtt{T} \qquad \text{deadlock-freeness}$$

The formula above expresses that each execution sequence ends up in a state where some transition can be performed.

$$[\mathtt{T}^*{\cdot}\mathit{error}]\,\mathtt{F} \qquad \text{no occurrence of } \mathit{error}$$

$\mathtt{T}^*{\cdot}\mathit{error}$ contains all traces that end with the action *error*. So the formula above expresses that all execution sequences that end with the action *error*, end up in a state where $\mathtt{F}$ holds. Since $\mathtt{F}$ does not hold in any state, this implies that no execution sequence performs the action *error*.

$$[\mathtt{T}^*]\,\mu X.[\tau]\,X \qquad \text{no reachable state exhibits an infinite } \tau\text{-sequence}$$

First we calculate the states where $\mu X.[\tau]\,X$ holds. As it is a minimal fixpoint, initially $X = \emptyset$. After the first iteration, X consists of all states that cannot perform a τ-transition. And after the nth iteration, X contains all states from which no sequence of n τ-transitions can be performed. So as a fixpoint, $\mu X.[\tau]\,X$ holds in those states that do not exhibit an infinite sequence of τ-transitions. Since $\mathtt{T}^*$ contains all traces, the formula above expresses that $\mu X.[\tau]\,X$ holds in all reachable states.

$[(\neg \mathit{send})^* \cdot \mathit{read}]\,\mathtt{F}$ each occurrence of *read* is preceded by a *send*

$(\neg \mathit{send})^*$ is the set of traces that do not contain an occurrence of the action *send*. So the formula above expresses that all execution sequences that do not contain the action *send* and that end with the action *read*, end up in a state where F holds. Since F does not hold in any state, this means that each occurrence of *read* in a trace must always be preceded by an occurrence of *send*.

$[\mathtt{T}^* \cdot \mathit{send}]\,\mu X.(\langle \mathtt{T} \rangle\,\mathtt{T} \wedge [\neg \mathit{read}]\,X)$ each *send* is eventually followed by a *read*

First we calculate the states where $\mu X.(\langle \mathtt{T} \rangle\,\mathtt{T} \wedge [\neg \mathit{read}]\,X)$ holds. As it is a minimal fixpoint, initially $X = \emptyset$. After the first iteration, X contains all states that can perform the action *read*, and from which no other action can be performed. And after the nth iteration, X contains all states from which every full path performs within n transitions the action *read*. So as a fixpoint, $\mu X.(\langle \mathtt{T} \rangle\,\mathtt{T} \wedge [\neg \mathit{read}]\,X)$ holds in those states from which every full path eventually performs the action *read*. Since $\mathtt{T}^* \cdot \mathit{send}$ contains all traces that end with the action *send*, the formula above expresses that each occurrence of *send* is eventually followed by an occurrence of *read*.

The CADP toolset, which serves as a back-end to the μCRL toolset, supports model checking of regular *alternation-free* μ-calculus formulas, meaning that it is not allowed to use a nesting of a minimal and a maximal fixpoint (see Appendix A for more information).

Note that regular μ-calculus formulas such as $\nu X.(\langle \beta^* \rangle\,X)$ and $\mu X.([\beta^*]\,X)$ are transformed to μ-calculus formulas that aren't alternation-free. Therefore such formulas are rejected by the CADP model checker.

To take values of variables into account in such a model checking verification, one can include artificial self-loops that carry these variables as action variables. For instance, to the process declaration of a recursion variable $X(d_1{:}D_1, d_2{:}D_2, d_3{:}D_3)$ one can add a summand

$$+\ \mathit{test}(d_1, d_3) \cdot X(d_1, d_2, d_3)$$

where *test* is a "fresh" action name.

6.7 Distributed State Space Generation

A fruitful approach to deal with state explosion is to generate a state space over a number of processors, so that the combined memory at these processors can be exploited. This requires a globally known hash function, so that states can be divided over processors on the basis of their hash values. When a state is generated at a processor, its hash value is calculated, and the state is forwarded to the appropriate processor. There it is determined whether the state was not generated before.

In μCRL, states are represented as a list of data terms, which are values of local variables at the parallel processes. Internally, these data terms are stored as ATerms, as explained at the end of Section 2.1 and in Example 14. ATerms are stored by means of maximal sharing, which is beneficial for memory usage and equality checking, but becomes a burden for computing hash values. Moreover, representations of states as a list of data terms tend to be very long, which makes the computation of hash values and sending states over the network needlessly expensive. Therefore, [20] introduced a database approach to distributed state space generation. It exploits the fact that in a transition between two states, usually most of the data parameters remain unchanged.

The idea is to maintain a central database in which the data terms occurring in states are provided with an index, which is a natural number. Every new data term encountered during state space generation gets as index $max + 1$, where max is the largest index in use so far. A state $(d_1, \ldots, d_k)$ can now be represented as a list of indices $(i_1, \ldots, i_k)$. The latter list is more compact, which makes the computation of hash values cheaper, and reduces bandwidth demands because lists of indices are sent between processors. Since the number of data terms used in states tends to be relatively small, compared to the total number of states in the state space, the database usually remains relatively small too.

To compute the successor states of a state $(i_1, \ldots, i_k)$, it first needs to be expanded into $(d_1, \ldots, d_k)$ again. Therefore the state space generator must continuously consult the database, to move back and forth between the two different representations of states. The state space generation algorithm from Section 6.3 needs to be adapted as follows. States are stored as lists of indices, so to compute the successor states of a state $(i_1, \ldots, i_k)$ in *Open*, first it must be expanded into its original form $(d_1, \ldots, d_k)$. For each successor state $(e_1, \ldots, e_k)$, the data terms in this state need to be resolved against the database, to obtain a list of indices $(j_1, \ldots, j_k)$; data terms that are not yet in the database are added to it. The hash value of $(j_1, \ldots, j_k)$ is computed, and $(j_1, \ldots, j_k)$ is sent to the responsible processor.

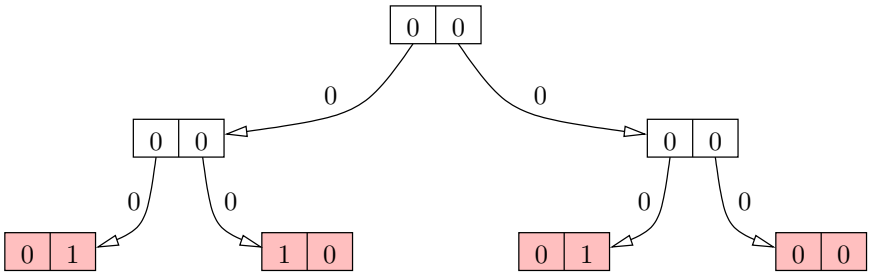
To reduce communication, the central database can be replicated at all processors. Each local database at a processor P always contains all indices from 0 up to some value max_P (which initially is -1). When P cannot find the indices of some data terms $d_{n_1}, \ldots, d_{n_m}$ in its local database, then it

sends these terms together with max_P to the central database. At the central database, first the data terms among $d_{n_1}, \ldots, d_{n_m}$ that are not yet present are included with fresh indices, and next the data terms with indices ranging from $max_P + 1$ up to max are sent back to P. Thus P is also provided with data terms that it has not encountered yet, to avoid future requests and thus reduce communication overhead.

To compress state representations further, lists of data-term indices are stored in a tree structure, exploiting the fact that in a transition often most data terms in the state representation remain unchanged, which means that lists of data-term indices representing states tend to contain the same indices at many places. For simplicity we assume that lists of data-term indices have length 2^k, for some $k > 1$. The idea is that each list of data-term indices is split into two halves that are stored in separate tables, where each half is associated to an index number, which we call a reference index to distinguish it from data-term indices. A root table only contains pairs of reference indices, pointing to the corresponding halves in the two aforementioned tables; so each original list of data-term indices is represented at the root by just a pair of reference indices. This split can be applied recursively to the first half of lists of data-term indices as well as to the second half of such lists (if $k > 2$), producing a binary tree that contains pairs of indices in each node. In a non-leaf these are reference indices, pointing to pairs of indices in a child of this non-leaf. In a leaf each pair of indices is a part of an original list of data-term indices.

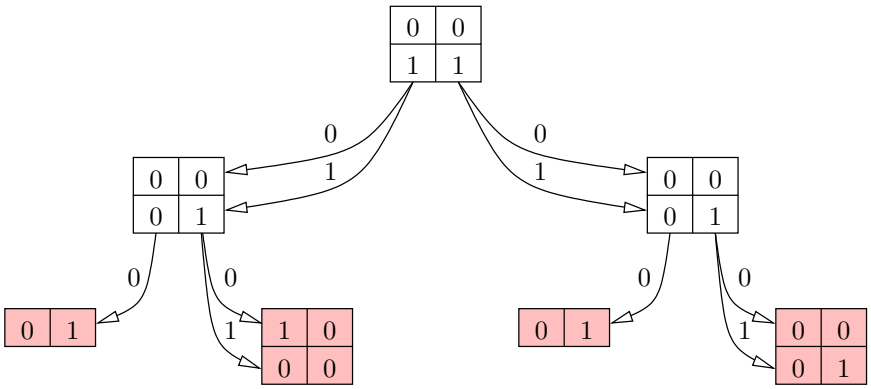
A state is added to the tree in a bottom-up fashion. First the list of data-term indices is chopped up into sublists of length two, and each of these sublists is added to the corresponding leaf if it is not yet present. Next, at each non-leaf, the pair of reference indices representing the part of the original list "stored" at this node is added, if this pair is not yet present. The state was already present if and only if the tree remains unchanged. This approach is explained by the following example.

Example 31. Let states be represented by lists of eight data-term indices. A processor P first needs to store the state 01100100, which is represented as 00 by means of the following tree.

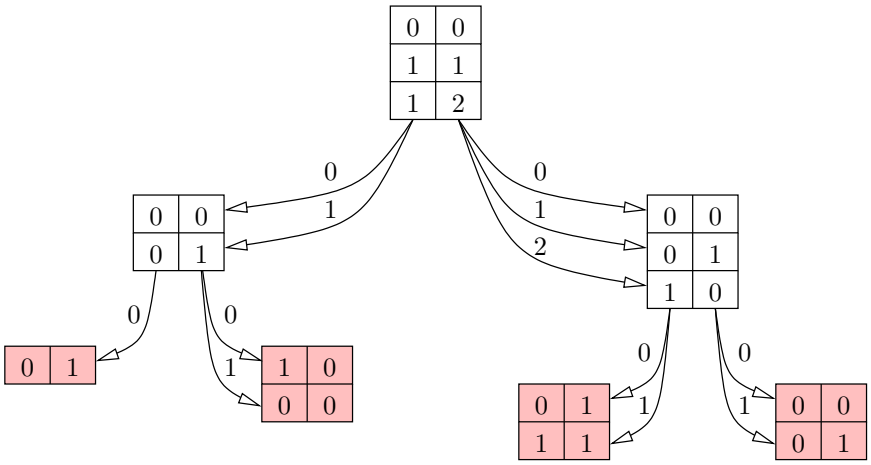


The leaves are colored to emphasize that they contain parts of the original list of data-term indices.

Next the state 01000101 needs to be stored at P. It is represented as 11, and the tree is adapted as follows.

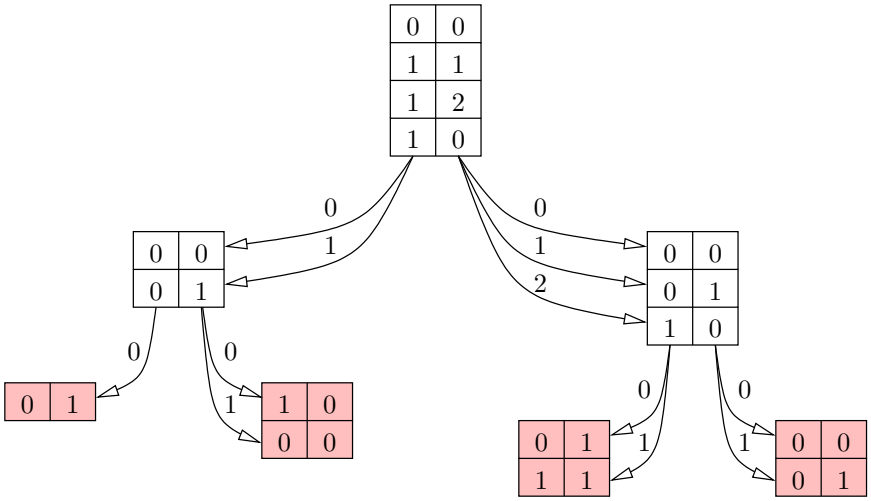


Next the state 01001100 needs to be stored at P. It is represented as 12, and the tree is adapted as follows.



Note that the left side of the tree remains unchanged. This is due to the fact that the first half of the list of data-term indices, 0100, coincides with the first half of the second stored state.

Finally the state 01000100 needs to be stored at P. It is represented as 10, and the tree is adapted as follows.



In this case only the root node changes. This is due to the fact that the first half of the list of data-term indices, 0100, coincides with the first half of the second (and third) stored state, while the second half, 0100, coincides with the second half of the first stored state.

In case states are represented by lists of data-term indices of which the length is not a power of 2, the tree representing the stored states is not a complete binary tree. In that case some nodes have only one child, being a leaf, and contain pairs of a reference index pointing to this leaf and a single element from an original list of data-term indices.

Algorithms have been developed for distributed versions of state space minimisation [21] and model checking [8], as well as specialised file formats to store distributed state spaces [19]. In [17] an overview is given of distributed state space generation and analysis tools that have been developed for μCRL, together with relevant case studies.

6.8 Abstraction

To reduce the size of a state space, one can bring it to a higher level of abstraction. That is, different states in the state space can be mapped to the same, *abstracted* state. Likewise, different actions in the state space can be mapped to the same, *abstracted* action.

Given a state space over a set of states S and a set of actions A. Assume surjective mappings $\pi : S \to \hat{S}$ and $\theta : A \to \hat{A}$, where $\hat{S}$ and $\hat{A}$ contain abstracted states and actions, respectively. The abstracted state space is obtained by applying π and θ to the original state space. That is, if $s \xrightarrow{a} s'$ in the original state space, then $\pi(s) \xrightarrow{\theta(a)} \pi(s')$ in the abstracted state space.

Example 32. Figure 6.2 depicts the state space of a bag of capacity $N \geq 3$, which contains elements from $\{0, 1\}$ (cf. Section 3.8). In state $s_{i,j}$, the bag contains i 0's and j 1's (so always $i + j \leq N$). The actions $in(b)$ and $out(b)$ represent putting a b into and taking a b out of the bag, respectively, for $b \in \{0, 1\}$.

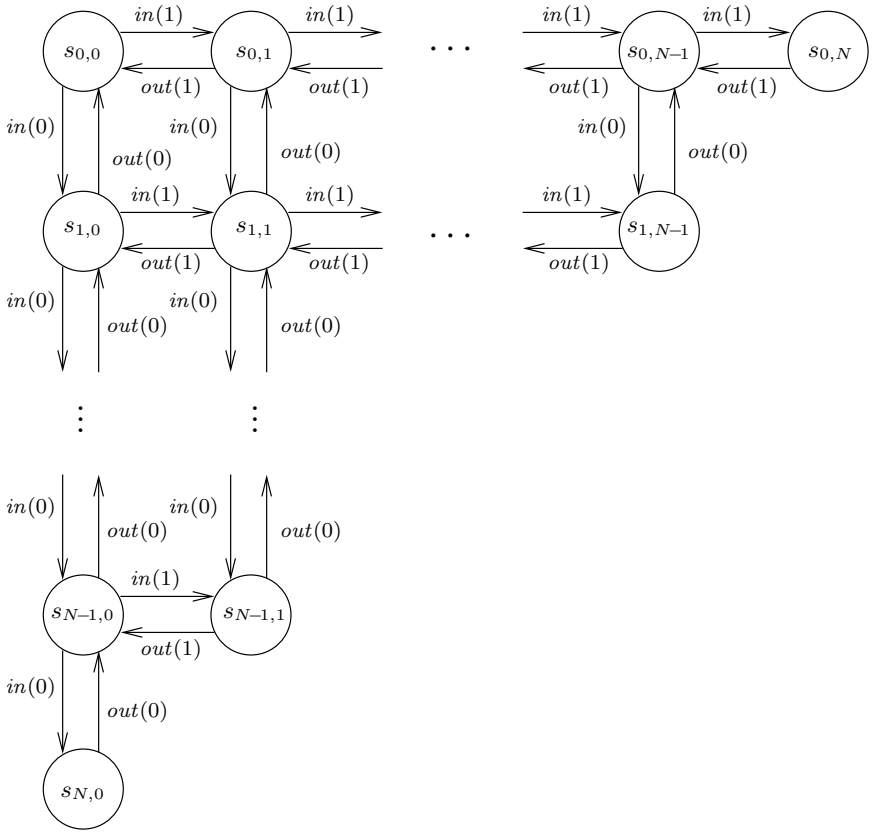


Fig. 6.2. The state space of a bag of size N

We define as set of abstracted states $\hat{S} = \{empty, middle, full\}$, where intuitively *empty* is the empty bag, *full* represents the bag with N elements, and *middle* represents those states where the bag is neither full nor empty. The set of abstracted actions is $\hat{A} = \{\hat{\imath}, \hat{o}\}$, where $\hat{\imath}$ represents putting an element into the bag, while $\hat{o}$ represents taking an element out of the bag. The mappings π and θ are defined as follows:

$$\begin{array}{lll} \pi(s_{0,0}) & = \mathit{empty} & \\ \pi(s_{i,j}) & = \mathit{middle} & 0 < i{+}j < N \\ \pi(s_{i,j}) & = \mathit{full} & i{+}j = N \\ & & \\ \theta(\mathit{in}(b)) & = \hat{\imath} & b \in \{0,1\} \\ \theta(\mathit{out}(b)) & = \hat{o} & b \in \{0,1\} \end{array}$$

The abstracted state space contains *must* transitions, denoted with a subscript $\Box$, and *may* transitions, denoted with a subscript $\Diamond$. An outgoing must transition at an abstracted state in the abstracted state space has a counterpart at all the corresponding states in the original state space. That is:

$\hat{s} \xrightarrow{\hat{a}}_{\Box} \hat{s}'$ if for all $s \in S$ with $\pi(s) = \hat{s}$ there is a transition $s \xrightarrow{a} s'$ with $\theta(a) = \hat{a}$ and $\pi(s') = \hat{s}'$.

An outgoing may transition at an abstracted state in the abstracted state space has a counterpart at one or more corresponding states in the original state space. That is:

$\hat{s} \xrightarrow{\hat{a}}_{\Diamond} \hat{s}'$ if there is a transition $s \xrightarrow{a} s'$ with $\pi(s) = \hat{s}$, $\theta(a) = \hat{a}$ and $\pi(s') = \hat{s}'$.

Since π is surjective, the must transitions are a subset of the may transitions.

Example 33. The abstracted state space for the bag of size N in Example 32 is depicted in Fig. 6.3. Must transitions are depicted as solid arrows, and may transitions as dashed arrows.

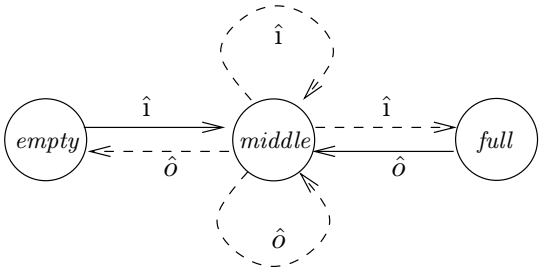


Fig. 6.3. The abstracted state space of a bag of size N

Since *empty* represents a single state from the original state space, its outgoing î-transition is by default a must transition. The state *full* has an outgoing $\hat{o}$-transition to *middle*; again this is a must transition, since each bag with N elements can do an $\mathit{out}(0)$- or $\mathit{out}(1)$-transition to a bag with $N-1$ elements. On the other hand, all outgoing transitions of *middle* are may transitions. After an output action, the resulting state is either *empty*, if the bag contained only one element, or *middle*, otherwise. Likewise, after an input

action, the resulting state is either *full*, if the bag contained $N-1$ elements, or *middle*, otherwise.

The μ-calculus from Section 6.6 can be defined for the abstracted state space as well, with formulas $\langle \hat{a} \rangle \, \phi$ and $[\hat{a}] \, \phi$, and with again $\mu X.\phi$ and $\nu X.\phi$ as minimal and maximal fixpoints. However, the meaning of such an abstracted μ-calculus formula on an abstracted state space is defined in a different fashion. Namely, the intention of abstraction is that a model checking result for the abstracted state space can be lifted to a result for the original state space. Given an abstracted μ-calculus formula ϕ, the set $C(\phi)$ of abstracted states that satisfy ϕ is defined as follows:

$$\begin{aligned}
C(\mathtt{T}) &= \hat{S} \\
C(\mathtt{F}) &= \emptyset \\
C(\phi \wedge \phi') &= C(\phi) \cap C(\phi') \\
C(\phi \vee \phi') &= C(\phi) \cup C(\phi') \\
C(\langle \hat{a} \rangle \, \phi) &= \{\hat{s} \in \hat{S} \mid \exists \hat{s}' \in C(\phi) \, (\hat{s} \xrightarrow{\hat{a}}_{\Box} \hat{s}')\} \\
C([\hat{a}] \, \phi) &= \{\hat{s} \in \hat{S} \mid \forall \hat{s}' \in \hat{S} \, ((\hat{s} \xrightarrow{\hat{a}}_{\Diamond} \hat{s}') \Rightarrow \hat{s}' \in C(\phi))\}
\end{aligned}$$

The definition of $C(\phi)$ is such that if $\pi(s) \in C(\phi)$ for some state s, then in the original state space, s is guaranteed to satisfy a concretisation of the abstracted formula ϕ. This concretisation is a regular μ-calculus formula, which is obtained by replacing all expressions $\langle \hat{a} \rangle$ and $[\hat{a}]$ in ϕ by $\langle \alpha \rangle$ and $[\alpha]$, respectively, where the regular expression α represents the union of all actions in $\theta^{-1}(\hat{a})$. Thus model checking on the abstracted state space can produce information about the original state space. An abstracted formula ϕ that is verified on the abstracted state space, is basically checked on a so-called over-approximation of the original state space, because in $C(\phi)$ both must and may transitions are taken into account.

The abstracted μ-calculus, and the definition of $C(\phi)$, can be extended without complication to expressions $\langle \hat{\beta} \rangle \, \phi$ and $[\hat{\beta}] \, \phi$, where $\hat{\beta}$ is a regular expression representing a set of traces over the abstracted actions (cf. the regular μ-calculus in Section 6.6.)

Example 34. The formula

$$[(\neg(\hat{\imath}))^* \cdot \hat{o}] \, \mathsf{F}$$

expresses that each output from the bag is preceded by an input to the bag. It holds in the initial state of the abstracted state space of the bag of size N in Example 33.

So the formula

$$[(\neg(in(0) \mid in(1)))^* \cdot (out(0) \mid out(1))] \, \mathsf{F}$$

holds in the initial state of the original state space in Example 32.

Actually, the stronger property

$$[(\neg(in(b)))^*\cdot out(b)]\,\mathsf{F}$$

holds in the initial state of the original state space, for $b \in \{0,1\}$. But this information is lost in the abstraction.

An abstract interpretation toolkit has been implemented for μCRL [95].

Exercises

Exercise 59. Describe the process declaration $X = (a + c)\cdot d\cdot X$ by means of an LPE.

Exercise 60. Linearise, using the default method,

$$\begin{aligned} Y(m{:}Nat) &= a(m)\cdot Z(S(m))\cdot Y(m)\\ Z(m{:}Nat) &= b(m)\cdot Z(S(m)) + c(m) \end{aligned}$$

Would the regular method terminate on this process declaration?

Exercise 61. Linearise, using the regular method,

$$\begin{aligned} Y(m{:}Nat) &= a(m)\cdot Z(S(m))\cdot Y(S(m))\\ Z(m{:}Nat) &= b(m)\cdot Z(m) + c(S(m)) \end{aligned}$$

Exercise 62. Let $a \mid b = c$, and consider the LPE

$$X(n{:}Nat) = a(f(n))\cdot X(g(n)) \lhd h(n) \rhd \delta + b(f'(n))\cdot X(g'(n)) \lhd h'(n) \rhd \delta$$

Give an LPE Y such that the process term $X(n_1) \parallel \cdots \parallel X(n_k)$ is equal to $Y(n_1,\ldots,n_k)$, for any $k > 0$ and $n_1,\ldots,n_k \in Nat$.

Exercise 63. Give an example to show that a Bloom filter can produce false positives.

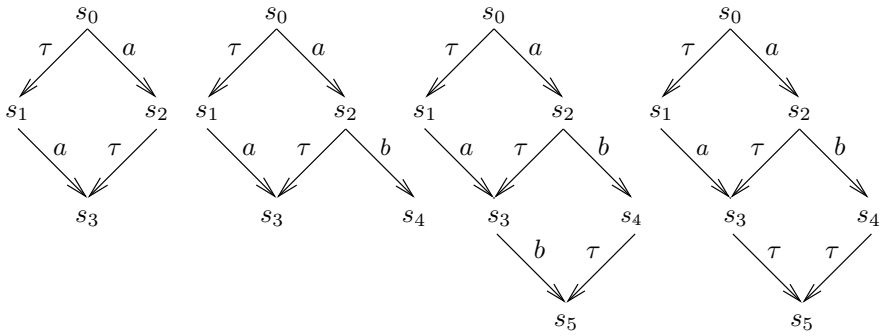
Exercise 64. Apply the minimisation algorithm modulo branching bisimilarity to the state spaces belonging to the following process terms and declarations:

1. $(a + \tau\cdot b)\cdot\delta$
2. $(a + \tau\cdot(a + b))\cdot\delta$
3. $(a\cdot b\cdot c + a\cdot b\cdot d)\cdot\delta$
4. $((a + b)\cdot\tau + \tau\cdot b)\cdot(a + b)\cdot\delta$
5. $a\cdot a\cdot a\cdot a\cdot\delta$
6. $X = (\tau + a)\cdot Y$
   $Y = b\cdot X$

7. $X = (\tau + a)\cdot Y$
   $Y = (a + b)\cdot X$

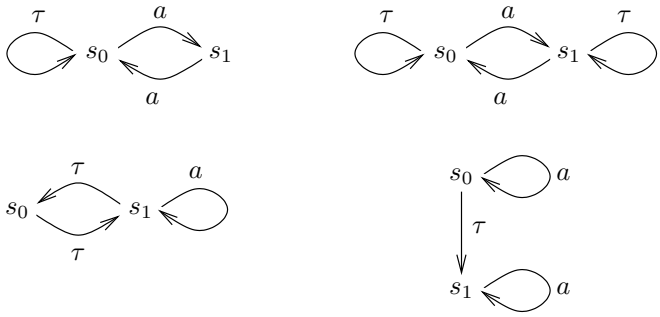
Exercise 65. Explain how the minimisation algorithm modulo branching bisimilarity can be adapted to take into account the successful termination predicate $\downarrow$ (cf. Definition 2).

Exercise 66. Give the maximal confluent sets of τ-transitions for the following four state spaces:



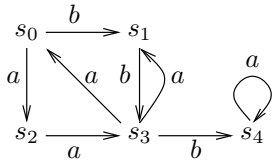
In each case give the state space that results when the confluent τ-transitions are prioritised.

Exercise 67. Give the maximal confluent sets of τ-transitions for the following four state spaces:



In which cases can we prioritise the confluent τ-transitions? And what is then the resulting state space?

Exercise 68. Consider the state space



Say for each of the following ACTL formulas in which states they are satisfied:

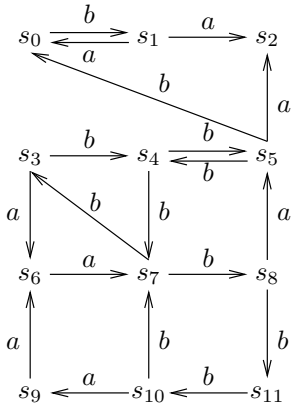
1. $\langle a \rangle \langle b \rangle \mathtt{T}$
2. $\langle b \rangle \langle a \rangle \mathtt{T}$
3. $\mathtt{EG} \langle a \rangle \mathtt{T}$
4. $\mathtt{EG} \langle b \rangle \mathtt{T}$
5. $\mathtt{EG} (\langle a \rangle \langle b \rangle \mathtt{T} \vee \langle b \rangle \langle a \rangle \mathtt{T})$
6. $\mathtt{E} (\neg \mathtt{EG} \langle a \rangle \mathtt{T} \vee \neg \mathtt{EG} \langle b \rangle \mathtt{T}) \mathtt{U} (\langle b \rangle \neg \langle b \rangle \mathtt{T})$

Furthermore, for each state in the state space, give an ACTL formula that is only satisfied by this state.

Exercise 69. Let $\mathsf{Act} = \{a, b\}$. Give ACTL formulas expressing the following properties:

1. there is an execution sequence to a deadlock state;
2. there is an infinite execution sequence.

Exercise 70. Consider the state space



Say for each of the following ACTL formulas in which states they are satisfied:

1. $[a] \langle b \rangle \mathtt{T}$
2. $\langle a \rangle \langle b \rangle \mathtt{T}$
3. $\neg \mathtt{EG} \neg \langle a \rangle \langle b \rangle \mathtt{T}$
4. $\langle b \rangle \langle a \rangle \neg (\langle a \rangle \mathtt{T} \vee \langle b \rangle \mathtt{T})$
5. $\langle b \rangle \langle b \rangle \langle b \rangle \langle a \rangle \langle b \rangle \langle a \rangle \neg (\langle a \rangle \mathtt{T} \vee \langle b \rangle \mathtt{T})$
6. $\mathtt{E}(\neg\langle a\rangle\langle b\rangle\mathtt{T})\mathtt{U}((\langle a\rangle\langle a\rangle\neg(\langle a\rangle\mathtt{T} \vee \langle b\rangle\mathtt{T})) \vee (\langle b\rangle\langle b\rangle\langle b\rangle\langle a\rangle\langle b\rangle\langle a\rangle\neg(\langle a\rangle\mathtt{T} \vee \langle b\rangle\mathtt{T})))$

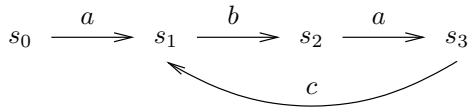
Exercise 71. Suppose that X does not occur in ϕ. Is $\mu X.\phi$ always equivalent to ϕ?

Exercise 72. Is the binary operation implication monotonic?

Exercise 73. Compute solutions for the following two formulas, with respect to the state space from Example 26:

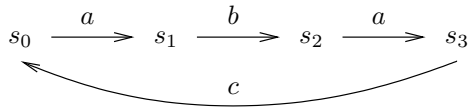
1. $\nu X.(\langle a\rangle X \vee \langle b\rangle \mathtt{T})$;
2. $\mu Y.(\langle a\rangle Y \vee \langle b\rangle \mathtt{T})$.

Exercise 74. Consider the state space



Compute the solutions for X and Y in the formula $\nu X.(\langle c\rangle\, \mu Y.(\langle a\rangle X \vee \langle b\rangle Y))$.

Exercise 75. Consider the state space



Compute the solutions for X and Y in the formula $\nu X.(\langle c\rangle\, \mu Y.(\langle a\rangle X \vee \langle b\rangle Y))$.

Exercise 76. Consider a mutual exclusion algorithm, where actions $claim_i$ and $release_i$ mean that process p_i claims or releases the critical region. Express in the regular μ-calculus the following properties:

1. each occurrence of $release_1$ is preceded by an occurrence of $claim_1$;
2. each occurrence of $claim_1$ is eventually followed by an occurrence of $release_1$;
3. after an occurrence of $claim_1$, no $claim_2$ can happen until an occurrence of $release_1$.

Exercise 77. Express in the regular μ-calculus the property 'after an occurrence of the action *send*, each fair trace will eventually perform the action *read*'.

Exercise 78. Express the operators $\mathtt{E}\,\phi\,\mathtt{U}\,\phi'$ and $\mathtt{EG}\,\phi$ from ACTL in the regular μ-calculus.

Exercise 79. Let a processor first store the list of indices 10010011, then 10110010, then 10010010, and finally 01100100. Show how the tree at this processor evolves.

Exercise 80. Give an abstracted state space for the state space of a bag of capacity $N = 1$ and of capacity $N = 2$.

Exercise 81. Given the state space of the μCRL specification $\partial_H(S(0) \parallel R(0) \parallel K \parallel L)$ of the ABP (see Section 5.1), with $\Delta = \{d, e\}$.

Consider the following abstraction. All communication actions over the internal channels and j are mapped to a special action $\hat{c}$, while the actions $r_{\mathrm{A}}(d)$, $r_{\mathrm{A}}(e)$, $s_{\mathrm{D}}(d)$ and $s_{\mathrm{D}}(e)$ are left in tact. The abstracted state space has three states: empty, d and e, where each state in the concrete state space is mapped to the datum that is currently being transported, or empty if no datum is being transported.

Give precise definitions of the mappings π and θ, and draw the state space of the abstracted ABP.

Exercise 82. Give an abstraction of the stopwatch in Example 7, where the abstracted state space contains two states, representing that the time is zero or greater than zero.

7

Symbolic Methods

Exhaustive state space generation and exploration suffers from the ominous state explosion problem, which refers to the fact that the number of states in a distributed system tends to grow exponentially with respect to its number of concurrent components. In this chapter, some methods are described to analyse and adapt μCRL specifications at a symbolic level. These methods do not require the generation of the state space belonging to an LPE, thus circumventing the state explosion problem.

7.1 CL-RSP

An LPE is *convergent* if it does not give rise to any infinite sequence of τ-transitions (cf. the notion of guardedness in Section 4.1.) The derivation rule *CL-RSP* (*Convergent Linear Recursive Specification Principle*) [16] says that the solutions for a convergent LPE are all equal. Convergence is essential for the soundness of CL-RSP modulo rooted branching bisimilarity (cf. Section 4.4). Hennessy and Lin [68] introduced a similar derivation rule called UFI-O.

Definition 6 (CL-RSP). *Let the LPE*

$$X(d{:}D) \;=\; \textstyle\sum_{i:I} \sum_{e:E} a_i(f_i(d,e)) \cdot X(g_i(d,e)) \lhd h_i(d,e) \rhd \delta$$

be convergent. Let $P(d)$ range over process terms. If for all $d \in D$,

$$P(d) \;=\; \textstyle\sum_{i:I} \sum_{e:E} a_i(f_i(d,e)) \cdot P(g_i(d,e)) \lhd h_i(d,e) \rhd \delta$$

then $P(d) = X(d)$ for all $d \in D$.

Without proof we state that CL-RSP is sound modulo rooted branching bisimilarity, meaning that if the axioms are all sound modulo $\underline{\leftrightarrow}_{rb}$, then from

$$P(d) \;=\; \textstyle\sum_{i:I} \sum_{e:E} a_i(f_i(d,e)) \cdot P(g_i(d,e)) \lhd h_i(d,e) \rhd \delta$$

for all $d \in D$, it follows that $P(d) \leftrightarrows_{rb} X(d)$ for all $d \in D$. Convergence of the LPE is essential for the soundness of CL-RSP. For example, consider the LPE $X = \tau{\cdot}X \lhd \mathtt{T} \rhd \delta$; note that this LPE not convergent. By axioms B1 and C1 (see Tables 4.1 and 3.4, respectively), for all $a \in \mathsf{Act}$, $\tau{\cdot}a = \tau{\cdot}\tau{\cdot}a = \tau{\cdot}\tau{\cdot}a \lhd \mathtt{T} \rhd \delta$. So without the restriction to convergent LPEs, CL-RSP would yield $\tau{\cdot}a = X$ for all $a \in \mathsf{Act}$. Clearly, these equalities are not sound, since $\tau{\cdot}a$ and $\tau{\cdot}b$ are not rooted branching bisimilar if $a \neq b$.

Example 35. Consider the following two LPEs:

$$X = a{\cdot}X$$

$$Y(b{:}Bool) = a{\cdot}Y(\neg b)$$

The second LPE basically consists of two recursive equations: $Y(\mathtt{T}) = a{\cdot}Y(\mathtt{F})$ and $Y(\mathtt{F}) = a{\cdot}Y(\mathtt{T})$. If we substitute X for both $Y(\mathtt{T})$ and $Y(\mathtt{F})$ in these recursive equations, they both yield $X = a{\cdot}X$. This equality follows immediately from the first LPE. Since the LPE for Y is convergent, by CL-RSP, $X = Y(\mathtt{T})$ and $X = Y(\mathtt{F})$.

Example 36. Consider the following two LPEs:

$$X(m{:}Nat) = a(2m){\cdot}X(S(m))$$

$$Y(n{:}Nat) = a(n){\cdot}Y(S(S(n)))$$

Substituting $Y(2m)$ for $X(m)$ in the first LPE, for $m \in Nat$, yields $Y(2m) = a(2m){\cdot}Y(2S(m))$. This equality follows from the second LPE by substituting $2m$ for n, because $S(S(2m)) = 2S(m)$. Since the LPE for X is convergent, by CL-RSP, $Y(2m) = X(m)$ for $m \in Nat$.

7.2 Invariants

Definition 7 (Invariant). *A mapping $\mathcal{I} : D \to Bool$ is an* invariant *for an LPE (written as in Definition 4) if, for all $d \in D$, $i \in I$ and $e \in E$,*

$$(\mathcal{I}(d) \wedge h_i(d, e)) \;\Rightarrow\; \mathcal{I}(g_i(d, e))$$

Intuitively, an invariant characterises the set of reachable states of an LPE. That is, if $\mathcal{I}(d) = \mathtt{T}$ and state d can evolve to state d' in zero or more transitions, then $\mathcal{I}(d') = \mathtt{T}$. Namely, if $\mathcal{I}$ holds in state d and it is possible to execute $a_i(f_i(d, e))$ in this state (meaning that $h_i(d, e) = \mathtt{T}$), then it is ensured by Definition 7 that $\mathcal{I}$ holds in the resulting state $g_i(d, e)$.

Invariants tend to play a crucial role in the algebraic verifications of systems that involve data. Namely, system properties generally do not hold in all states, but only in the reachable ones. Since invariants are guaranteed to hold in all reachable states, they can be used in proving system properties for the reachable states only.

Example 37. Consider the LPE $X(n{:}Nat) = a(n){\cdot}X(S(S(n)))$. Invariants for this LPE are

$$\mathcal{I}_1(n) = \begin{cases} \mathtt{T} \text{ if } n \text{ is even} \\ \mathtt{F} \text{ if } n \text{ is odd} \end{cases} \qquad \mathcal{I}_2(n) = \begin{cases} \mathtt{F} \text{ if } n \text{ is even} \\ \mathtt{T} \text{ if } n \text{ is odd} \end{cases}$$

In Definition 6, CL-RSP ranged over the complete data set D. That is, if the process terms $P(d)$ are a solution for a convergent LPE X for all $d \in D$, then we could conclude $P(d) = X(d)$ for all $d \in D$. Actually, it is sufficient to show that the $P(d)$ are a solution for X for all *reachable* $d \in D$, which can be captured by some invariant $\mathcal{I}$. In that case we can of course only conclude $P(d) = X(d)$ for data elements d with $\mathcal{I}(d) = \mathtt{T}$. CL-RSP with invariants is similar to the derivation rule UFI-Inv of Rathke [96].

Definition 8 (CL-RSP with invariants). *Let the LPE*

$$X(d{:}D) = \textstyle\sum_{i:I} \sum_{e:E} a_i(f_i(d,e)){\cdot}X(g_i(d,e)) \lhd h_i(d,e) \rhd \delta$$

be convergent, and let $\mathcal{I} : D \to Bool$ be an invariant for this LPE. Let $P(d)$ range over process terms. If for all $d \in D$ with $\mathcal{I}(d) = \mathtt{T}$,

$$P(d) = \textstyle\sum_{i:I} \sum_{e:E} a_i(f_i(d,e)){\cdot}P(g_i(d,e)) \lhd h_i(d,e) \rhd \delta$$

then $P(d) = X(d)$ for all $d \in D$ with $\mathcal{I}(d) = \mathtt{T}$.

CL-RSP with invariants is sound modulo rooted branching bisimilarity.

Example 38. Let $even : Nat \to Bool$ map even numbers to $\mathtt{T}$ and odd numbers to $\mathtt{F}$ (see Exercise 2). Consider the following two LPEs:

$$X(n{:}Nat) = a(even(n)){\cdot}X(S(S(n)))$$

$$Y = a(\mathtt{T}){\cdot}Y$$

Substituting Y for $X(n)$ for even numbers n in the first LPE yields $Y = a(\mathtt{T}){\cdot}Y$, which follows from the second LPE.

$$\mathcal{I}(n) = \begin{cases} \mathtt{T} \text{ if } n \text{ is even} \\ \mathtt{F} \text{ if } n \text{ is odd} \end{cases}$$

is an invariant for the first LPE (cf. Example 37), and this LPE is convergent. So by Definition 8, $Y = X(n)$ for even n.

7.3 Cones and Foci

The *cones and foci* method of [63] aims to eliminate all hidden actions from an LPE (see Definition 4). The main idea of this technique is that often τ-transitions progress inertly towards a state in which no hidden action can

be executed. Such a state is declared to be a *focus point*; the *cone* of a focus point consists of the states that can reach this focus point by a string of hidden actions. Figure 7.1 visualises the core idea underlying this method. Note that the external actions at the edge of the depicted cone can also be executed in the ultimate focus point; this is essential if one wants to apply the cones and foci method, as otherwise the τ-transitions in the cone would not be inert.

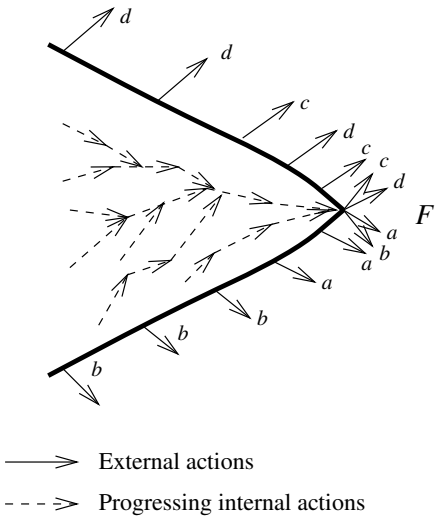
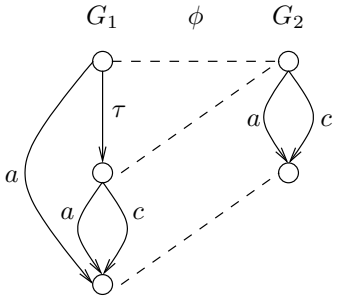


Fig. 7.1. A focus point and its cone

Assume a convergent LPE X (see Section 7.1). Due to convergence, each state belongs to the cone of some focus point. In the cones and foci method, the states of X are mapped to states of an LPE Z that does not contain hidden actions; intuitively, Z represents the external behaviour of X. This *state mapping* ϕ must satisfy a number of *matching criteria*, which ensure that the mapping establishes a branching bisimulation relation (see Definition 2) between the two LPEs in question, and moreover that all states in a cone of X are mapped to the same state in Z. The state mapping ϕ satisfies the matching criteria if for all states d and d' of X and $a \in \mathsf{Act}$:

- if $d \xrightarrow{\tau} d'$, then $\phi(d) = \phi(d')$;
- if $d \xrightarrow{a(\mathbf{e})} d'$, then $\phi(d) \xrightarrow{a(\mathbf{e})} \phi(d')$; and
- if d is a focus point for X and $\phi(d) \xrightarrow{a(\mathbf{e})} d''$, then $d \xrightarrow{a(\mathbf{e})} d'$ with $\phi(d') = d''$.

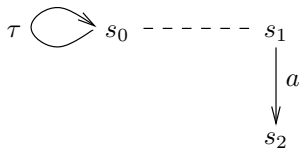
Example 39. Below is depicted a mapping ϕ from the states in a state space G_1 to states in a state space G_2 without τ's. Note that each state in G_1 belongs to the cone of some focus point.



It is not hard to see that ϕ satisfies the matching criteria. Note that ϕ establishes a branching bisimulation relation.

The following example shows that for the soundness of the cones and foci method, modulo branching bisimilarity, it is essential that the LPE X is convergent.

Example 40. Note that the state s_0 below does not belong to the cone of a focus point.



It is not hard to see that if $\phi(s_0) = s_1$, then ϕ satisfies the matching criteria. However, ϕ does not establish a branching bisimulation relation.

The crux of the cones and foci method is that the matching criteria can be formulated syntactically, in terms of relations between data terms. Thus, one obtains clear proof obligations. We proceed to present precise definitions of the notions that underly the cones and foci method, including syntactic formulations of the matching criteria.

Definition 9 (Matching criteria). *Assume a convergent LPE*

$$X(d{:}D) = \sum_{a \in \mathsf{Act} \cup \{\tau\}} \sum_{e:E} a(f_a(d,e)) \cdot X(g_a(d,e)) \lhd h_a(d,e) \rhd \delta$$

Furthermore, assume an LPE without hidden actions

$$Z(d'{:}D') = \sum_{b \in \mathsf{Act}} \sum_{e:E} b(f'_b(d',e)) \cdot Z(g'_b(d',e)) \lhd h'_b(d',e) \rhd \delta$$

A state mapping $\phi : D \to D'$ *satisfies the* matching criteria *at a* $d \in D$ *if for all* $b \in \mathsf{Act}$*:*

- $\forall e{:}E\,(h_\tau(d,e) \Rightarrow \phi(d) = \phi(g_\tau(d,e)))$
- $\forall e{:}E\,(h_b(d,e) \Rightarrow (h'_b(\phi(d),e) \wedge f_b(d,e) = f'_b(\phi(d),e) \wedge \phi(g_b(d,e)) = g'_b(\phi(d),e)))$
- $FC_X(d) \;\Rightarrow\; \forall e{:}E\,(h'_b(\phi(d),e) \Rightarrow h_b(d,e))$

The first matching criterion in Definition 9 requires that all states in a cone of X are mapped to the same state in Z. The second criterion expresses that if a state in X can perform an external transition, then it can be simulated by the corresponding state in Z. Finally, the third criterion expresses that if a state in Z can perform an external transition, then the corresponding *focus points* in X can simulate this transition; the so-called *focus condition* $FC_X(d)$, which abbreviates $\forall e{:}E\,(\neg h_\tau(d,e))$, expresses that d is a focus point.

As in the case of CL-RSP, it is sufficient if the matching criteria are only satisfied for the reachable states of X (cf. Section 7.2). The matching criteria above can therefore be weakened by adding a condition that $\mathcal{I}(d) = \mathtt{T}$, where $\mathcal{I}$ is some invariant for X (see Definition 7).

Theorem 1. *Assume a convergent LPE X over D. Let $\mathcal{I}$ be an invariant for X. Furthermore, assume an LPE Z without hidden actions. If $\phi : D \to D'$ satisfies the matching criteria from Definition 9 for all $d \in D$ with $\mathcal{I}(d) = \mathtt{T}$, then for all $d \in D$ with $\mathcal{I}(d) = \mathtt{T}$:*

$$X(d) \leftrightarrow_b Z(\phi(d)).$$

If moreover $FC_X(d)$, then

$$X(d) \leftrightarrow_{rb} Z(\phi(d)).$$

If there are non-inert τ's, meaning that the LPE Z contains τ's, then these must be treated in the same way as actions from Act. In that case, non-inert τ's must be traced in the LPE X, and renamed to a fresh action (say $\bar{\tau}$) in Act. The τ's in Z are then also renamed to $\bar{\tau}$. In practice, tracing non-inert τ's in X can be difficult.

In [47], an adaptation of the cones and foci technique is proposed, in which LPEs do not need to be convergent. One is free to assign which states are focus points, as long as each state is able to reach a focus point by means of a sequence of τ-transitions. The matching criteria remain unchanged. This whole framework has been cast in the theorem prover PVS [88], and was used in [5] for the verification of the SWP as presented in Section 5.3.

7.4 Verification of the Tree Identify Protocol

In this section we give a formal proof, based on the cones and foci method, that for all connected networks that are free of cycles, the synchronous version of the TIP produces a unique root.

We recall that Implementation A of the TIP, here without parametrisation of the *leader* action, consists of node processes

$$\begin{aligned}
&X(i{:}Node, p{:}Nodelist, s{:}State)\\
&= \textstyle\sum_{j:Node} r(j,i)\cdot X(i, p\backslash\{j\}, s) \lhd \mathtt{T} \rhd \delta\\
&+ \textstyle\sum_{j:Node} s(i,j)\cdot X(i,p,1) \lhd p = \{j\} \wedge s = 0 \rhd \delta\\
&+ \mathit{leader}\cdot X(i,p,1) \lhd p = [] \wedge s = 0 \rhd \delta
\end{aligned}$$

i is the identifier of the node; p is the list of possible parents of node i; and s is the state 0 or 1. In state 0 a node is looking for a parent, while in state 1 it has found a parent or has become the root. A network is specified by

$$\tau_I(\partial_H(X(i_0, p_0[i_0], 0) \| \cdots \| X(i_k, p_0[i_k], 0)))$$

where $p_0[i]$ consists of the neighbours of node i in the network. We will prove that, modulo branching bisimilarity, the external behaviour of this process term, for any connected network without cycles, is $\mathit{leader}\cdot\delta$.

Let p be of the data type *Nodelistlist*, meaning that p maps nodes to lists of nodes, and let s be of the data type *Statelist*, meaning that s maps nodes to $\{0,1\}$. By CL-RSP, and using linearisation with respect to type II recursion variables (see Section 6.2), one can prove that

$$\tau_I(\partial_H(X(i_0, p[i_0], s[i_0]) \| \cdots \| X(i_k, p[i_k], s[i_k])))$$

is equal to $Y(p,s)$, defined by the following LPE:

$$\begin{aligned}
&Y(p{:}Nodelistlist, s{:}Statelist)\\
&= \textstyle\sum_{i,j:Node} \tau\cdot Y(p[i] := p[i]\backslash\{j\}, s[j] := 1) \lhd p[j] = \{i\} \wedge s[j] = 0 \rhd \delta\\
&+ \textstyle\sum_{i:Node} \mathit{leader}\cdot Y(p, s[i] := 1) \lhd \mathit{empty}(p[i]) \wedge s[i] = 0 \rhd \delta
\end{aligned}$$

At the right-hand side of this recursive equation, $Y(p[i] := p[i]\backslash\{j\}, s[j] := 1)$ has the same *Nodelistlist* and *Statelist* as $Y(p,s)$, except that in the former process term $p[i]$ is changed into $p[i]\backslash\{j\}$ and $s[j]$ is set to 1; likewise for $Y(p, s[i] := 1)$. See Example 20 and Exercise 83 for examples how to derive such an equality, using CL-RSP. The initial state of the LPE Y is $Y(p_0, s_0)$, with $s_0[i] = 0$ for all $i \in Node$.

For the soundness of this application of CL-RSP it is important to note that the LPE Y is convergent. This is due to the fact that each execution of a hidden action reduces the number of nodes j with $s[j] = 0$. Hence, there cannot exist an infinite sequence of τ-transitions.

We list four invariants for the LPE Y. They include data parameters $p{:}Nodelistlist$ and $s{:}Statelist$ (and $\mathcal{I}_1$ also $i, j{:}Node$).

$$\mathcal{I}_1 : j \in p[i] \vee i \in p[j]$$

$$\mathcal{I}_2 : \forall i, j{:}\mathit{Node}\,((j \notin p[i] \wedge i \in p[j]) \Rightarrow s[j] = 1)$$

$$\mathcal{I}_3 : \forall i, j{:}\mathit{Node}\,(s[j] = 1 \Rightarrow (\mathit{empty}(p[j]) \vee \mathit{singleton}(p[j])))$$

$$\mathcal{I}_4 : \forall i, j{:}\mathit{Node}\,((j \in p[i] \wedge s[i] = 0) \Rightarrow (i \in p[j] \wedge s[j] = 0))$$

Note that $\mathcal{I}_2$, $\mathcal{I}_3$ and $\mathcal{I}_4$ hold in the initial state $Y(p_0, s_0)$. Moreover, $\mathcal{I}_1$ holds in $Y(p_0, s_0)$ for neighbours i, j.

We show that the first three formulas are invariants for Y. The fourth one is left as an exercise for the reader (see Exercise 91).

1. Suppose $j \in p[i]$, while after executing some action, in the resulting state $j \notin p[i]$. Then this action was τ, and in the resulting state $p[j] = \{i\}$, so in particular $i \in p[j]$.
   Likewise, suppose $i \in p[j]$, while after executing some action, in the resulting state $i \notin p[j]$. Then this action was τ, and in the resulting state $p[i] = \{j\}$, so in particular $j \in p[i]$.
2. If $i \notin p[j]$ or $s[j] = 1$, then after performing a transition this still holds. Suppose $j \in p[i]$ and $s[j] = 0$, while after executing some action $j \notin p[i]$. Then this action was τ, and in the resulting state $s[j] = 1$.
3. If $\mathit{empty}(p[j]) \vee \mathit{singleton}(p[j])$, then after performing an action this formula still holds, because no elements are ever added to $p[j]$.
   Suppose $s[j] = 0$ and $p[j]$ contains more than one element. Then after executing some action still $s[j] = 0$.

We derive one more invariant $\mathcal{I}$ for Y, stating that no more than one root is elected. Namely, if the list $p[i]$ of possible parents of a node i has become empty, then all other nodes are already finished.

Lemma 1 (Uniqueness of the root). *For all i and j,*

$$(\mathit{empty}(p[i]) \wedge j \neq i) \Rightarrow (s[j] = 1 \wedge \mathit{singleton}(p[j]))$$

Proof. By connectedness, there are distinct nodes $i = i_0, i_1, \ldots, i_m = j$ with $i_{k+1} \in p_0[i_k]$ for $k = 0, \ldots, m-1$. We derive, by induction on k, that $i_{k-1} \in p[i_k]$, $s[i_k] = 1$ and $\mathit{singleton}(p[i_k])$ for $k = 1, \ldots, m-1$. We start with the base case $k = 1$.

- $\mathit{empty}(p[i_0]) \Rightarrow i_1 \notin p[i_0]$
- $(\mathcal{I}_1 \wedge i_1 \in p_0[i_0] \wedge i_1 \notin p[i_0]) \Rightarrow i_0 \in p[i_1]$
- $(\mathcal{I}_2 \wedge i_1 \notin p[i_0] \wedge i_0 \in p[i_1]) \Rightarrow s[i_1] = 1$
- $(\mathcal{I}_3 \wedge s[i_1] = 1 \wedge i_0 \in p[i_1]) \Rightarrow \mathit{singleton}(p[i_1])$

We proceed with the inductive case. We know that $i_{k+1} \in p_0[i_k]$ and $i_{k+1} \neq i_{k-1}$, and by induction $i_{k-1} \in p[i_k]$ and $\mathit{singleton}(p[i_k])$. Hence,

- $(\mathit{singleton}(p[i_k]) \wedge i_{k-1} \in p[i_k] \wedge i_{k+1} \neq i_{k-1}) \Rightarrow i_{k+1} \notin p[i_k]$
- $(\mathcal{I}_1 \wedge i_{k+1} \in p_0[i_k] \wedge i_{k+1} \notin p[i_k]) \Rightarrow i_k \in p[i_{k+1}]$
- $(\mathcal{I}_2 \wedge i_{k+1} \notin p[i_k] \wedge i_k \in p[i_{k+1}]) \Rightarrow s[i_{k+1}] = 1$
- $(\mathcal{I}_3 \wedge s[i_{k+1}] = 1 \wedge i_k \in p[i_{k+1}] \Rightarrow \mathit{singleton}(p[i_{k+1}])$

We conclude that $s[i_m] = 1$ and $\mathit{singleton}(p[i_m])$. □

We recall that (p, s) is a *focus point* of the LPE Y if $Y(p, s)$ cannot execute a hidden action. To be more precise, the *focus condition* is that there do not exist nodes i and j with $p[j] = \{i\} \wedge s[j] = 0$.

The LPE for the external behaviour is

$$Z(b{:}\mathit{Bool}) \;=\; \mathit{leader}{\cdot}Z(\mathtt{F}) \lhd b \rhd \delta$$

Clearly, $Z(\mathtt{T}) = \mathit{leader}{\cdot}\delta$ and $Z(\mathtt{F}) = \delta$.

We define the *state mapping* ϕ from pairs (p, s) to *Bool* by

$$\phi(p,s) = \begin{cases} \mathtt{T} \text{ if } s[i] = 0 \text{ for some node } i \\ \mathtt{F} \text{ if } s[i] = 1 \text{ for all nodes } i. \end{cases}$$

Then the matching criteria from Definition 9, applied to the LPEs Y and Z, produce the following four formulas:

$$\forall i, j{:}\mathit{Node}\,((p[j] = \{i\} \wedge s[j] = 0) \;\Rightarrow\; \phi(p,s) = \phi(p[i] := p[i] \backslash \{j\}, s[j] := 1))$$

$$\forall i{:}\mathit{Node}\,((\mathit{empty}(p[i]) \wedge s[i] = 0) \;\Rightarrow\; \phi(p,s))$$

$$\forall i{:}\mathit{Node}\,((\mathit{empty}(p[i]) \wedge s[i] = 0) \;\Rightarrow\; \neg\phi(p, s[i] := 1))$$

$$\forall i, j{:}\mathit{Node}\,((p[j] \neq \{i\} \vee s[j] = 1) \wedge \phi(p,s)) \;\Rightarrow\; \exists i'{:}\mathit{Node}\,(\mathit{empty}(p[i']) \wedge s[i'] = 0)$$

We show that these four formulas are true.

- Since $i \in p[j]$ and $s[j] = 0$, by invariant $\mathcal{I}_4$, $s[i] = 0$. So $\phi(p,s) = \mathtt{T} = \phi(p[i]\backslash\{j\}, s[j] := 1)$, because $s[i] = 0$ at both the left- and the right-hand side.
- $\phi(p,s)$ because $s[i] = 0$.
- By uniqueness of the root, $\mathit{empty}(p[i])$ implies $s[j] = 1$ for all $j \neq i$. Hence, $\neg\phi(p, s[i] := 1)$.

- Since $\phi(p,s)$, there is a node i' with $s[i'] = 0$. Suppose $p[i']$ is non-empty; we derive a contradiction.
  Let $j \in p[i]$ and $s[i] = 0$ for some nodes i, j. By invariant $\mathcal{I}_4$, $i \in p[j]$ and $s[j] = 0$. Then by the assumption in the matching criterion $p[j] \neq \{i\}$, so there is a $k \neq i$ with $k \in p[j]$. Likewise $s[k] = 0$ and there is an $\ell \neq j$ with $\ell \in p[k]$, etc. This contradicts the fact that there is no cycle.

  By Theorem 1, if p establishes a connected network without cycles, then

$$Y(p,s) \;\underline{\leftrightarrow}_b\; Z(\phi(p,s))$$

This implies

$$\begin{aligned} & \tau_I(\partial_H(X(i_0,p_0[i_0],0) \| \cdots \| X(i_k,p_0[i_k],0))) \\ \underline{\leftrightarrow}_{rb}\; & Y(p_0,s_0) \\ \underline{\leftrightarrow}_b\; & Z(\mathtt{T}) \\ \underline{\leftrightarrow}_{rb}\; & \mathit{leader}{\cdot}\delta \end{aligned}$$

7.5 Partial Order Reduction

Much research is devoted to algorithms that use the confluence notion for τ-transitions from Section 6.5 to generate a reduced state space from a formal system specification. Collectively, these methods are called *partial order reduction* techniques. In this section it is explained how it can be detected that a summand of an LPE (see Definition 4) gives rise to confluent τ-transitions in the corresponding state space, and how this information can be exploited during the generation of the state space from this LPE. This can lead to a considerable reduction of the number of states; in some cases, the number of states of the reduced state space can actually grow in a linear fashion with respect to the number of concurrent components in the original system.

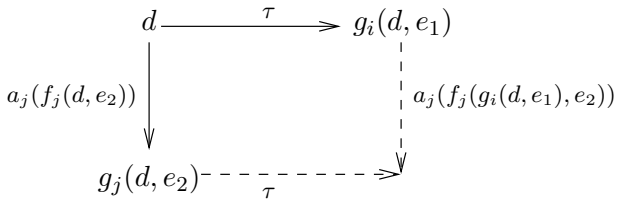
Symbolic Priorisation of Confluent τ-Summands

Assume an LPE

$$X(d{:}D) \;=\; \sum_{i:I} \sum_{e:E} a_i(f_i(d,e)){\cdot}X(g_i(d,e)) \lhd h_i(d,e) \rhd \delta$$

We want to establish which τ-summands give rise to confluent τ-transitions. Let $a_i = \tau$, and suppose that there are transitions $d \xrightarrow{\tau} g_i(d,e_1)$ and $d \xrightarrow{a_j(f_j(d,e_2))} g_j(d,e_2)$ for some $e_1, e_2 \in E$ (i.e., both $h_i(d,e_1)$ and $h_j(d,e_2)$ are true). We want to try and complete the picture of these two transitions of d in a confluent fashion; cf. Section 6.5. In principle, one could try all possible ways to complete this picture with transitions $g_i(d,e_1) \xrightarrow{a_j(f_j(g_i(d,e_1),x))} g_j(g_i(d,e_1),x)$

and $g_j(d, e_2) \xrightarrow{\tau} g_i(g_j(d, e_2), y)$ for any $x, y \in E$. However, since we want to be able to derive the formulas resulting from this completion by means of an automated theorem prover, we avoid the existential quantification introduced by the x and the y, and only attempt the completion for $x = e_2$ and $y = e_1$. This choice is motivated by the fact that confluence in system behaviour is usually caused by two different concurrent components within a distributed system that can perform independent transitions. Independence of these transitions implies that $x = e_2$ and $y = e_1$.



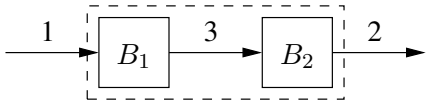
Thus we obtain the picture above. In case the transitions $d \xrightarrow{\tau} g_i(d, e_1)$ and $d \xrightarrow{a_j(f_j(d,e_2))} g_j(d, e_2)$ are distinct, we want to prove the presence of two more transitions (denoted by dashed arrows in the picture): $g_i(d, e_1) \xrightarrow{a_j(f_j(g_i(d,e_1),e_2))} g_j(g_i(d, e_1), e_2)$ and $g_j(d, e_2) \xrightarrow{\tau} g_i(g_j(d, e_2), e_1)$ with $f_j(d, e_2) = f_j(g_i(d, e_1), e_2)$ and $g_j(g_i(d, e_1), e_2) = g_i(g_j(d, e_2), e_1)$. These requirements are captured by the following formula.

Let $a_i = \tau$. If for all summands j and $e_1, e_2 \in E$, $h_i(d, e_1) \wedge h_j(d, e_2)$ implies

$$\begin{array}{lll} & h_j(g_i(d, e_1), e_2) & \\ \wedge\ & h_i(g_j(d, e_2), e_1) & \bigvee \quad a_j = \tau \\ \wedge\ & f_j(d, e_2) = f_j(g_i(d, e_1), e_2) & \quad \wedge\ g_i(d, e_1) = g_j(d, e_2) \\ \wedge\ & g_j(g_i(d, e_1), e_2) = g_i(g_j(d, e_2), e_1) & \end{array}$$

then summand i of the LPE is confluent.

Example 41. We consider two unbounded queues in sequence (cf. the example of two one-bit buffers in sequence in Section 4.3).



Action $r_i(d)$ represents reading datum d from channel i, while action $s_i(d)$ represents sending datum d into channel i. Let Δ denote the data domain. The two unbounded queues are defined by the process declaration

$$B_1(\lambda{:}List) = \sum_{d:\Delta} r_1(d)\cdot B_1(in(d,\lambda)) + s_3(toe(\lambda))\cdot B_1(untoe(\lambda)) \lhd nonempty(\lambda) \rhd \delta$$

$$B_2(\lambda{:}List) = \sum_{d:\Delta} r_3(d)\cdot B_2(in(d,\lambda)) + s_2(toe(\lambda))\cdot B_2(untoe(\lambda)) \lhd nonempty(\lambda) \rhd \delta$$

The initial state of the system is

$$\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1([]) \parallel B_2([])))$$

$toe(\lambda)$ denotes the bottom element of the list λ and $untoe(\lambda)$ is the list that results if the bottom element of λ is removed, and $nonempty(\lambda)$ tests whether λ is non-empty (cf. Exercise 4).

sort $List$
func $[] :\to List$
$in : \Delta \times List \to List$
map $nonempty : List \to Bool$
$if : Bool \times List \times List \to List$
$toe : List \to \Delta$
$untoe : List \to List$
var $d : \Delta$
$\lambda, \lambda_1, \lambda_2 : List$
rew $nonempty([]) = \mathtt{F}$
$nonempty(in(d,\lambda)) = \mathtt{T}$
$if(\mathtt{T}, \lambda_1, \lambda_2) = \lambda_1$
$if(\mathtt{F}, \lambda_1, \lambda_2) = \lambda_2$
$toe(in(d,\lambda)) = if(nonempty(\lambda), toe(\lambda), d)$
$untoe(in(d,\lambda)) = if(nonempty(\lambda), in(d, untoe(\lambda)), [])$

The external behaviour of these two unbounded queues in sequence is again an unbounded queue. Clearly, the resulting state space is infinite.

The process term $\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1(\lambda_1) \parallel B_2(\lambda_2)))$ is by means of linearisation (see Section 6.2) transformed into the LPE

$$\begin{aligned} X(\lambda_1{:}List, \lambda_2{:}List) = {} & \sum_{d:\Delta} r_1(d)\cdot X(in(d,\lambda_1),\lambda_2) \\ & + \tau\cdot X(untoe(\lambda_1), in(toe(\lambda_1),\lambda_2)) \lhd nonempty(\lambda_1) \rhd \delta \\ & + s_2(toe(\lambda_2))\cdot X(\lambda_1, untoe(\lambda_2)) \lhd nonempty(\lambda_2) \rhd \delta \end{aligned}$$

We compute the confluence formulas.

- Commutation of τ and $r_1(d)$:

$$\begin{aligned} nonempty(\lambda_1) \Rightarrow {} & nonempty(in(d,\lambda_1)) \\ & \wedge\ in(d, untoe(\lambda_1)) = untoe(in(d,\lambda_1)) \\ & \wedge\ in(toe(\lambda_1),\lambda_2) = in(toe(in(d,\lambda_1)),\lambda_2) \end{aligned}$$

- Commutation of τ and $s_2(toe(\lambda_2))$:

$$\begin{aligned} & nonempty(\lambda_1) \wedge nonempty(\lambda_2) \\ \Rightarrow\ & nonempty(in(toe(\lambda_1), \lambda_2)) \wedge nonempty(\lambda_1) \\ \wedge\ & toe(in(toe(\lambda_1), \lambda_2)) = toe(\lambda_2) \\ \wedge\ & untoe(in(toe(\lambda_1), \lambda_2)) = in(toe(\lambda_1), untoe(\lambda_2)) \end{aligned}$$

It is not difficult to see that these confluence formulas can be derived from the equational specification of the data type *List* above (Exercise 93). Hence, the τ-summand of the LPE X is confluent.

We note that it would be more involved to derive the confluence formulas if we had used the standard definitions of *toe* and *untoe*:

$$\begin{aligned} & toe(in(d, [])) = d \\ & toe(in(d, in(e, \lambda))) = toe(in(e, \lambda)) \\ & untoe(in(d, [])) = [] \\ & untoe(in(d, in(e, \lambda))) = in(d, untoe(in(e, \lambda))) \end{aligned}$$

Namely, in that case the formulas

$$\begin{aligned} & nonempty(\lambda) \Rightarrow toe(in(d, \lambda)) = toe(\lambda) \\ & nonempty(\lambda) \Rightarrow untoe(in(d, \lambda)) = in(d, untoe(\lambda)) \end{aligned}$$

would need to be derived.

A theorem prover within the μCRL toolset [94], for boolean combinations over a user-defined algebraic data type, makes it possible to prove such (in general large) confluence formulas in an automated fashion. The theorem prover is based on an extension of so-called ordered binary decision diagrams (see Section 7.7), in which each node is labelled with an equality between data terms [57]. The theorem prover is not complete, as equalities over an abstract data type are in general undecidable. In case the theorem prover cannot prove validity of a formula, diagnostics are provided; the user can then add equations to the data specification. In some cases, the formula is not valid in all states of the system, but does hold in all reachable states. The user may supply an invariant $\mathcal{I}$ (see Section 7.2), and confluence formulas can be proved under the assumption $\mathcal{I}(d)$. Such an invariant must be proved separately, which can again be done using the theorem prover.

If a τ-summand in a convergent LPE (i.e., an LPE that does not give rise to any infinite sequence of τ-transitions, see Section 7.1) is confluent, then it can be given priority over the other summands. So the negation of the condition of the confluent τ-summand can be added as a conjunct to the conditions of the other summands in the LPE. The state spaces belonging to the original and the resulting LPE are branching bisimilar. Convergence of the LPE is essential, as in Section 6.5 it was shown that in the presence of τ-loops, a priorisation of confluent τ's may be unsound.

Example 42. The LPE $X(\lambda_1, \lambda_2)$ in Example 41 is convergent. Namely, with each τ-transition, the list λ_1 is made shorter. As a result of τ-priorisation, the negation of the condition of the τ-summand (i.e., $empty(\lambda_1)$) is added as a conjunct to the conditions of the other summands:

$$\begin{aligned} & X(\lambda_1{:}List, \lambda_2{:}List) \\ = {} & \textstyle\sum_{d:\Delta} r_1(d) \cdot X(in(d, \lambda_1), \lambda_2) \lhd empty(\lambda_1) \rhd \delta \\ + {} & \tau \cdot X(untoe(\lambda_1), in(toe(\lambda_1), \lambda_2)) \lhd nonempty(\lambda_1) \rhd \delta \\ + {} & s_2(toe(\lambda_2)) \cdot X(\lambda_1, untoe(\lambda_2)) \lhd empty(\lambda_1) \wedge nonempty(\lambda_2) \rhd \delta \end{aligned}$$

So if the list λ_1 is non-empty, then only the τ-summand can be executed.

State Space Generation Modulo Confluence

Blom and van de Pol [23] showed how confluence can be exploited during state space generation from an LPE.

First, assume a finite state space, and a set of confluent τ-transitions. For each reachable state s, we compute a *representative* state $repr(s)$, such that:

- if $s \xrightarrow{\tau} s'$ is confluent, then $repr(s) = repr(s')$; and
- each state s can evolve to $repr(s)$ by confluent τ-transitions.

To compute the representative of a state, a depth-first search traversal via the confluent τ-transitions is made, until a state with a known representative is encountered, or a 'terminal' strongly connected component of confluent τ-transitions is found. (Terminal means that for each state s in the strongly connected component and for each confluent transition $s \xrightarrow{\tau} s'$, s' is in this same strongly connected component.) In the former case the known representative is returned, and in the latter case the state where the terminal strongly connected component was entered is returned.

Next, assume an LPE with one or more confluent τ-summands, which gives rise to a finite state space. Let d_0 be the initial state. In the state space generation algorithm from [23], only representatives of states are generated. State space generation is started from $repr(d_0)$. Moreover, if a state $repr(d)$ is added to the state space, then for each transition $d \xrightarrow{a} d'$ we add the state $repr(d')$ and the transition $repr(d) \xrightarrow{a} repr(d')$ to the generated state space (except for transitions with $a = \tau$ and $repr(d) = repr(d')$).

The state spaces generated by the 'standard' algorithm (see Section 6.3) and by the partial order reduction technique described above are branching bisimilar. Namely, a state and its representative are branching bisimilar. Moreover, in the 'standard' state space, for each transition $d \xrightarrow{a} d'$ there is transition $repr(d) \xrightarrow{a} d''$ with $repr(d') = repr(d'')$.

This exploitation of confluence within an LPE may lead to the generation of a state space that is several orders of magnitudes smaller compared to the

standard state space algorithm (see Section 6.3). In the table below, which is taken from [23], the number of states and transitions are given for the state spaces generated with and without taking confluence into account, for μCRL specifications of the ABP and the BRP with a data domain of two elements, and for the asynchronous version of the TIP with networks of 10, 12 and 14 nodes.

	standard state space		reduced state space	
system	states	transitions	states	transitions
ABP(2)	97	122	29	54
BRP(2)	1,952	2,387	1,420	1,855
TIP(10)	72,020	389,460	6,171	22,668
TIP(12)	446,648	2,853,960	27,219	123,888
TIP(14)	2,416,632	17,605,592	105,122	544,483

7.6 Elimination of Parameters and Sum Variables

The μCRL toolset comprises four static analysis tools (`rewr`, `constelm`, `parelm` and `sumelm`) [52] that target the automated simplification of LPEs by eliminating data parameters that do not influence the LPE's behaviour.

At first sight the underlying algorithms may seem somewhat simplistic, because in principle a sensible μCRL specification will be free of data parameters that are inert or that remain constant throughout any execution sequence. However, keep in mind that LPEs are usually produced by means of linearisation algorithms (see Section 6.2), where such side-effects can be quite easily introduced. It turns out that the tools described in this section are remarkably successful at simplifying the LPEs belonging to existing μCRL specifications. In some cases these simplifications lead to a substantial reduction in the size of the corresponding state space.

We proceed to explain the simplification algorithms with respect to an LPE

$$X(x_1{:}D_1, \ldots, x_m{:}D_m) = \sum_{i=1}^{k} \sum_{y_1^i : E_1^i} \cdots \sum_{y_{n_i}^i : E_{n_i}^i} a_i(f_i(\mathbf{x}, \mathbf{y^i})) \cdot X(g_1^i(\mathbf{x}, \mathbf{y^i}), \ldots, g_m^i(\mathbf{x}, \mathbf{y^i})) \lhd h_i(\mathbf{x}, \mathbf{y^i}) \rhd \delta$$

Here, we spell out the data parameters $x_1, \ldots, x_m$ (abbreviated to $\mathbf{x}$) of the LPE and the sum variables $y_1^i, \ldots, y_{n_i}^i$ (abbreviated to $\mathbf{y^i}$) of the ith summand, because they are referred to in the algorithms. The initial value of x_j is assumed to be the datum d_j of sort D_j, for $j = 1, \ldots, m$.

The tools `constelm` and `sumelm` assume the presence of equality functions eq for data types (see Section 2.3).

Rewriting of Data Terms

The data terms occurring in the LPE can be rewritten using the equations of the data types (see Section 2.2). If a condition in the LPE is rewritten to `F`, then the corresponding summand in the LPE is removed. This tool is called `rewr`.

Elimination of Constant Data Parameters

A data parameter x_j for $j = 1, \ldots, m$ of the LPE can be replaced by its initial value if it can be determined that this parameter remains constant throughout any execution of the process. This elimination of constant data parameters may shorten the time needed to generate a state space from the LPE. Furthermore, this elimination can make other simplification tools more effective.

The algorithm underlying `constelm` works as follows. Throughout the algorithm, some data parameters are marked and some are not, where unmarked parameters are candidates for elimination, while marked parameters are no longer candidates. Initially, all data parameters in the LPE are unmarked.

We check whether, if all unmarked parameters are replaced by their initial values, the conditions of the LPE guarantee that these initial values are all preserved. To be more precise, for each unmarked $j = 1, \ldots, m$ and each summand $i = 1, \ldots, k$, it is checked whether

$$\Big(\bigwedge_{j \text{ unmarked}} eq(x_j, d_j) \wedge h_i(\mathbf{x}, \mathbf{y^i}) \Big) \Rightarrow eq(g^i_j(\mathbf{x}, \mathbf{y^i}), d_j)$$

rewrites to `T` (using `rewr`). If this check fails for some unmarked j and some i, then j is marked. Namely, then it cannot be guaranteed that the value of x_j remains d_j throughout any execution.

The above check is repeated until it succeeds for all unmarked parameters and for all summands. After each failure, the check must be repeated for all unmarked parameters. When the check succeeds, we conclude that the unmarked parameters remain at their initial values throughout any execution of the process. This means that all occurrences of unmarked parameters x_j at the right-hand side of the LPE can be replaced by their initial values d_j, and that unmarked parameters can be eliminated from the parameter list of the LPE.

`constelm` preserves bisimilarity (see Section 3.10), meaning that the original and the resulting LPE give rise to bisimilar state spaces. The worst-case time complexity of the algorithm is $O(m^2k)$, where m is the number of data parameters and k the number of summands in the original LPE.

Elimination of Inert Data Parameters

A data parameter x_j for $j = 1, \ldots, m$ of the LPE that has no (direct or indirect) influence on the data parameters of actions and on conditions, does not

influence the LPE's behaviour and can be removed. Elimination of inert data parameters may lead to a substantial reduction of the state space underlying the LPE. If the inert parameter ranges over an infinite data domain, the number of states can even reduce from infinite to finite.

The algorithm underlying `parelm` works as follows. Throughout the algorithm, some data parameters of the LPE are marked and some are not, where unmarked parameters are candidates for elimination, while marked parameters are no longer candidates. Initially, all data parameters of the LPE are unmarked. A parameter is marked if it occurs at one of the following three kinds of places in the LPE:

1. in a condition $h_i(\mathbf{x}, \mathbf{y^i})$ for some $i \in \{1, \ldots, k\}$;
2. in an argument $f_i(\mathbf{x}, \mathbf{y^i})$ for some $i \in \{1, \ldots, k\}$; or
3. in an argument $g^i_j(\mathbf{x}, \mathbf{y^i})$ for some $i \in \{1, \ldots, k\}$ and some marked $j \in \{1, \ldots, m\}$.

In the first two cases, the data parameter under scrutiny has a direct influence on the LPE's behaviour. In the third case, it influences the value of data parameter x_j, which in turn was found to influence the LPE's behaviour.

When no further data parameter can be marked, we conclude that the unmarked parameters are inert. This means that all unmarked parameters can be eliminated from the data parameter list of the LPE. Since at the right-hand side of the LPE these unmarked parameters only occur in arguments $g^i_j(\mathbf{x}, \mathbf{y^i})$ where the jth data parameter is unmarked, this elimination effectively removes all occurrences of unmarked parameters from the LPE.

`parelm` preserves bisimilarity. The worst-case time complexity of the algorithm is $O(mk)$.

Elimination of Constant and Inert Sum Variables

If a sum variable is inert, meaning that it does not influence the system's behaviour, then it can be eliminated. Furthermore, if the range of values of a sum variable is restricted to only one datum, then the sum variable is basically a constant, so that it can be replaced by this datum. In both cases, the corresponding summation sign can be eliminated.

The tool `sumelm` consists of three parts. First, if a sum variable only occurs below the summation sign where it is declared (i.e., it does not occur in the process term within the scope of this summation sign), then this sum variable is clearly inert, so that it can be eliminated. Note that this simplification works best after application of `parelm` and `rewr`, to guarantee that occurrences of sum variables in inert data parameters and obsolete conditions have been eliminated beforehand. For example, only after application of `parelm` to the LPE in Exercise 96, the inert sum variable w can be eliminated.

Second, if the data type of a sum variable consists of a single element, then clearly this sum variable can be replaced by this element.

The third and most interesting aspect of `sumelm` is that it tries to detect for each condition $h_i(\mathbf{x}, \mathbf{y^i})$ whether it restricts the range of sum variables y^i_j for $j = 1, \ldots, n_i$ to a single value. If this is the case, then the occurrences of y^i_j in the ith summand can be replaced by this value. We proceed to explain the algorithm behind this part of `sumelm`.

For each boolean condition b, the set $\mathit{Values}(y, b)$ consists of data terms that do not contain y and that are guaranteed to be equal to y if b holds:

$$\begin{aligned}
&\mathit{Values}(y, \mathit{eq}(y, d)) = \{d\} \text{ if } y \text{ does not occur in } d\\
&\mathit{Values}(y, \mathit{eq}(d, y)) = \{d\} \text{ if } y \text{ does not occur in } d\\
&\mathit{Values}(y, b_1 \wedge b_2) \;=\; \mathit{Values}(y, b_1) \cup \mathit{Values}(y, b_2)\\
&\mathit{Values}(y, b_1 \vee b_2) \;=\; \mathit{Values}(y, b_1) \cap \mathit{Values}(y, b_2)\\
&\mathit{Values}(y, b) \qquad = \emptyset \text{ otherwise}
\end{aligned}$$

Conditions $\mathit{eq}(y, d)$ and $\mathit{eq}(d, y)$ imply that y is equal to the data term d (see Section 2.3). If both b_1 and b_2 hold, then y is equal to the data terms that occur in $\mathit{Values}(y, b_1)$ or in $\mathit{Values}(y, b_2)$. If b_1 or b_2 holds, then y is guaranteed to be equal to the data terms that occur both in $\mathit{Values}(y, b_1)$ and in $\mathit{Values}(y, b_2)$. In order to calculate $\mathit{Values}(y, b_1 \vee b_2)$, `rewr` is applied to expressions $\mathit{eq}(d, e)$ with $d \in \mathit{Values}(y, b_1)$ and $e \in \mathit{Values}(y, b_2)$, to determine which data terms in these two sets are equal.

If $d \in \mathit{Values}(y^i_j, h_i(\mathbf{x}, \mathbf{y^i}))$ for some $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, n_i\}$, then the condition $h_i(\mathbf{x}, \mathbf{y^i})$ implies that the value of y^i_j is equal to d. So all occurrences of y^i_j in $f_i(\mathbf{x}, \mathbf{y^i})$ and $g^i_\ell(\mathbf{x}, \mathbf{y^i})$, for $\ell = 1, \ldots, m$, can be replaced by d. Since y^i_j does not occur in the data term d, the summation sign for y^i_j can then be eliminated from the ith summand of the LPE.

`sumelm` preserves bisimilarity. The worst-case time complexity of the algorithm is $O((n_1 + \cdots + n_k)k)$.

Example 43. We assume the data types *Bool*, *Bit* consisting of $\{0, 1\}$, and D consisting of $\{d_1, d_2\}$. The three data types are supplied with an equality function *eq*. Consider the LPE

$$\begin{aligned}
X(d{:}D, b{:}\mathit{Bit}) = {} & \textstyle\sum_{d':D} a{\cdot}X(d', b) \lhd \mathit{eq}(d, d_2) \vee \mathit{eq}(b, 0) \rhd \delta\\
& + \textstyle\sum_{b':\mathit{Bit}} c{\cdot}X(d, b') \lhd \mathit{eq}(b', 0) \rhd \delta
\end{aligned}$$

The initial state is $X(d_1, 0)$.

- We apply `sumelm` to the LPE. Owing to the condition $\mathit{eq}(b', 0)$ in the second summand, the sum variable b' is replaced by 0 and the summation sign for b' is removed. The resulting LPE is

$$\begin{aligned}
X(d{:}D, b{:}\mathit{Bit}) = {} & \textstyle\sum_{d':D} a{\cdot}X(d', b) \lhd \mathit{eq}(d, d_2) \vee \mathit{eq}(b, 0) \rhd \delta\\
& + c{\cdot}X(d, 0) \lhd \mathit{eq}(0, 0) \rhd \delta
\end{aligned}$$

- `parelm` still does not change the LPE, because the data parameters d and b both occur in a condition.
  We apply `constelm` to the LPE. It marks the data parameter d (due to the occurrence of d' in the first argument of X), but leaves the data parameter b unmarked. So the latter parameter is replaced by its initial value 0 and removed from the data parameter list of X. The resulting LPE is

$$\begin{aligned} X(d{:}D) = \textstyle\sum_{d':D} a{\cdot}X(d') \lhd eq(d, d_2) \vee eq(0,0) \rhd \delta \\ + c{\cdot}X(d) \lhd eq(0,0) \rhd \delta \end{aligned}$$

- We apply `rewr` to the LPE. The resulting LPE is

$$X(d{:}D) = \textstyle\sum_{d':D} a{\cdot}X(d') \lhd \mathtt{T} \rhd \delta \;+\; c{\cdot}X(d) \lhd \mathtt{T} \rhd \delta$$

- We apply `parelm` to the LPE. Since the data parameter d occurs neither in conditions nor in arguments of actions, it is found to be inert. The resulting LPE is

$$X = \textstyle\sum_{d':D} a{\cdot}X \lhd \mathtt{T} \rhd \delta \;+\; c{\cdot}X \lhd \mathtt{T} \rhd \delta$$

- Finally, we apply `sumelm` to the LPE once again. The sum variable d' is found to be inert. The resulting LPE is

$$X = a{\cdot}X \lhd \mathtt{T} \rhd \delta \;+\; c{\cdot}X \lhd \mathtt{T} \rhd \delta$$

7.7 Symbolic Model Checking

In this section, some methods are sketched to perform model checking of μ-calculus formulas on a symbolic representation of a state space.

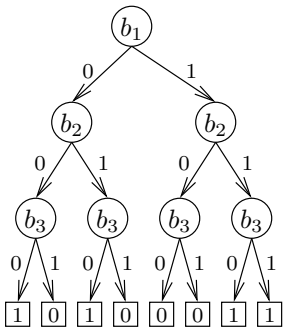
Ordered Binary Decision Diagrams

Suppose the number of states in a state space does not exceed 2^n, for some $n > 0$. Then each state can be represented uniquely by a sequence of bits of length n. (In this context it is usance to write true as the bit 1 and false as the bit 0.) As a result, the transitions in the state space can be captured by mappings $\varphi_a : Bool^{2n} \to Bool$ for $a \in \mathsf{Act} \cup \{\tau\}$, where $\varphi_a(b_1, \ldots, b_n, b'_1, \ldots, b'_n) = 1$ if and only if $(b_1, \ldots, b_n) \xrightarrow{a} (b'_1, \ldots, b'_n)$ is a transition in the state space. To handle large state spaces, we are interested in compressed representations of the mappings φ_a; this observation is due to McMillan [84].

For a start, we note that a mapping $\varphi : Bool^m \to Bool$ can be represented as a binary decision tree of depth m. Each node in the binary decision tree at depth $i \in \{0, \ldots, m-1\}$ is associated with the $(i+1)$th argument of φ, and has two outgoing edges to nodes at depth $i+1$; one edge is labelled 0 and the other is labelled 1, capturing the two possible values of the boolean

variable b_{i+1}. The leaves (at depth m) carry a label from *Bool*, such that a path from the root of the binary decision tree over edges labelled with the boolean values $b_1, \ldots, b_m$, respectively, always leads to a leaf that carries the label $\varphi(b_1, \ldots, b_m)$.

Example 44. The boolean formula $(b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_3)$, with the boolean variable b_i taken as the ith argument of this formula, results in the binary decision tree



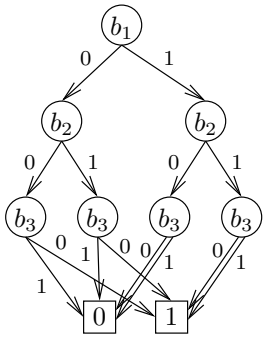
(For the sake of clarity, non-leaves are depicted as circles, while leaves are depicted as boxes.)

An *ordered binary decision diagram* (OBDD), introduced by Bryant [28], is obtained by collapsing nodes in a binary decision tree. First of all, the leaves labelled 0 are collapsed, and likewise for the leaves labelled 1. Next, two minimisation steps are performed, until neither of them can be applied anymore:

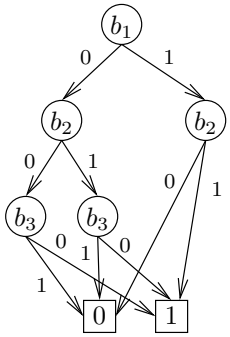
- If for two distinct non-leaves, associated with the same boolean variable, their 0-transitions lead to the same node, and their 1-transitions too, then they can be collapsed.
- If the 0-transition and the 1-transition of a non-leaf ν both lead to the same node ν', then ν can be eliminated, and all its incoming edges are redirected to ν'.

The two minimisation steps above are repeated until neither of them can be applied anymore. The outcome is independent of the order in which nodes are collapsed and eliminated. Checking equivalence of two mappings from $Bool^m$ to *Bool* boils down to checking isomorphism of the corresponding OBDDs.

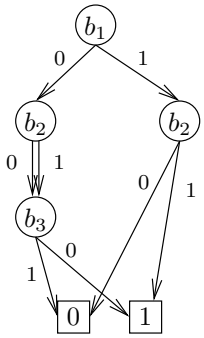
Example 45. We transform the binary decision tree from Example 44 into an OBDD. First, leaves with the same label are collapsed.



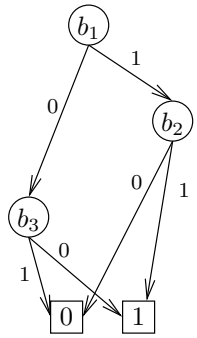
Next, we perform two reduction steps at once: the two rightmost nodes associated with b_3 can be eliminated, because for both nodes, their 0- and 1-transition lead to the same node.



Next, the two remaining nodes associated with b_3 can be collapsed, because their 0-transitions lead to the same node, and likewise for their 1-transitions.



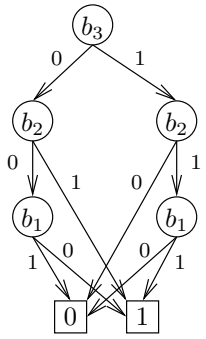
Finally, the leftmost node associated with b_2 can be eliminated, because its 0- and 1-transition lead to the same node. The result is called a *reduced* OBDD.



In general, an OBDD is called *reduced* if no minimisation step can be applied to the OBDD anymore. Each order of applying minimisation steps to a binary decision tree leads to the same reduced OBDD.

The adjective 'ordered' in OBDD refers to the fact that implicitly an ordering $b_1, \ldots, b_m$ on boolean variables was selected, where b_1 is the smallest and b_m the largest. A different ordering on variables usually produces a very different OBDD.

Example 46. We consider the boolean formula from Example 45 again, but for the reverse ordering $b_3 < b_2 < b_1$. The reduced OBDD for $(b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_3)$ then becomes



The chosen ordering can have a huge impact on the number of nodes in the resulting reduced OBDD. Assume a family of mappings $\varphi_m : \mathit{Bool}^m \to \mathit{Bool}$ for $m > 0$. While one ordering on the boolean arguments $b_1, \ldots, b_m$ could produce reduced OBDDs that grow in a polynomial fashion (with respect to m), another ordering could produce reduced OBDDs that grow exponentially. For some families, the reduced OBDDs grow in an exponential fashion for any ordering on boolean variables. Given a mapping $\varphi : \mathit{Bool}^m \to \mathit{Bool}$, finding an ordering on its variables to obtain a reduced OBDD of minimal size is an NP-complete problem [29], which suggests that this problem cannot always be solved in polynomial time.

Given two OBDDs $\mathcal{O}$ and $\hat{\mathcal{O}}$ over the same boolean variables $b_1, \ldots, b_m$, the OBDDs for $\neg\mathcal{O}$, $\mathcal{O} \wedge \hat{\mathcal{O}}$, $\mathcal{O} \vee \hat{\mathcal{O}}$, $\exists b\, \mathcal{O}$ and $\forall b\, \mathcal{O}$ (with $b \in \{b_1, \ldots, b_m\}$) can

be computed efficiently [28]. The resulting graph is actually in general not yet an OBDD; as a final step, the aforementioned minimisation procedure will have to be applied to obtain an OBDD.

The OBDD for $\neg\mathcal{O}$ is obtained by simply inverting the labels 0 and 1 of the leaves in the OBDD $\mathcal{O}$. The OBDD for $\mathcal{O} \wedge \hat{\mathcal{O}}$ is computed recursively as follows. We distinguish four cases.

1. Suppose the roots of $\mathcal{O}$ and $\hat{\mathcal{O}}$ are associated with the same boolean variable b. Let the 0-transitions for the roots of $\mathcal{O}$ and $\hat{\mathcal{O}}$ lead to the OBDDs $\mathcal{O}'$ and $\hat{\mathcal{O}}'$, respectively. Moreover, let their 1-transitions lead to the OBDDs $\mathcal{O}''$ and $\hat{\mathcal{O}}''$, respectively.
   The root of the OBDD for $\mathcal{O} \wedge \hat{\mathcal{O}}$ is associated with b. Furthermore, the 0- and 1-transition from this root lead to the OBDDs for $\mathcal{O}' \wedge \hat{\mathcal{O}}'$ and $\mathcal{O}'' \wedge \hat{\mathcal{O}}''$, respectively, which are computed recursively.

2. Suppose the root of $\mathcal{O}$ is associated with a boolean variable b, while either $\hat{\mathcal{O}}$ is a single leaf or the root of $\hat{\mathcal{O}}$ is associated with a boolean variable $b' > b$. Let the 0- and 1-transition for the root of $\mathcal{O}$ lead to the OBDDs $\mathcal{O}'$ and $\mathcal{O}''$, respectively.
   The root of the OBDD for $\mathcal{O} \wedge \hat{\mathcal{O}}$ is associated with b. Furthermore, the 0- and 1-transition from this root lead to the OBDDs for $\mathcal{O}' \wedge \hat{\mathcal{O}}$ and $\mathcal{O}'' \wedge \hat{\mathcal{O}}$, respectively, which are computed recursively.

3. The case where the root of $\hat{\mathcal{O}}$ is associated with a boolean variable b, while either $\mathcal{O}$ is a single leaf or its root is associated with a boolean variable $b' > b$ is treated similar to the previous case.

4. Finally, suppose both $\mathcal{O}$ and $\hat{\mathcal{O}}$ consist of a single leaf. If both leaves are labelled 1, then the OBDD for $\mathcal{O} \wedge \hat{\mathcal{O}}$ is a leaf labelled 1. Otherwise, it is a leaf labelled 0.

The OBDD for $\mathcal{O} \vee \hat{\mathcal{O}}$ is constructed likewise. The computations basically only differ for the case where both $\mathcal{O}$ and $\hat{\mathcal{O}}$ are leaves. Computing the OBDD for $\mathcal{O} \wedge \hat{\mathcal{O}}$ or $\mathcal{O} \vee \hat{\mathcal{O}}$ takes $O(k \cdot \ell)$, where k and ℓ denote the number of nodes in $\mathcal{O}$ and $\hat{\mathcal{O}}$, respectively.

The OBDD for $\exists b\, \mathcal{O}$ is obtained as follows.

- Compute the OBDD $\mathcal{O}'$ that is obtained from $\mathcal{O}$ by taking the value of b to be 0. That is, each node ν in $\mathcal{O}$ associated with b is eliminated, and the incoming edges of ν are redirected to the node that results after taking the 0-transition from ν.

- Compute the OBDD $\mathcal{O}''$ that is obtained from $\mathcal{O}$ by taking the value of b to be 1. That is, each node ν in $\mathcal{O}$ associated with b is eliminated, and the incoming edges of ν are redirected to the node that results after taking the 1-transition from ν.

- Finally, compute the OBDD for $\mathcal{O}' \vee \mathcal{O}''$.

The OBDD for $\forall b\,\mathcal{O}$ is constructed likewise, except that in the last step the OBDD for $\mathcal{O}' \wedge \mathcal{O}''$ is computed.

Symbolic Model Checking of OBDDs

In Section 6.6, the μ-calculus was presented, together with an algorithm for computing which states in a state space satisfy a given μ-calculus formula. And at the start of Section 7.7 it was explained how a state space with at most 2^n states can be captured by OBDDs $\mathcal{O}_a : \mathit{Bool}^{2n} \to \mathit{Bool}$ for $a \in \mathsf{Act} \cup \{\tau\}$, where $\mathcal{O}_a(b_1, \ldots, b_n, b'_1, \ldots, b'_n) = 1$ if and only if $(b_1, \ldots, b_n) \xrightarrow{a} (b'_1, \ldots, b'_n)$ is a transition in the state space.

We explain how the state explosion problem can be circumvented by computing, for a given state space in the form of OBDDs over $b_1, \ldots, b_n, b'_1, \ldots, b'_n$, and a μ-calculus formula ϕ, an OBDD $\mathcal{B}(\phi)$ over $b_1, \ldots, b_n$ representing the states that satisfy ϕ (cf. [37, Section 7.4]). In other words, $\mathcal{B}(\phi)$ maps the states that satisfy ϕ to 1, and all other states to 0. The OBDD $\mathcal{B}(\phi)$ is defined inductively on the structure of ϕ, using the algorithms for computing negations, conjunctions and existential quantifications of OBDDs (see Section 7.7). Let *TRUE* and *FALSE* denote the OBDD (over $b_1, \ldots, b_n$) consisting of a single leaf with the label 1 and 0, respectively.

$$
\begin{aligned}
\mathcal{B}(\mathtt{T}) &= \mathit{TRUE} \\
\mathcal{B}(\mathtt{F}) &= \mathit{FALSE} \\
\mathcal{B}(\phi \wedge \phi') &= \mathcal{B}(\phi) \wedge \mathcal{B}(\phi') \\
\mathcal{B}(\phi \vee \phi') &= \mathcal{B}(\phi) \vee \mathcal{B}(\phi') \\
\mathcal{B}(\langle a \rangle \phi) &= \exists b'_1 \cdots \exists b'_n \, (\mathcal{O}_a(b_1, \ldots, b_n, b'_1, \ldots, b'_n) \wedge \mathcal{B}'(\phi)) \\
\mathcal{B}([a]\phi) &= \forall b'_1 \cdots \forall b'_n \, (\neg \mathcal{O}_a(b_1, \ldots, b_n, b'_1, \ldots, b'_n) \vee \mathcal{B}'(\phi)) \\
\mathcal{B}(\mu X.\phi) &= \mathit{FIX}(\phi, X{=}\mathit{FALSE}) \\
\mathcal{B}(\nu X.\phi) &= \mathit{FIX}(\phi, X{=}\mathit{TRUE})
\end{aligned}
$$

At the right-hand side of the fifth and sixth case, $\mathcal{B}'(\phi)$ is an OBDD over $b'_1, \ldots, b'_n$ (which is calculated against OBDDs $\mathcal{O}_a(b'_1, \ldots, b'_n, b''_1, \ldots, b''_n)$ for $a \in \mathsf{Act} \cup \{\tau\}$).

In the last two cases, the construct $\mathit{FIX}(\phi, X{=}\mathcal{O})$ operates as follows, for $\mathcal{O}$ an OBDD over $b_1, \ldots, b_n$ (cf. the construct $\mathit{FIX}(\phi, X{=}S)$, with S a set of states, in the model checking algorithm for the μ-calculus in Section 6.6). $\mathit{FIX}(\phi, X{=}\mathcal{O})$ basically computes the OBDD $\mathcal{B}(\phi)$ over $b_1, \ldots, b_n$. As said, this computation is by induction over the structure of ϕ; in this induction, the recursion variable X is simply replaced by the OBDD $\mathcal{O}$. When this computation has completed, the OBDD $\mathit{FIX}(\phi, X{=}\mathit{FIX}(\phi, X{=}\mathcal{O}))$ over $b_1, \ldots, b_n$ is computed, et cetera, until a fixpoint is reached (i.e., until $\mathit{FIX}(\phi, X{=}\mathcal{O}')$ equals $\mathcal{O}'$, for some intermediate result $\mathcal{O}'$).

Boolean Equation Systems

An alternative way to symbolically model check a μ-calculus formula against a state space is by means of a *boolean equation system* [3]. This consists of an ordered sequence of boolean equations of the form $\mu x = b$ and $\nu x = b$, where μ and ν denote the minimal and maximal fixpoint operators, x a boolean variable, and b a boolean formula built from $\mathtt{T}$, $\mathtt{F}$, boolean variables, conjunction and disjunction. Given a μ-calculus formula ϕ and a state space, a boolean equation system can be produced (where each state in the state space corresponds to a boolean variable in the boolean equation system), such that a state satisfies ϕ if and only if the boolean variable corresponding to this state has the solution $\mathtt{T}$ in the boolean equation system. We refer to, e.g., [82] for a definition of the meaning of a boolean equation system, and for the transformation from a μ-calculus formula ϕ and a state space to a boolean equation system. Also, [82] presents some heuristics to try and solve boolean equation systems.

A *parametrised* boolean equation system [53] consists of an ordered sequence of boolean equations of the form $\mu x(d_1{:}D_1, \ldots, d_n{:}D_n) = b$ and $\nu x(d_1{:}D_1, \ldots, d_n{:}D_n) = b$, where the boolean formula b can contain occurrences of the data parameters $d_1, \ldots, d_n$. Given a μ-calculus formula ϕ and an LPE, a parametrised boolean equation system can be produced, such that the question whether a state of the LPE satisfies ϕ can be verified at the level of the parametrised boolean equation system. In [65] some heuristics are presented to try and solve parametrised boolean equation systems.

Exercises

Exercise 83. Prove with the help of CL-RSP that the process term $X(m) \parallel X(n)$ in Example 20 is equal to $Y(m, n)$, for $m, n \in Nat$.

Exercise 84. Show that the mappings $\mathcal{I}_1$ and $\mathcal{I}_2$ in Example 37 are invariants.

Exercise 85. Show that the mappings $\mathcal{I}_1(d) = \mathtt{T}$ for all $d \in D$ and $\mathcal{I}_2(d) = \mathtt{F}$ for all $d \in D$ are invariants for all LPEs.

Exercise 86. Consider the LPE

$$Y(n{:}Nat) \;=\; \sum_{m:Nat} a{\cdot}Y(2m \dot{-} n) \lhd m + n < 5 \rhd \delta$$

(with $\dot{-}$ the 'cut-off minus', see Exercise 2). Give the 'optimal' invariant $\mathcal{I}$ for this LPE with $\mathcal{I}(0) = \mathtt{T}$. The same for $\mathcal{I}(1) = \mathtt{T}$. (Here, optimal means that $\mathcal{I}(n) = \mathtt{F}$ for as many $n \in Nat$ as possible.)

Exercise 87. Let $div3 : Nat \to Bool$ map natural numbers to $\mathtt{T}$ if and only if they can be divided by three. Consider the process declarations

$$X(n{:}Nat) = a(even(n))\cdot X(S(S(S(n)))) \lhd div3(n) \rhd \delta$$

and

$$\begin{aligned} Y &= a(\mathtt{T})\cdot Z \\ Z &= a(\mathtt{F})\cdot Y \end{aligned}$$

Prove, using CL-RSP with invariants, that $X(n) = Y$ for natural numbers n that can be divided by six.

Exercise 88. Let

$$\begin{aligned} X(n{:}Nat) = &\textstyle\sum_{m:Nat}(a\cdot X(S(n)) \lhd n = 3m \rhd \delta \\ &+ c\cdot X(S(n)) \lhd n = S(3m) \rhd \delta \\ &+ \tau\cdot X(S(n)) \lhd n = S(S(3m)) \rhd \delta) \\ Y(b{:}Bool) = &\ a\cdot Y(\mathtt{F}) \lhd b \rhd \delta + c\cdot Y(\mathtt{T}) \lhd \neg b \rhd \delta \end{aligned}$$

Verify that X is convergent. What is the focus condition for X? Give a state mapping $\phi : Nat \to Bool$ such that the matching criteria are satisfied with respect to X and Y. Prove that the matching criteria are satisfied indeed.

Exercise 89. The data type *Nil* consists of the single element *nil*. Let

$$\begin{aligned} X(n{:}Nat) = &\textstyle\sum_{m:Nat}(a\cdot X(S(S(n))) \lhd n = 3m \rhd \delta \\ &+ c\cdot X(S(n)) \lhd n = S(3m) \rhd \delta \\ &+ \tau\cdot X(S(n)) \lhd n = S(S(3m)) \rhd \delta) \\ Y(\eta{:}Nil) = &\ a\cdot Y(\eta) \lhd \mathtt{T} \rhd \delta \end{aligned}$$

Verify that X is convergent. What is the focus condition for X? Give an invariant $\mathcal{I} : Nat \to Bool$ with $\mathcal{I}(0) = \mathtt{T}$ such that the matching criteria are satisfied with respect to X and Y for all $n \in Nat$ with $\mathcal{I}(n) = \mathtt{T}$. (Of course the state mapping $\phi : Nat \to Nil$ is defined by $\phi(n) = nil$ for all $n \in Nat$.)

Exercise 90. Derive the linearisation of the synchronous version of the TIP in Section 7.4:

$$\tau_I(\partial_H(X(i_0, p[i_0], s[i_0]) \| \cdots \| X(i_k, p[i_k], s[i_k]))) = Y(p, s)$$

for pairs (p, s).

Exercise 91. Prove, as is claimed in the verification of the synchronous version of the TIP, that $\mathcal{I}_4$ is an invariant for Y.

Exercise 92. Derive the linearisation of the two unbounded queues in sequence in Example 41:

$$\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1(\lambda_1) \parallel B_2(\lambda_2))) = X(\lambda_1, \lambda_2)$$

Exercise 93. Derive the two confluence formulas in Example 41 from the equational specification for lists that is given in that example.

Exercise 94. Consider the following LPEs. In each case, check for which τ-summands the confluence formulas are true.

1. $X(n{:}Nat) = \tau{\cdot}X(n) \lhd even(n) \rhd \delta \;+\; a(n){\cdot}X(S(S(n))) \lhd \mathtt{T} \rhd \delta$
2. $X(n{:}Nat) = \tau{\cdot}X(S(n)) \lhd even(n) \rhd \delta \;+\; a(n){\cdot}X(S(S(n))) \lhd \mathtt{T} \rhd \delta$
3. $X(n{:}Nat) = \tau{\cdot}X(S(n)) \lhd even(n) \rhd \delta \;+\; a{\cdot}X(S(S(n))) \lhd \mathtt{T} \rhd \delta$
4. $X(n{:}Nat) = \tau{\cdot}X(S(n)) \lhd \mathtt{T} \rhd \delta \;+\; a{\cdot}X(S(S(n))) \lhd even(n) \rhd \delta$
5. $X(n{:}Nat) = \tau{\cdot}X(S(n)) \lhd even(n) \rhd \delta \;+\; \tau{\cdot}X(S(n)) \lhd \neg even(n) \rhd \delta$
   $\qquad\qquad + \;\; a{\cdot}X(S(S(n))) \lhd even(n) \rhd \delta$

Exercise 95. Apply `constelm` to the LPE

$$X(w{:}Nat, x{:}Nat, y{:}Nat, z{:}Nat) = a(x){\cdot}X(x,w,z,y) \lhd eq(y,0) \rhd \delta \\ + \; b(y){\cdot}X(S(0),x,0,plus(y,z))$$

with initial state $X(0,0,0,0)$. Which data parameters are found to be constant? What is the resulting LPE?

Exercise 96. Apply `parelm` to the LPE

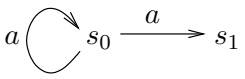
$$X(x{:}D, y{:}D, z{:}D) \;=\; a{\cdot}X(x,z,y) \;+\; \sum_{w:D} b(z){\cdot}X(w,y,z)$$

Apply `sumelm` to the resulting LPE.

Exercise 97. Compute the reduced OBDD for $b \wedge \neg b$.

Exercise 98. Compute the reduced OBDD for $(b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_2)$ with $b_1 < b_2$.

Exercise 99. Consider the state space



1. Calculate the reduced OBDD $\mathcal{O}_a$ representing the a-transitions in this state space.
2. Calculate using the symbolic model checking algorithm which states satisfy $\mu X.([a]X)$ and which states satisfy $\nu X.([a]X)$.

A

The μCRL Toolset in a Nutshell

The μCRL [18, 106], LTSmin [22] and CADP [48] toolsets support the analysis and manipulation of μCRL specifications and the state spaces generated from it. A μCRL specification can be automatically transformed into an LPE, which is stored in a binary format or as a plain text file, using ATerms. All other tools in the μCRL toolset use LPEs as their starting point. The simulator allows an interactive simulation of an LPE. There are a number of tools that allow optimisations on the level of LPEs. And theorem proving and symbolic model checking techniques can be applied to an LPE. The instantiator generates a state space from an LPE (under the condition that it is finite), and the resulting state space can be visualised, minimised and model checked.

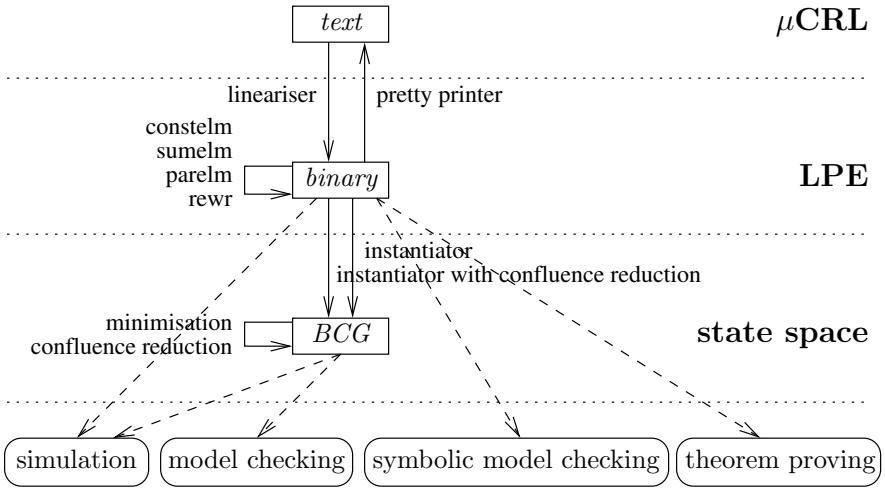


Fig. A.1. The main components of the μCRL and CADP toolsets

An overview of the μCRL toolset is presented in Fig. A.1. This picture is divided into four layers: μCRL specifications, LPEs, state spaces and analysis methods. The rectangular boxes denote different ways to represent instances of the corresponding layer (for example, LPEs can be represented in a binary or a textual form). A solid arrow denotes a transformation from one instance to another that is supported by the μCRL toolset; keywords are provided to these arrows to give some information on which kinds of transformations are involved. Finally, the oval boxes represent several ways to analyse systems, and dashed arrows show how the different representations of LPEs and state spaces can be analysed. The box named BCG and its outgoing dashed arrows belong to the CADP toolset.

The toolset expects a μCRL specification in plain ASCII. The notation deviates in certain ways from the LaTeX typesetting that is used in the main text. An overview of the μCRL language that is accepted as input for the toolset can be found in [59] and in the toolset manual [106, Chapter2].

We will now explained how some of the more basic tools of μCRL and CADP can be used. More information can be found at the μCRL homepage (`www.cwi.nl/~mcrl/`) and at the CADP homepage (`www.inrialpes.fr/vasy/cadp`).

Linearisation

The μCRL toolset is built around a special form of μCRL specifications, called an LPE (see Definition 4). The tool `mcrl` determines whether a specification is correct μCRL, and if so, transforms this specification into an LPE. Suppose you have written a μCRL specification of the TIP, that is stored in a text file called `tip`, and that adheres to the syntactic restrictions of parallel pCRL (see Section 6.2). This specification can be transformed into an LPE by the following instruction:

```
mcrl -tbfile -regular tip
```

This instruction produces a file `tip.tbf`, which contains an LPE. The `-tbfile` option takes care that the resulting LPE is in so-called 'toolbus format'. The option `-regular` determines that the regular linearisation algorithm is used to produce an LPE. (An alternative is `-regular2`, which calls yet another linearisation algorithm; see [106, Section 3.1].) In case the flag `-regular` is omitted, the default linearisation algorithm is used (see Section 6.2). The binary file `tip.tbf` can be pretty printed by means of

```
pp tip.tbf
```

Be warned that the μCRL toolset cannot cope with successful termination of processes. It assumes that either the process proceeds for an infinite amount of time (as is the case for the ABP, the BRP and the SWP), or ends in a deadlock (as is the case for the TIP).

Simulator

A simulator `msim`, which allows one to step through (the state space underlying) the LPE stored in the file `tip.tbf`, is started with the instruction

```
msim tip.tbf
```

A dialogue screen will appear, with a number of buttons. Click on `Start` to initiate a simulation. In the `Menu display` screen the actions will appear that can be performed in the initial state. Clicking on such an actions results in a transition to the resulting state. By clicking on the button `State`, a screen will appear with a description of the state space. The button `Term` produces a screen that takes as input a data term, and outputs the value of this data term in the current state.

Simulating a state space containing τ's tends to be quite useless, because the hiding operator eliminates information on the internal structure of a μCRL specification. Therefore, before using the simulator, it is recommended to temporarily remove the hiding operator from the **init** description of the initial state of the μCRL specification.

State Space Generation

From the file `tip.tbf`, a file `tip.aut`, containing the corresponding state space, is generated (see Section 6.3) with the instruction

```
instantiator -i tip.tbf
```

The `-i` option takes care that the internal action τ is represented in a form that is interpreted as the hidden action by CADP (namely as an i).

Visualisation

The state space stored in `tip.aut` file can be visualised using CADP. After starting

```
xeuca
```

a screen will appear, with at the left an overview of the files in the directory in which `xeuca` was started. Click with the left mouse button on the file `tip.aut`. A pop-up menu will appear. First click on `Visualize` and then on `Draw`. An image of the state space will appear. Furthermore, the state space can be simulated with the option `Execute`.

Minimisation

The state space stored in `tip.aut` can be minimised (see Section 6.4) with the tool `BCG_MIN`. In `xeuca`, click on `tip.aut` once again. This time choose the option `Reduce`. A pop-up menu will appear, entitled `Reduction of tip.aut`. Set the flags of the following options:

1. `Reduce using BCG_MIN`
2. `Branching Equivalence`

Start the reduction by clicking on `OK`. A file `tip_b.bcg` is produced, which will appear at the leftmost upper corner of the overview of the files in the directory. This file can be visualised as before.

Model Checking

A temporal formula in the regular alternation-free μ-calculus can be model checked (see Section 6.6) as follows. Click on the file `tip.aut`, and choose the option `Verify temporal formulas`. A pop-up menu will appear, entitled `Verify temporal formulas in tip.aut`. This pop-up window requires, under `Use Open/Caesar's Evaluator`, an input file with a name ending on `.mcl`, containing a regular alternation-free μ-calculus formula. The `.mcl` files in the directory will be given as options (under `Files`). Choose one, and click on `OK` (with the flag `Use Open/Caesar's Evaluator` set, and the option `Generation of Diagnostic Files` on; these flags are set by default). At the right-hand side the result of the verification is given: either `TRUE`, or `FALSE` together with an error trace that violates the property.

For example, a file `leader.mcl` could contain the formula

```
[(not "leader")*] mu X. (<true> true and [not "leader"] X)
```

expressing that always eventually a leader will be elected. (Beware that between quotes, actions are interpreted exactly as they are written; for instance, `"send(d,e)"` and `"send(d, e)"` are interpreted as different actions, because the second action contains a space behind the comma.)

Hints for Debugging

If a μCRL specification cannot be parsed, or no LPE or state space can be generated from it, or the generated state space does not meet your expectations, in general it is of no use to devote large amounts of time studying the specification itself. Instead you could try to analyse the behaviour of the μCRL specification. We describe two techniques that can help in doing so.

- By adding an extra dummy summand to a process declaration, featuring a special `echo` action, information can be obtained about the values of the data parameters in a state. The `echo` action carries as data parameter the state, and after executing this action the state remains unchanged. That is, a process declaration $X(d{:}D) = p$ can be changed into

$$X(d{:}D) = p + \texttt{echo}(d) \cdot X(d)$$

- The instantiator, which generates a state space from an LPE, covers all possible combinations of input data, which can cause state explosion. By adding an explicit environment, with a specific scenario of input data, a considerable reduction of the state space can be obtained. For example, in case of the patient support system described in Section 5.5, such a scenario could define a sequence of inputs into the user console. In the **comm** section, the send actions in the scenario can be declared to communicate with the corresponding read actions in the μCRL specification of the system under consideration. And in the **init** section, the scenario itself is put in parallel with the system specification.

Exercises

Exercise 100. 1. Specify in μCRL two one-bit buffers that are put in sequence (see Section 4.3).

2. Generate, using `mcrl -tbfile -regular`, a `.tbf` file (with as data domain $\Delta = \{d_1, d_2\}$).
3. Pretty print, using `pp`, the μCRL specification that belongs to the `.tbf` file that you generated.
4. Simulate the state space underlying the `.tbf` file using `msim`. (Remove temporarily the hiding operator from the **init** section.)
5. Generate the corresponding state space using the μCRL toolset using `instantiator -i`.
6. Visualise the state space using `xeuca`.
7. Formulate in the regular μ-calculus the property 'each occurrence of the action $r_1(d_1)$ is eventually followed by an occurrence of the action $s_2(d_1)$'. Use the model checker of CADP to verify this temporal formula.
8. Minimise the state space modulo branching bisimulation using `BCG_MIN`, and visualise the result.
9. Analyse whether this state space corresponds to the external behaviour of a two-bit buffer. Correct your μCRL specification if necessary.
10. Specify in μCRL a system consisting of two two-bit buffers that are put in sequence, and apply the μCRL and CADP toolsets once again.

B

Solutions to Exercises

1 **map** $\wedge, \Rightarrow, \Leftrightarrow$: $Bool \times Bool \to Bool$
$\neg : Bool \to Bool$
var x:$Bool$
rew $x \vee \mathtt{T} = \mathtt{T}$
$x \vee \mathtt{F} = x$
$\neg \mathtt{T} = \mathtt{F}$
$\neg \mathtt{F} = \mathtt{T}$
$\mathtt{T} \Rightarrow x = x$
$\mathtt{F} \Rightarrow x = \mathtt{T}$
$\mathtt{T} \Leftrightarrow x = x$
$\mathtt{F} \Leftrightarrow x = \neg x$

2 **map** $\geq, >$: $Nat \times Nat \to Bool$
$power, \dot{-}$: $Nat \times Nat \to Nat$
$even : Nat \to Bool$
var m, n:Nat
rew $n \geq 0 = \mathtt{T}$
$0 \geq S(n) = \mathtt{F}$
$S(n) \geq S(m) = n \geq m$
$0 > n = \mathtt{F}$
$S(n) > 0 = \mathtt{T}$
$S(n) > S(m) = n > m$
$power(m, 0) = S(0)$
$power(m, S(n)) = mul(power(m, n), m)$
$0 \dot{-} n = 0$
$n \dot{-} 0 = n$
$S(n) \dot{-} S(m) = n \dot{-} m$
$even(0) = \mathtt{T}$
$even(S(n)) = \neg even(n)$

3 **var** m, n:Nat
$divides(0, n) = \mathtt{F}$
$divides(S(m), 0) = \mathtt{T}$
$divides(S(m), S(n)) = (n \geq m) \wedge divides(S(m), n \dot{-} m)$

4 **var** d, e:D, λ, λ':$List$

rew $head(in(d, \lambda)) = d$
$toe(in(d, []) = d$
$toe(in(d, in(e, \lambda))) = toe(in(e, \lambda))$
$tail(in(d, \lambda)) = \lambda$
$untoe(in(d, [])) = []$
$untoe(in(d, in(e, \lambda))) = in(d, untoe(in(e, \lambda)))$
$++([], \lambda) = \lambda$
$++(in(d, \lambda), \lambda') = in(d, ++(\lambda, \lambda'))$
$append(d, []) = in(d, [])$
$append(d, in(e, \lambda)) = in(e, append(d, \lambda))$
$nonempty([]) = \mathtt{F}$
$nonempty(in(d, \lambda)) = \mathtt{T}$
$length([]) = 0$
$length(in(d, \lambda)) = S(length(\lambda))$

5 **var** $d, e{:}D$
$\lambda, \lambda'{:}List$
rew $eq([], []) = \mathtt{T}$
$eq(in(d, \lambda), []) = \mathtt{F}$
$eq([], in(d, \lambda)) = \mathtt{F}$
$eq(in(d, \lambda), in(e, \lambda')) = eq(d, e) \wedge eq(\lambda, \lambda')$

6 The help function $if : Bool \times Set \times Set \to Set$ acts as an if-then-else construct: $if(\mathtt{T}, \sigma, \sigma') = \sigma$ and $if(\mathtt{F}, \sigma, \sigma') = \sigma'$.

1. **var** $d, e{:}D$
$\sigma, \sigma'{:}Set$
$element(d, \textit{empty-set}) = \mathtt{F}$
$element(d, in(e, \sigma)) = if(eq(d, e), \mathtt{T}, element(d, \sigma))$
$remove(d, \textit{empty-set}) = \textit{empty-set}$
$remove(d, in(e, \sigma)) = if(eq(d, e), remove(d, \sigma), in(e, remove(d, \sigma)))$
$eq(\textit{empty-set}, \textit{empty-set}) = \mathtt{T}$
$eq(\textit{empty-set}, in(d, \sigma)) = \mathtt{F}$
$eq(in(d, \sigma), \sigma') = if(element(d, \sigma'), eq(remove(d, \sigma), remove(d, \sigma')), \mathtt{F})$
2. Replace the second equation of *remove* by
$$remove(d, in(e, \sigma)) = if(eq(d, e), \sigma, in(e, remove(d, \sigma)))$$
so that $remove(d, \sigma)$ removes only a single occurrence of d from σ (if present). Moreover, replace the second equation of *eq* by
$$eq(in(d, \sigma), \sigma') = if(element(d, \sigma'), eq(\sigma, remove(d, \sigma')), \mathtt{F})$$

7 **var** $d, e{:}D,\ \lambda, \lambda'{:}List$
rew $add(d, []) = in(d, [])$
$add(d, in(e, \lambda)) = if(d < e, in(d, in(e, \lambda)), in(e, add(d, \lambda)))$
$if(\mathtt{T}, \lambda, \lambda') = \lambda$
$if(\mathtt{F}, \lambda, \lambda') = \lambda'$

8 $\mathtt{T} \vee \mathtt{T} = \mathtt{T}$ and $\mathtt{F} \vee \mathtt{F} = \mathtt{F}$.
$\neg\neg\mathtt{T} = \neg\mathtt{F} = \mathtt{T}$ and $\neg\neg\mathtt{F} = \neg\mathtt{T} = \mathtt{F}$.

9 1. $\mathtt{F} \wedge \mathtt{T} = \mathtt{F}$.
$\mathtt{F} \wedge \mathtt{F} = \mathtt{F}$.
2. $\mathtt{T} \Rightarrow \mathtt{T} = \mathtt{T}$.
$\mathtt{F} \Rightarrow \mathtt{T} = \mathtt{T}$.

3. $\mathtt{T} \Rightarrow \mathtt{F} = \mathtt{F} = \neg\mathtt{T}$.
   $\mathtt{F} \Rightarrow \mathtt{F} = \mathtt{T} = \neg\mathtt{F}$.
4. $\mathtt{T} \Rightarrow b_2 = b_2 = \neg\neg b_2 = \neg b_2 \Rightarrow \mathtt{F} = \neg b_2 \Rightarrow \neg\mathtt{T}$.
   $\mathtt{F} \Rightarrow b_2 = \mathtt{T} = \neg b_2 \Rightarrow \mathtt{T} = \neg b_2 \Rightarrow \neg\mathtt{F}$.
5. $\mathtt{T} \Leftrightarrow \mathtt{T} = \mathtt{T}$.
   $\mathtt{F} \Leftrightarrow \mathtt{T} = \neg\mathtt{T} = \mathtt{F}$.
6. $\mathtt{T} \Leftrightarrow \mathtt{T} = \mathtt{T}$.
   $\mathtt{F} \Leftrightarrow \mathtt{F} = \neg\mathtt{F} = \mathtt{T}$.
7. $\mathtt{T} \Leftrightarrow \neg b_2 = \neg b_2 = \neg(\mathtt{T} \Leftrightarrow b_2)$.
   $\mathtt{F} \Leftrightarrow \neg b_2 = \neg\neg b_2 = \neg(\mathtt{F} \Leftrightarrow b_2)$.
8. $(b_1 \vee \mathtt{T}) \Leftrightarrow b_1 = \mathtt{T} \Leftrightarrow b_1 = b_1 = b_1 \vee \mathtt{F} = b_1 \vee \neg\mathtt{T}$.
   $(b_1 \vee \mathtt{F}) \Leftrightarrow b_1 = b_1 \Leftrightarrow b_1 = \mathtt{T} = b_1 \vee \mathtt{T} = b_1 \vee \neg\mathtt{F}$.
9. $even(plus(k, 0)) = even(k) = even(k) \Leftrightarrow \mathtt{T} = even(k) \Leftrightarrow even(0)$.
   $even(plus(k, S(\ell))) = even(S(plus(k, \ell))) = \neg even(plus(k, \ell)) =$
   $\neg(even(k) \Leftrightarrow even(\ell)) = even(k) \Leftrightarrow \neg even(\ell) = even(k) \Leftrightarrow even(S(\ell))$.
10. $even(mul(k, 0)) = even(0) = \mathtt{T} = even(k) \vee \mathtt{T} = even(k) \vee even(0)$.
    $even(mul(k, S(\ell))) = even(plus(mul(k, \ell), k)) = even(mul(k, \ell)) \Leftrightarrow even(k) =$
    $(even(k) \vee even(\ell)) \Leftrightarrow even(k) = even(k) \vee \neg even(\ell) = even(k) \vee even(S(\ell))$.

10 1. $plus(0, 0) = 0$.
   $plus(0, S(k)) = S(plus(0, k)) = S(k)$.
2. $mul(0, 0) = 0$.
   $mul(0, S(k)) = plus(mul(0, k), 0) = mul(0, k) = 0$.
3. $plus(plus(k, \ell), 0) = plus(k, \ell) = plus(k, plus(\ell, 0))$.
   $plus(plus(k, \ell), S(m)) == S(plus(plus(k, \ell), m)) = S(plus(k, plus(\ell, m))) =$
   $plus(k, S(plus(\ell, m))) = plus(k, plus(\ell, S(m)))$.
4. $mul(k, plus(\ell, 0)) = mul(k, \ell) = plus(mul(k, \ell), 0) = plus(mul(k, \ell), mul(k, 0))$.
   $mul(k, plus(\ell, S(m))) = mul(k, S(plus(\ell, m))) = plus(mul(k, plus(\ell, m)), k) =$
   $plus(plus(mul(k, \ell), mul(k, m)), k) = plus(mul(k, \ell), plus(mul(k, m), k)) =$
   $plus(mul(k, \ell), mul(k, S(m)))$.
5. $mul(mul(k, \ell), 0) = 0 = mul(k, 0) = mul(k, mul(\ell, 0))$.
   $mul(mul(k, \ell), S(m)) = plus(mul(mul(k, \ell), m), mul(k, \ell)) =$
   $plus(mul(k, mul(\ell, m)), mul(k, \ell)) = mul(k, plus(mul(\ell, m), \ell)) =$
   $mul(k, mul(\ell, S(m)))$.
6. $mul(power(m, k), power(m, 0)) = mul(power(m, k), S(0)) =$
   $plus(mul(power(m, k), 0), power(m, k)) = plus(0, power(m, k)) =$
   $power(m, k) = power(m, plus(k, 0))$.
   $mul(power(m, k), power(m, S(\ell))) = mul(power(m, k), mul(power(m, \ell), m)) =$
   $mul(mul(power(m, k), power(m, \ell)), m) = mul(power(m, plus(k, \ell)), m) =$
   $power(m, S(plus(k, \ell))) = power(m, plus(k, S(\ell)))$.

11 $++(untoe(\lambda), in(toe(\lambda), \lambda'))$.
Base case: $++(untoe(in(d, [])), in(toe(in(d, [])), \lambda')) = ++([], in(d, \lambda')) = in(d, \lambda') =$
$in(d, ++([], \lambda')) = ++(in(d, []), \lambda')$.
Inductive case:

$$
\begin{aligned}
& ++(untoe(in(d, in(e, \lambda))), in(toe(in(d, in(e, \lambda))), \lambda')) \\
&= ++(in(d, untoe(in(e, \lambda))), in(toe(in(e, \lambda)), \lambda')) \\
&= in(d, ++(untoe(in(e, \lambda)), in(toe(in(e, \lambda)), \lambda'))) \\
&= in(d, ++(in(e, \lambda), \lambda')) && \text{(by induction)} \\
&= ++(in(d, in(e, \lambda)), \lambda').
\end{aligned}
$$

12 $a(d)\cdot(b(\mathit{stop},\mathtt{F})+c)$
$a(d)\cdot b(\mathit{stop},\mathtt{F})+a(d)\cdot c$.

13 1. $((a+a)\cdot(b+b))\cdot(c+c) \overset{A3}{=} (a\cdot(b+b))\cdot(c+c) \overset{A3}{=} (a\cdot b)\cdot(c+c) \overset{A3}{=} (a\cdot b)\cdot c \overset{A5}{=} a\cdot(b\cdot c)$.
2. $(a+a)\cdot(b\cdot c)+(a\cdot b)\cdot(c+c) \overset{A3}{=} a\cdot(b\cdot c)+(a\cdot b)\cdot(c+c) \overset{A5}{=} (a\cdot b)\cdot c+(a\cdot b)\cdot(c+c) \overset{A3}{=} (a\cdot b)\cdot(c+c)+(a\cdot b)\cdot(c+c) \overset{A3}{=} (a\cdot b)\cdot(c+c) \overset{A3}{=} (a\cdot(b+b))\cdot(c+c)$.
3. $((a+b)\cdot c+a\cdot c)\cdot d \overset{A4}{=} ((a\cdot c+b\cdot c)+a\cdot c)\cdot d \overset{A1}{=} ((b\cdot c+a\cdot c)+a\cdot c)\cdot d \overset{A2}{=} (b\cdot c+(a\cdot c+a\cdot c))\cdot d \overset{A3}{=} (b\cdot c+a\cdot c)\cdot d \overset{A4}{=} ((b+a)\cdot c)\cdot d \overset{A5}{=} (b+a)\cdot(c\cdot d)$.

14 Suppose $p \subseteq q$ and $q \subseteq p$. By definition, (1) $p+q=q$ and (2) $q+p=p$. Thus $p \overset{(2)}{=} q+p \overset{A1}{=} p+q \overset{(1)}{=} q$.

15 The crux of this exercise is to show that from A2′ and A3 together one can derive A1.
$x+y \overset{A3}{=} (x+y)+(x+y) \overset{A2'}{=} y+((x+y)+x) \overset{A2'}{=} y+(y+(x+x)) \overset{A2'}{=} ((x+x)+y)+y \overset{A2'}{=} (x+(y+x))+y \overset{A2'}{=} (y+x)+(y+x) \overset{A3}{=} y+x$.

16 $a \parallel (b+c) \overset{CM1}{=} (a \between (b+c)+(b+c) \between a)+a \mid (b+c) \overset{CM9}{=} (a \between (b+c)+(b+c) \between a)+(a \mid b+a \mid c) \overset{CF}{=} (a \between (b+c)+(b+c) \between a)+(b'+c') \overset{CF}{=} (a \between (b+c)+(b+c) \between a)+(b \mid a+c \mid a) \overset{CM8}{=} (a \between (b+c)+(b+c) \between a)+(b+c) \mid a \overset{A1}{=} ((b+c) \between a+a \between (b+c))+(b+c) \mid a \overset{CM1}{=} (b+c) \parallel a$.

17 $(b\cdot a) \parallel a \overset{CM1}{=} ((b\cdot a) \between a+a \between (b\cdot a))+(b\cdot a) \mid a \overset{CM2,3,5}{=} (b\cdot(a \parallel a)+a\cdot(b\cdot a))+(b \mid a)\cdot a \overset{CM1}{=} (b\cdot((a \between a+a \between a)+a \mid a)+a\cdot(b\cdot a))+(b \mid a)\cdot a \overset{CM2,CF'}{=} (b\cdot((a\cdot a+a\cdot a)+\delta)+a\cdot(b\cdot a))+(b \mid a)\cdot a \overset{A3,6}{=} (b\cdot(a\cdot a)+a\cdot(b\cdot a))+(b \mid a)\cdot a \overset{A4,5}{=} (b\cdot a+a\cdot b)\cdot a+(b \mid a)\cdot a \overset{CM2}{=} (b \between a+a \between b)\cdot a+(b \mid a)\cdot a \overset{A4}{=} ((b \between a+a \between b)+b \mid a)\cdot a \overset{CM1}{=} (b \parallel a)\cdot a$.

19

$$
\begin{aligned}
& \partial_{\{a,b\}}((a\cdot b) \parallel (b\cdot a)) \\
\overset{CM1}{=}\ & \partial_{\{a,b\}}(((a\cdot b) \between (b\cdot a)+(b\cdot a) \between (a\cdot b))+(a\cdot b) \mid (b\cdot a)) \\
\overset{CM3,CM7}{=}\ & \partial_{\{a,b\}}(a\cdot(b \parallel (b\cdot a))+b\cdot(a \parallel (a\cdot b))+c\cdot(b \parallel a)) \\
\overset{D1\text{-}4}{=}\ & \delta\cdot\partial_{\{a,b\}}(b \parallel (b\cdot a))+\delta\cdot\partial_{\{a,b\}}(a \parallel (a\cdot b))+c\cdot\partial_{\{a,b\}}(b \parallel a) \\
\overset{A6,7}{=}\ & c\cdot\partial_{\{a,b\}}(b \parallel a) \\
\overset{CM1}{=}\ & c\cdot\partial_{\{a,b\}}(b \between a+a \between b+b \mid a) \\
\overset{CM2,CF}{=}\ & c\cdot\partial_{\{a,b\}}(b\cdot a+a\cdot b+c) \\
\overset{D1\text{-}4}{=}\ & c\cdot(\delta\cdot\partial_{\{a,b\}}(a)+\delta\cdot\partial_{\{a,b\}}(b)+c) \\
\overset{A6,7}{=}\ & c\cdot c.
\end{aligned}
$$

20 $\mathit{send}(d)$, $\mathit{read}(d)$ and $\mathit{comm}(d)$ are abbreviated to $s(d)$, $r(d)$ and $c(d)$, respectively, for $d \in \{0,1\}$, and H denotes $\{s,r\}$.

$$\begin{aligned}
& (s(0)+s(1)) \parallel (r(0)+r(1)) \\
\overset{\text{CM1}}{=}\ & (s(0)+s(1)) \between (r(0)+r(1)) + (r(0)+r(1)) \between (s(0)+s(1)) \\
& + (s(0)+s(1)) \mid (r(0)+r(1)) \\
\overset{\text{CM4,CM8,9}}{=}\ & s(0) \between (r(0)+r(1)) + s(1) \between (r(0)+r(1)) + r(0) \between (s(0)+s(1)) \\
& + r(1) \between (s(0)+s(1)) + s(0) \mid r(0)+s(0) \mid r(1)+s(1) \mid r(0) \\
& + s(1) \mid r(1) \\
\overset{\text{CM2,CF,CF}'}{=}\ & s(0)\cdot(r(0)+r(1)) + s(1)\cdot(r(0)+r(1)) + r(0)\cdot(s(0)+s(1)) \\
& + r(1)\cdot(s(0)+s(1)) + c(0) + \delta + \delta + c(1) \\
\overset{\text{A6}}{=}\ & s(0)\cdot(r(0)+r(1)) + s(1)\cdot(r(0)+r(1)) + r(0)\cdot(s(0)+s(1)) \\
& + r(1)\cdot(s(0)+s(1)) + c(0) + c(1).
\end{aligned}$$

Hence,

$$\begin{aligned}
& \partial_H((s(0)+s(1)) \parallel (r(0)+r(1))) \\
=\ & \partial_H(s(0)(r(0)+r(1)) + s(1)(r(0)+r(1)) + r(0)(s(0)+s(1)) \\
& + r(1)(s(0)+s(1)) + c(0) + c(1)) \\
\overset{\text{D1-4}}{=}\ & \delta\cdot\partial_H(r(0)+r(1)) + \delta\cdot\partial_H(r(0)+r(1)) + \delta\cdot\partial_H(s(0)+s(1)) \\
& + \delta\cdot\partial_H(s(0)+s(1)) + c(0) + c(1) \\
\overset{\text{A6,7}}{=}\ & c(0) + c(1).
\end{aligned}$$

21 Let a and b communicate to c. Then $\partial_{\{a,b\}}(a \parallel b)$ can execute c, while $\partial_{\{a,b\}}(a) \parallel \partial_{\{a,b\}}(b)$ cannot execute any action.

22 $\delta = p + q \overset{\text{A3}}{=} (p+q) + (p+q) \overset{\text{A1,2}}{=} p + (p + (q+q)) \overset{\text{A3}}{=} p + (p+q) = p + \delta \overset{\text{A6}}{=} p$.

23 yes; no; yes; yes; no.

24

$$\begin{aligned}
\partial_{\{tick\}}(Clock(0)) &= \partial_{\{tick\}}(tick\cdot Clock(S(0)) + display(0)\cdot Clock(0)) \\
&= \partial_{\{tick\}}(tick\cdot Clock(S(0))) + \partial_{\{tick\}}(display(0)\cdot Clock(0)) \\
&= \delta\cdot\partial_{\{tick\}}(Clock(S(0))) + display(0)\cdot\partial_{\{tick\}}(Clock(0)) \\
&= display(0)\cdot\partial_{\{tick\}}(Clock(0)).
\end{aligned}$$

25 We add the elements in a list λ_0 of natural numbers.

act $print{:}Nat$
var $n, m{:}Nat$
$\lambda{:}List$
proc $Add\text{-}list(\lambda{:}List, n{:}Nat) =$
$print(n) \lhd eq(\lambda, []) \rhd Add\text{-}list(tail(\lambda), plus(n, head(\lambda)))$
init $Add\text{-}list(\lambda_0, 0)$

26 1. $x \lhd \mathtt{T} \rhd y = x = x + \delta = x \lhd \mathtt{T} \rhd \delta + y \lhd \mathtt{F} \rhd \delta = x \lhd \mathtt{T} \rhd \delta + y \lhd \neg\mathtt{T} \rhd \delta$.
$x \lhd \mathtt{F} \rhd y = y = \delta + y = x \lhd \mathtt{F} \rhd \delta + y \lhd \mathtt{T} \rhd \delta = x \lhd \mathtt{F} \rhd \delta + y \lhd \neg\mathtt{F} \rhd \delta$.

2. $x \lhd \mathtt{T} \vee \mathtt{T} \rhd \delta = x \lhd \mathtt{T} \rhd \delta = x \lhd \mathtt{T} \rhd \delta + x \lhd \mathtt{T} \rhd \delta$.
$x \lhd \mathtt{T} \vee \mathtt{F} \rhd \delta = x \lhd \mathtt{T} \rhd \delta = x \lhd \mathtt{T} \rhd \delta + \delta = x \lhd \mathtt{T} \rhd \delta + x \lhd \mathtt{F} \rhd \delta$.
$x \lhd \mathtt{F} \vee \mathtt{T} \rhd \delta = x \lhd \mathtt{T} \rhd \delta = \delta + x \lhd \mathtt{T} \rhd \delta = x \lhd \mathtt{F} \rhd \delta + x \lhd \mathtt{T} \rhd \delta$.
$x \lhd \mathtt{F} \vee \mathtt{F} \rhd \delta = x \lhd \mathtt{F} \rhd \delta = x \lhd \mathtt{F} \rhd \delta + x \lhd \mathtt{F} \rhd \delta$.

3. if $b = \mathtt{T}$, then by the assumption $(b = \mathtt{T} \Rightarrow x = y)$, $x = y$, and so $x \lhd b \rhd z = y \lhd b \rhd z$.
if $b = \mathtt{F}$, then $x \lhd b \rhd z = z = y \lhd b \rhd z$.

27
var $d{:}D$, $\lambda{:}List$
rew $head(in(d,\lambda)) = d$
$tail(in(d,\lambda)) = \lambda$

proc $Stack(\lambda{:}List) = \sum_{d:D} r(d)\cdot Stack(in(d,\lambda))$
$+\,\delta \lhd eq(\lambda,[]) \rhd s(head(\lambda))\cdot Stack(tail(\lambda))$
init $Stack([])$

proc $Queue(\lambda{:}List) = \sum_{d:D} r(d)\cdot Queue(in(d,\lambda))$
$+(\delta \lhd eq(\lambda,[]) \rhd s(toe(\lambda))\cdot Queue(untoe(\lambda)))$
init $Queue([])$

28 ($\supseteq$) By SUM2,

$$\sum_{b:Bool} x \lhd b \rhd y = \sum_{b:Bool} x \lhd b \rhd y + x \lhd \mathtt{T} \rhd y + x \lhd \mathtt{F} \rhd y = \sum_{b:Bool} x \lhd b \rhd y + x + y.$$

So $x + y \subseteq \sum_{b:Bool} x \lhd b \rhd y$.

($\subseteq$) $x \lhd \mathtt{T} \rhd y = x \subseteq x + y$ and $x \lhd \mathtt{F} \rhd y = y \subseteq x + y$. So by induction on the booleans, $x \lhd b \rhd y \subseteq x+y$. Then by SUM8, SUM3 and SUM1, $\sum_{b:Bool} x \lhd b \rhd y \subseteq x + y$.

29 ($\subseteq$) By SUM2,

$$\sum_{d:D} x \lhd b(d) \rhd \delta = \sum_{d:D} x \lhd b(d) \rhd \delta + x \lhd b(e) \rhd \delta = \sum_{d:D} x \lhd b(d) \rhd \delta + x.$$

So $x \subseteq \sum_{d:D} x \lhd b(d) \rhd \delta$.

($\supseteq$) If $b(d) = \mathtt{T}$ then $x \lhd b(d) \rhd \delta = x$, and if $b(d) = \mathtt{F}$ then $x \lhd b(d) \rhd \delta = \delta \subseteq x$, so $x \lhd b(d) \rhd \delta \subseteq x$ for all $d{:}D$. So by induction on the booleans, $x \lhd b(d) \rhd \delta \subseteq x$. Then by SUM8, SUM3 and SUM1, $\sum_{d:D} x \lhd b(d) \rhd \delta \subseteq x$.

30 Let D denote $\{d_1, d_2\}$.

act $in, out{:}D$
proc $X(n{:}Nat, m{:}Nat) = in(d_1)\cdot X(S(n), m) + in(d_2)\cdot X(n, S(m))$
$+\,(out(d_1)\cdot X(n \dot{-} S(0), m) \lhd n > 0 \rhd \delta)$
$+\,(out(d_2)\cdot X(n, m \dot{-} S(0)) \lhd m > 0 \rhd \delta)$
init $X(0,0)$

31
1. yes: $(b+c)\cdot a + b\cdot a + c\cdot a \,\mathcal{B}\, b\cdot a + c\cdot a$ and $a \,\mathcal{B}\, a$.
2. no.
3. yes: $(a+a)\cdot(b\cdot c) + (a\cdot b)\cdot(c+c) \,\mathcal{B}\, (a\cdot(b+b))\cdot(c+c)$, $b\cdot c \,\mathcal{B}\, (b+b)\cdot(c+c)$, $b\cdot(c+c) \,\mathcal{B}\, (b+b)\cdot(c+c)$, $c \,\mathcal{B}\, c+c$, and $c + c \,\mathcal{B}\, c + c$.

33 Base case: $a \xrightarrow{a} \surd$, while aa cannot terminate successfully by the execution of an a-transition. Hence, $a \not\leftrightarrows aa$.
Inductive case: $a^{k+1} \xrightarrow{a} a^k$ is the only transition of a^{k+1}, while $a^{k+2} \xrightarrow{a} a^{k+1}$ is the only transition of a^{k+2}. By induction, a^k and a^{k+1} cannot be related by a bisimulation relation. Hence, $a^{k+1} \not\leftrightarrows a^{k+2}$.

35
1. $a(\tau b + b) \overset{\text{A3}}{=} a(\tau(b+b) + b) \overset{\text{B2}}{=} a(b+b) \overset{\text{A3}}{=} ab$
2. $a(\tau(b+c) + b) \overset{\text{B2}}{=} a(b+c) \overset{\text{A1}}{=} a(c+b) \overset{\text{B2}}{=} a(\tau(c+b) + c) \overset{\text{A1}}{=} a(\tau(b+c) + c)$

3. $\tau_{\{a\}}(a(a(b+c)+b)) \overset{\text{TI1-4}}{=} \tau(\tau(b+c)+b) = \tau(\tau(b+c)+c) \overset{\text{TI1-4}}{=} \tau_{\{d\}}(d(d(b+c)+c))$
4. If $x+y=x$, then $\tau(\tau x+y) = \tau(\tau(x+y)+y) \overset{\text{B2}}{=} \tau(x+y) = \tau x$

36

```
sort Bool
func T,F: -> Bool
map  and,or,eq: Bool # Bool -> Bool
     not: Bool -> Bool
var  x:Bool
rew  and(T,T)=T
     and(F,x)=F
     and(x,F)=F
     or(T,x)=T
     or(x,T)=T
     or(F,F)=F
     not(F)=T
     not(T)=F
     {\it eq}(T,T)=T
     {\it eq}(F,F)=T
     {\it eq}(T,F)=F
     {\it eq}(F,T)=F

sort D
func d1,d2: -> D
map  eq: D # D -> Bool
rew  {\it eq}(d1,d1)=T
     {\it eq}(d2,d2)=T
     {\it eq}(d1,d2)=F
     {\it eq}(d2,d1)=F

act  s2,s3,r1,r3,c3:D

comm s3|r3=c3

proc Buf1 = sum(d:D,r1(d).s3(d).Buf1)
     Buf2 = sum(d:D,r3(d).s2(d).Buf2)

init hide({c3},encap({s3,r3}, Buf2 || Buf1))
```

37

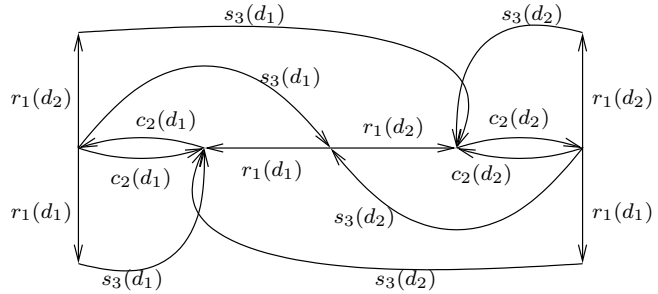
```
Buf2 = rename({r1 -> r3, s3 -> s2},Buf1)
```

38 $a\,\mathcal{B}_1\,a{\cdot}\tau$, $\surd\,\mathcal{B}_1\,\tau$, and $\surd\,\mathcal{B}_1\,\surd$ proves $a \leftrightarrows_b a{\cdot}\tau$.
$a\,\mathcal{B}_2\,\tau{\cdot}a$, $a\,\mathcal{B}_2\,a$, and $\surd\,\mathcal{B}_2\,\surd$ proves $a \leftrightarrows_b \tau{\cdot}a$.
$a{\cdot}\tau\,\mathcal{B}_3\,\tau{\cdot}a$, $a{\cdot}\tau\,\mathcal{B}_3\,a$, $\tau\,\mathcal{B}_3\,\surd$, and $\surd\,\mathcal{B}_3\,\surd$ proves $a{\cdot}\tau \leftrightarrows_b \tau{\cdot}a$.

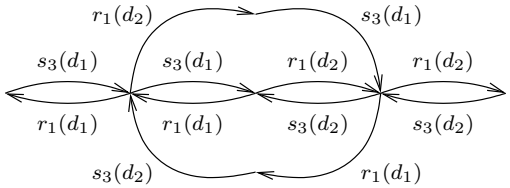
39 $\tau{\cdot}(\tau{\cdot}(a+b)+b)+a\,\mathcal{B}\,a+b$, $\tau{\cdot}(a+b)+b\,\mathcal{B}\,a+b$, $a+b\,\mathcal{B}\,a+b$, and $\surd\,\mathcal{B}\,\surd$.

41 not branching bisimilar; bisimilar; branching bisimilar but not rooted branching bisimilar; rooted branching bisimilar but not bisimilar; not branching bisimilar.

42 1. The node in the middle of the graph below is the initial state.

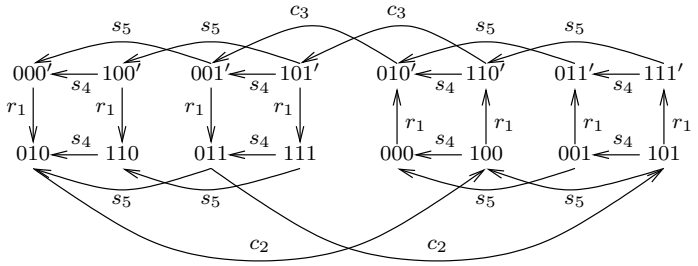


2. yes
3. An execution trace of $\partial_{\{s_3\}}(p)$ to a deadlock state is $r_1(d_1)\, c_2(d_1)\, r_1(d_1)$.
4. The minimised state space is



43 1. **act** $r_1, s_2, r_2, c_2, s_3, r_3, c_3, s_4, s_5{:}\Delta$
comm $s_2 \,|\, r_2 = c_2$
$s_3 \,|\, r_3 = c_3$
proc $X(b{:}Bool) = \sum_{d:\Delta} r_1(d){\cdot}(s_2(d) \lhd eq(b,\mathtt{T}) \rhd s_3(d)){\cdot}X(\neg b)$
$Y = \sum_{d:\Delta} r_2(d){\cdot}s_4(d){\cdot}Y$
$Z = \sum_{d:\Delta} r_3(d){\cdot}s_5(d){\cdot}Z$
init $\partial_{\{s_2,r_2,s_3,r_3\}}(X(\mathtt{T}) \| Y \| Z)$

2. The three bits in the states below denote whether there is a datum in the buffer of Y, X or Z, respectively. In a state with or without prime the next incoming datum (via channel 1) is sent on via channel 2 or 3, respectively. The initial state is $000'$.



50

$$
\begin{aligned}
&\mathit{plusmod}(k,0) &&= k\\
&\mathit{plusmod}(k,S(\ell)) &&= \mathit{succmod}(\mathit{plusmod}(k,\ell))\\
&\mathit{ordered}(k,\ell,m) &&= (k \le \ell \wedge l < m) \vee (m < k \wedge k \le \ell) \vee (\ell < m \wedge m < k)
\end{aligned}
$$

The algebraic formulation reads $\mathit{ordered}(k,\ell,\mathit{plusmod}(k,m))$.

52 Let $d, d', d'' \in \Delta$. An error trace is:

$$\mathbf{(0)}\ r_{\mathrm{A}}(d)\, c_{\mathrm{B}}(d,0)\, j\, c_{\mathrm{C}}(d,0)\, s_{\mathrm{D}}(d)\ \mathbf{(1)}\ r_{\mathrm{A}}(d')\, c_{\mathrm{B}}(d',S(0))\, j\, c_{\mathrm{C}}(d',S(0))\, s_{\mathrm{D}}(d')$$
$$\mathbf{(2)}\ r_{\mathrm{A}}(d)\, c_{\mathrm{B}}(d,0)\, j\, c_{\mathrm{C}}(d,0)\ \mathbf{(3)}\ c_{\mathrm{E}}(S(S(0)))\, j\, c_{\mathrm{F}}(S(S(0)))$$
$$\mathbf{(4)}\ r_{\mathrm{A}}(d'')\, c_{\mathrm{B}}(d'',S(S(0)))\, j\, c_{\mathrm{C}}(d'',S(S(0)))\, s_{\mathrm{D}}(d'')\ \mathbf{(5)}\ s_{\mathrm{D}}(d)$$

(0): initially, sending and receiving windows are $[0, S(0)]$
(1): receiving window becomes $[S(0), S(S(0))]$
(2): receiving window becomes $[S(S(0)), 0]$
(3): d is erroneously stored in the receiving window
(4): sending window becomes $[S(S(0)), 0]$
(5): receiving window becomes $[0, S(0)]$

53

```
X(first-in:Nat,first-empty:Nat,buffer:Buffer,first-in':Nat,
                                                  buffer':Buffer) =
sum(d:Delta,rA(d).X(first-in,succmod(first-empty),
                    in(d,first-empty,buffer),first-in',buffer')
  <| ordered(first-in,first-empty,plusmod(first-in,n)) |> delta)
+ sum(k:Nat,sB(retrieve(k,buffer),k).X(first-in,first-empty,
           buffer,first-in',buffer') <| test(k,buffer) |> delta)
+ sum(k:Nat,rF(k).X(k,first-empty,release(first-in,k,buffer),
                                                first-in',buffer'))
+ sum(d:Delta,sum(k:Nat,rF(d,k).(X(first-in,first-empty,buffer,
                                    first-in',add(d,k,buffer'))
                 <| ordered(first-in',k,plusmod(first-in',n)) |>
              X(first-in,first-empty,buffer,first-in',buffer'))))
+ sA(retrieve(first-in',buffer')).X(first-in,first-empty,buffer,
                   succmod(first-in'),remove(first-in',buffer'))
                            <| test(first-in',buffer') |> delta
+ sB(get-empty(first-in',buffer')).X(first-in,first-empty,
                                      buffer,first-in',buffer')

Y(first-in:Nat,first-empty:Nat,buffer:Buffer,first-in':Nat,
                                                  buffer':Buffer) =
rename({rA->rD,sA->sD,sB->sE,rF->rC},
               X(first-in,first-empty,buffer,first-in',buffer'))

K = sum(d:Delta,sum(k:Nat,rB(d,k).(j.sC(d,k)+j).K))
  + sum(k:Nat,rB(k).(j.sC(k)+j).K)

L = rename({rB->rE,sC->sF},K)
```

54

```
X(first-in:Nat,first-empty:Nat,buffer:Buffer,first-in':Nat,
                                                  buffer':Buffer) =
sum(d:Delta,rA(d).X(first-in,succmod(first-empty),
                    in(d,first-empty,buffer),first-in',buffer')
  <| ordered(first-in,first-empty,plusmod(first-in,n)) |> delta)
+ sum(k:Nat,sB(retrieve(k,buffer),k,get-empty(first-in',
      buffer')).X(first-in,first-empty,buffer,first-in',buffer')
```

```
                                          <| test(k,buffer) |> delta)
  + sum(d:Delta,sum(k:Nat,sum(l:Nat,rF(d,k,l).(X(l,first-empty,
            release(first-in,l,buffer),first-in',add(d,k,buffer'))
                  <| ordered(first-in',k,plusmod(first-in',n)) |>
    X(l,first-empty,release(first-in,l,buffer),first-in',
                                                      buffer')))))
  + sA(retrieve(first-in',buffer')).X(first-in,first-empty,buffer,
                     succmod(first-in'),remove(first-in',buffer'))
                               <| test(first-in',buffer') |> delta
```

55 Each non-inert τ-transition loses the possibility to execute one of the $leader(n)$-transitions at the end.

57 The network is captured by

$$\tau_I(\partial_H(X(i_0,\{i_1,i_2\},0) \parallel X(i_1,\{i_0,i_2\},0) \parallel X(i_2,\{i_0,i_1\},0)))$$

where H consists of all sends and reads of parent requests, and I of all communications of parent requests.
No node is allowed to send a parent request.

59 The process declaration is captured by $Z(\mathtt{T})$, where

$$Z(b{:}Bool) = a{\cdot}Z(\neg b) \lhd b \rhd \delta + c{\cdot}Z(\neg b) \lhd b \rhd \delta + d{\cdot}Z(\neg b) \lhd \neg b \rhd \delta.$$

60

$$\begin{aligned}
&X(\lambda{:}List)\\
&= \textstyle\sum_{m:Nat} a(m){\cdot}X(in(Z,S(m),in(Y,m,tail(\lambda)))) \lhd eq(head(\lambda),(Y,m)) \rhd \delta\\
&+ \textstyle\sum_{m:Nat} b(m){\cdot}X(in(Z,S(m),tail(\lambda))) \lhd eq(head(\lambda),(Z,m)) \rhd \delta\\
&+ \textstyle\sum_{m:Nat} (c(m){\cdot}X(tail(\lambda)) \lhd nonempty(tail(\lambda)) \rhd c(m)) \lhd eq(head(\lambda),(Z,m)) \rhd \delta
\end{aligned}$$

The regular method does not terminate.

61

$$\begin{aligned}
&Y(m{:}Nat) = a(m){\cdot}X(m)\\
&Z(m{:}Nat) = b(m){\cdot}Z(m) + c(S(m))\\
&X(m{:}Nat) = Z(S(m)){\cdot}Y(S(m))\\
\hline
&Y(m{:}Nat) = a(m){\cdot}X(m)\\
&Z(m{:}Nat) = b(m){\cdot}Z(m) + c(S(m))\\
&X(m{:}Nat) = b(S(m)){\cdot}Z(S(m)){\cdot}Y(S(m)) + c(S(S(m))){\cdot}Y(S(m))\\
\hline
&Y(m{:}Nat) = a(m){\cdot}X(m)\\
&Z(m{:}Nat) = b(m){\cdot}Z(m) + c(S(m))\\
&X(m{:}Nat) = b(S(m)){\cdot}X(m) + c(S(S(m))){\cdot}Y(S(m))
\end{aligned}$$

62

$$\begin{aligned}
&Y(n_1{:}Nat,\ldots,n_k{:}Nat) =\\
&\textstyle\sum_{i=1}^{k} a(f(n_i)){\cdot}Y(n_i := g(n_i)) \lhd h(n_i) \rhd \delta\\
&+ \textstyle\sum_{j=1}^{k} b(f'(n_j)){\cdot}Y(n_j := g'(n_j)) \lhd h'(n_j) \rhd \delta\\
&+ \textstyle\sum_{i,j=1}^{k} c(f(n_i)){\cdot}Y(n_i := g(n_i), n_j := g'(n_j))\\
&\qquad\qquad \lhd h(n_i) \wedge h'(n_j) \wedge f(n_i) = f'(n_j) \wedge i \neq j \rhd \delta
\end{aligned}$$

63 Let $k = 2$ and $m = 5$. Define $h_1(d) = 1$, $h_2(d) = 2$, $h_1(d') = 3$, $h_2(d') = 4$, $h_1(d'') = 2$, $h_2(d'') = 3$.
First state d is generated, so that bits one and two are set to 1. Next state d' is generated, so that bits three and four are set to 1. Finally, state d'' is generated. Since bits two and three are already set to 1, the Bloom filter falsely concludes that d'' was already generated before.

64 The minimised state spaces belong to the following process terms and declarations:
1. $(a + \tau{\cdot}b){\cdot}\delta$
2. $(a + b){\cdot}\delta$
3. $(a{\cdot}b{\cdot}c + a{\cdot}b{\cdot}d){\cdot}\delta$
4. $(a + b + \tau{\cdot}b){\cdot}(a + b){\cdot}\delta$
5. $a{\cdot}a{\cdot}a{\cdot}a{\cdot}\delta$
6. $X = (\tau + a){\cdot}Y$
   $Y = b{\cdot}X$
7. $Z = (a + b){\cdot}Z$

65 Assume a state space. For P a set of states, define $s_0 \in \mathit{split}_{\downarrow}(P)$ if there exists a sequence $s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \downarrow$ for some $n \geq 0$ such that $s_i \in P$ for $i = 0, \ldots, n$.
P can be split with respect to $\downarrow$ if $\emptyset \subset \mathit{split}_{\downarrow}(P) \subset P$.

66 all τ-transitions
$\emptyset$
all τ-transitions
$\{s_3 \xrightarrow{\tau} s_5, s_4 \xrightarrow{\tau} s_5\}$

67 $\emptyset$
$\{s_0 \xrightarrow{\tau} s_0, s_1 \xrightarrow{\tau} s_1\}$
$\{s_0 \xrightarrow{\tau} s_1\}$
$\{s_0 \xrightarrow{\tau} s_1\}$

68 s_2 and s_3; s_1 and s_3; s_0, s_2, s_3 and s_4; s_0, s_1 and s_3; s_1, s_2 and s_3; s_1, s_2 and s_3.
For each state, an ACTL formula that is satisfied by this state only can (for example) be composed as a conjunction of an appropriate subset of the six aforementioned ACTL formulas and their negations.

69 1. $\mathtt{E}\,\mathtt{T}\,\mathtt{U}\,(\neg\,(\langle a\rangle\mathtt{T} \vee \langle b\rangle\mathtt{T}))$
2. $\mathtt{EG}\,(\langle a\rangle\mathtt{T} \vee \langle b\rangle\mathtt{T})$

70 s_0, s_2, s_4, s_6, s_7, s_8 and s_{11}; s_1, s_6 and s_8; s_6, s_8 and s_9; s_0 and s_4; s_4; s_3, s_4, s_5, s_7, s_8, s_{10} and s_{11}.

71 yes

72 Implication is not monotonic in its first argument. For example, $\mathtt{F} \Rightarrow \mathtt{F}$ equals $\mathtt{T}$ (i.e., holds in all states), while $\mathtt{T} \Rightarrow \mathtt{F}$ equals $\mathtt{F}$ (i.e., does not hold in any state).

73 1. $X = \{s_0, s_1, s_2\}$
2. $Y = \{s_0, s_1, s_2\}$

74 The subsequent intermediate solutions for X and Y are:

Y	X
$\emptyset$	$\{s_0, s_1, s_2, s_3\}$
$\{s_0, s_1, s_2\}$	$\{s_3\}$
$\{s_1, s_2\}$	$\{s_3\}$

75 $\emptyset$ is the solution for both X and Y.

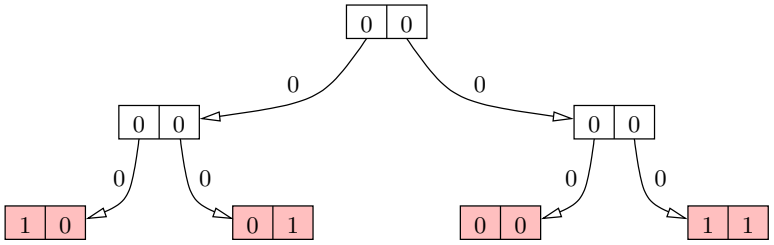
76 1. $[(\neg \mathit{claim}_1)^* \cdot \mathit{release}_1]\, \mathtt{F}$
2. $[\mathtt{T}^* \cdot \mathit{claim}_1]\, \mu X.(\langle \mathtt{T} \rangle\, \mathtt{T} \wedge [\neg \mathit{release}_1]\, X)$
3. $[\mathtt{T}^* \cdot \mathit{claim}_1 \cdot (\neg \mathit{release}_1)^* \cdot \mathit{claim}_2]\, \mathtt{F}$

77 $[\mathtt{T}^* \cdot \mathit{send} \cdot (\neg \mathit{read})^*]\, \langle \mathtt{T}^* \cdot \mathit{read} \rangle\, \mathtt{T}$.

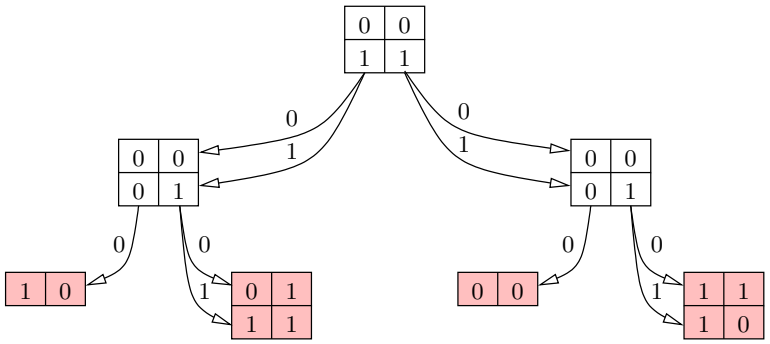
$[\mathtt{T}^* \cdot \mathit{send} \cdot (\neg \mathit{read})^*]$ says: after all traces that perform *send*, and do not perform a *read* afterwards, end up in a state where ...
$\langle \mathtt{T}^* \cdot \mathit{read} \rangle\, \mathtt{T}$ expresses: there exists a trace that contains *read*.
So when *send* has been performed, the option to perform *read* will remain open, as long as *read* has not yet been performed. Fairness then, in case of a finite state space, implies that eventually the action *read* will be performed.

78 $\mu X.(\phi' \vee (\phi \wedge \langle \mathtt{T} \rangle\, X))$
$(\nu X.(\phi \wedge \langle \mathtt{T} \rangle\, X)) \vee (\mu Y.(\phi \wedge ([\mathtt{T}]\, \mathtt{F} \vee \langle \mathtt{T} \rangle\, Y)))$

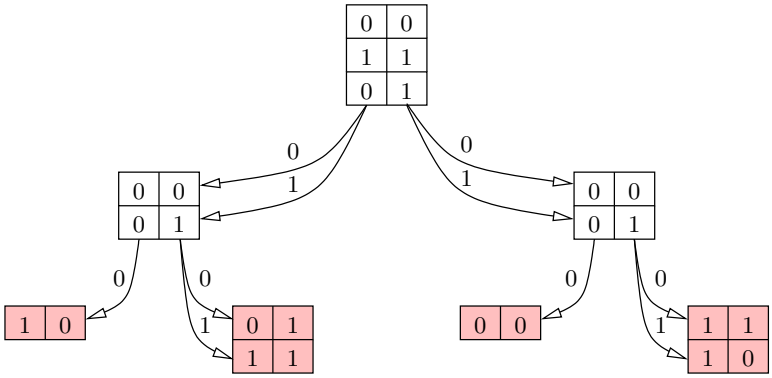
79 The processor first stores 10010011, which is represented as 00 by means of the following tree.



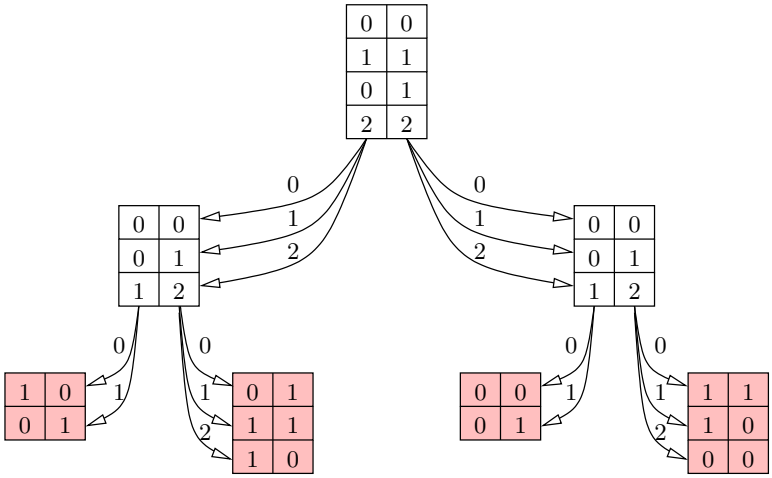
Next 10110010 is stored as 11, and the tree is adapted as follows.



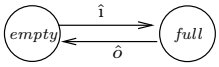
Next 10010010 is stored as 01, and the tree is adapted as follows.



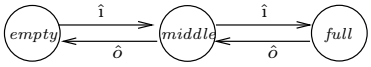
Finally 01100100 is stored as 22, and the tree is adapted as follows.



80 For $N = 1$:



For $N = 2$:



83

$$
\begin{aligned}
& X(m) \parallel X(n) \\
&= X(m) \between X(n) + X(n) \between X(m) + X(m) \mid X(n) \\
&= (a(m){\cdot}X(S(m)) \lhd m < 10 \rhd \delta + b(m){\cdot}X(S(S(m))) \lhd m > 5 \rhd \delta) \between X(n) \\
&+ (a(n){\cdot}X(S(n)) \lhd n < 10 \rhd \delta + b(n){\cdot}X(S(S(n))) \lhd n > 5 \rhd \delta) \between X(m) \\
&+ (a(m){\cdot}X(S(m)) \lhd m < 10 \rhd \delta + b(m){\cdot}X(S(S(m))) \lhd m > 5 \rhd \delta) \mid \\
&\quad (a(n){\cdot}X(S(n)) \lhd n < 10 \rhd \delta + b(n){\cdot}X(S(S(n))) \lhd n > 5 \rhd \delta) \\
&= a(m){\cdot}(X(S(m)) \parallel X(n)) \lhd m < 10 \rhd \delta \;+\; b(m){\cdot}(X(S(S(m))) \parallel X(n)) \lhd m > 5 \rhd \delta \\
&+ a(n){\cdot}(X(m) \parallel X(S(n))) \lhd n < 10 \rhd \delta \;+\; b(n){\cdot}(X(m) \parallel X(S(S(n)))) \lhd n > 5 \rhd \delta \\
&+ c(m){\cdot}(X(m) \parallel X(n)) \lhd m < 10 \wedge n > 5 \wedge m = n \rhd \delta \\
&+ c(m){\cdot}(X(m) \parallel X(n)) \lhd m > 5 \wedge m > 10 \wedge m = n \rhd \delta
\end{aligned}
$$

84 We need to prove $\mathcal{I}_i(n) \Rightarrow \mathcal{I}_i(S(S(n)))$ for $i = 1, 2$.
If n is even, then $\mathcal{I}_1(n) = \mathcal{I}_1(S(S(n))) = \mathtt{T}$ and $\mathcal{I}_2(n) = \mathcal{I}_2(S(S(n))) = \mathtt{F}$.
If n is odd, then $\mathcal{I}_1(n) = \mathcal{I}_1(S(S(n))) = \mathtt{F}$ and $\mathcal{I}_2(n) = \mathcal{I}_2(S(S(n))) = \mathtt{T}$.

85 We prove that $\mathcal{I}_i$ is an invariant for the LPE X in Definition 4; i.e., $\mathcal{I}_i(d) \wedge h(d, e) \Rightarrow \mathcal{I}_i(g(d, e))$, for $i = 1, 2$.
If $i = 1$, then we obtain $\mathtt{T} \wedge h(d, e) \Rightarrow \mathtt{T}$, which is true.
If $i = 2$, then we obtain $\mathtt{F} \wedge h(d, e) \Rightarrow \mathtt{F}$, which is true.

86 $\mathcal{I}(n) = \mathtt{T}$ for $n = 0, 2, 4, 6, 8$, and $\mathcal{I}(n) = \mathtt{F}$ for all other n.
$\mathcal{I}(n) = \mathtt{T}$ for $n = 0{-}6, 8$, and $\mathcal{I}(n) = \mathtt{F}$ for all other n.

88 The focus condition for $n{:}Nat$ is: $\exists m{:}Nat(n = 3m \vee n = S(3m))$.

$$\phi(n) = \begin{cases} \mathtt{T} \text{ if } \exists m{:}Nat(n = 3m \vee n = S(S(3m))) \\ \mathtt{F} \text{ if } \exists m{:}Nat(n = S(3m)) \end{cases}$$

The matching criteria are fulfilled:
- $n = S(S(3m)) \Rightarrow \phi(n) = \phi(S(n)) = \mathtt{T}$
- $n = 3m \Rightarrow h'_a(\phi(n)) = \phi(n) = \mathtt{T}$
  $n = S(3m) \Rightarrow h'_c(\phi(n)) = \neg\phi(n) = \neg\mathtt{F} = \mathtt{T}$
- $((n = 3m \vee n = S(3m)) \wedge \phi(n)) \Rightarrow n = 3m$
  $((n = 3m \vee n = S(3m)) \wedge \neg\phi(n)) \Rightarrow n = S(3m)$
- actions do not carry data parameters;
- $n = 3m \Rightarrow \phi(S(n)) = \mathtt{F}$
  $n = S(3m) \Rightarrow \phi(S(n)) = \mathtt{T}$

89 The focus condition for $n{:}Nat$ is: $\exists m{:}Nat(n = 3m \vee n = S(3m))$.

$$\mathcal{I}(n) = \begin{cases} \mathtt{T} \text{ if } \exists m{:}Nat(n = 3m \vee n = S(S(3m))) \\ \mathtt{F} \text{ if } \exists m{:}Nat(n = S(3m)) \end{cases}$$

The matching criteria are fulfilled for all n with $\mathcal{I}(n) = \mathtt{T}$:
- $n = S(S(3m)) \Rightarrow \phi(n) = \phi(S(n)) = nil$
- $n = 3m \Rightarrow h'_a(\phi(n)) = \mathtt{T}$
- $n = 3m \Rightarrow n = 3m$
- actions do not carry data parameters
- $n = 3m \Rightarrow \phi(S(S(n))) = nil$

91 If $j \notin p[i]$ or $s[i] = 1$, then clearly after performing an action still $j \notin p[i]$ or $s[i] = 1$, respectively.
Suppose $i \in p[j]$, $j \in p[i]$ and $s[i] = s[j] = 0$. Then after executing an action, $i \notin p[j]$ implies $s[i] = 1$, while $s[j] = 1$ implies $j \notin p[i]$.

94 1. $even(n) \Rightarrow even(S(S(n)))$ is true.
2. $even(n) \Rightarrow (even(S(S(n))) \wedge n = S(n))$ is not true.
3. $even(n) \Rightarrow even(S(S(n)))$ is true.

4. $even(n) \Rightarrow even(S(n))$ is not true.

95 Parameters y and z are found to be constant.

$$X(w{:}Nat, x{:}Nat) \;=\; a(x){\cdot}X(x,w) \triangleleft eq(0,0) \triangleright \delta \;+\; b(0){\cdot}X(S(0),x)$$

96 Parameter x is found to be inert.

$$X(y{:}D, z{:}D) \;=\; a{\cdot}X(z,y) \;+\; \sum_{w:D} b(z){\cdot}X(y,z)$$

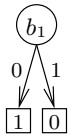
`sumelm` now finds that the sum variable w is inert.

$$X(y{:}D, z{:}D) \;=\; a{\cdot}X(z,y) \;+\; b(z){\cdot}X(y,z)$$

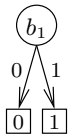
97 A single leaf with the value 0.

98 After collapsing the two leaves with the value 0 and the two leaves with the value 1 in the binary decision tree, no further reduction is possible.

99 1. The two states can be represented by a single Boolean variable; for example, s_0 is represented by 0 and s_1 by 1.
Transitions are represented by two Boolean variables b_1, b_2, where b_1 represents the source and b_2 the target state of the transition. The state space is then captured by the equations $\mathcal{O}_a(00) = 1$, $\mathcal{O}_a(01) = 1$, $\mathcal{O}_a(10) = 0$ and $\mathcal{O}_a(11) = 0$.
The binary decision tree over b_1 and b_2 (with $b_1 < b_2$) that corresponds to $\mathcal{O}_a$ reduces (by collapsing the leaves 0 and the leaves 1 and performing three reduction steps) to the reduced OBDD



2. In case of $\mu X.([a]X)$, in the first iteration step, $FIX(\phi, X{=}FALSE)$ produces the following OBBD $\mathcal{O}$:



In the second iteration step, $FIX(\phi, X{=}\mathcal{O})$ produces $\mathcal{O}$ again, so this is the final result.
In case of $\nu X.([a]X)$, in the first iteration step, $FIX(\phi, X{=}TRUE)$ produces the OBBD $TRUE$, so this is the final result.

References

1. P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: application to the bounded retransmission protocol. In W.R. Cleaveland, ed., *Proc. 5th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, Amsterdam, The Netherlands, LNCS 1579, pp. 208–222. Springer-Verlag, 1999.
2. L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, eds, *Handbook of Process Algebra*, pp. 197–292. Elsevier, 2001.
3. H.R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.
4. B. Badban. *Verification Techniques for Extensions of Equality Logic*. PhD thesis, Vrije Universiteit Amsterdam, 2006.
5. B. Badban, W.J. Fokkink, J.F. Groote, J. Pang, and J.C. van de Pol. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.
6. J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.
7. J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, 1982.
8. J. Barnat, L. Brim, I. Cerná, P. Moravec, P. Rockai, and P. Simecek. DiVinE - A tool for distributed verification. In Th. Ball and R.B. Jones, eds, *Proc. 18th Conference of Computer Aided Verification (CAV'06)*, LNCS 4144, pp. 278–281. Springer-Verlag, 2006.
9. K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.
10. T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.
11. M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, 2006.
12. J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.
13. J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, ed., *Proc. 11th Colloquium*

on Automata, Languages and Programming (ICALP'84), Antwerp, Belgium, LNCS 172, pp. 82–95. Springer-Verlag, 1984.
14. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.
15. M.A. Bezem, R.N. Bol, and J.F. Groote. Formalizing process algebraic verifications in the calculus of constructions. *Formal Aspects of Computing*, 9(1):1–48, 1997.
16. M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, eds, *Proc. 5th Conference on Concurrency Theory (CONCUR'94)*, Uppsala, Sweden, LNCS 836, pp. 401–416. Springer-Verlag, 1994.
17. S.C.C. Blom, J. Calamé, B. Lisser, S. Orzan, J. Pang, J.C. van de Pol, M. Torabi Dashti, and A.J. Wijs. Distributed analysis with μCRL: A compendium of case studies. In O. Grumberg and M. Huth, eds, *Proc. 13th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, Braga, Portugal, LNCS 4424, pp. 683–689. Springer-Verlag, 2007.
18. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, eds, *Proc. 13th Conference on Computer Aided Verification (CAV'01)*, Paris, France, LNCS 2102, pp. 250–254. Springer-Verlag, 2001.
19. S.C.C. Blom, B. Lisser, and I.A. van Langevelde. Compressed and distributed file formats for labeled transition systems. In L. Brim and O. Grumberg, eds, *Proc. 2nd Workshop on Parallel and Distributed Methods in verifiCation (PDMC'03)*, ENTCS 89(1), pp. 68–83. Elsevier, 2003.
20. S.C.C. Blom, B. Lisser, J.C. van de Pol, and M. Weber. A database approach to distributed state-space generation. Journal of Logic and Computation, 21(1):45–62, 2011.
21. S.C.C. Blom and S. Orzan. Distributed state space minimization. *Software Tools for Technlogy Transfer*, 7(3):280–291, 2005.
22. S.C.C. Blom, J.C. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, eds, *Proc. 22nd Conference on Computer Aided Verification (CAV'10)*, Edinburgh, UK, LNCS 6174, pp. 354–359. Springer-Verlag, 2010.
23. S.C.C. Blom and J.C. van de Pol. State space reduction by proving confluence. In E. Brinksma and K.G. Larsen, eds, *Proc. 14th Conference on Computer Aided Verification (CAV'02)*, Copenhagen, Denmark, LNCS 2404, pp. 596–609. Springer-Verlag, 2002.
24. B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
25. G. von Bochmann. Logical verification and implementation of protocols. In *Proc. 4th ACM-IEEE Data Communications Symposium*, Quebec, Canada, pp. 7-15–7-20. IEEE, 1975.
26. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software – Practice and Experience*, 30(3):259–291, 2000.
27. G. Bruns. *Distributed Systems Analysis with CCS*. Prentice Hall, 1997.
28. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

29. R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
30. D.E. Carlson. Bit-oriented data link control. In P.E. Green Jr, ed., *Computer Network Architectures and Protocols*, pp. 111–143. Plenum Press, 1982.
31. J.G. Cederquist, R. Corin, and M. Torabi Dashti. On the quest for impartiality: Design and analysis of a fair non-repudiation protocol. In S. Qing, W. Mao, J. Lopez, and G. Wang, eds, *Proc. 7th Conference on Information and Communications Security (ICICS'05)*, Beijing, China, LNCS 3783, pp. 27–39. Springer-Verlag, 2005.
32. V.G. Cerf and R.E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22:637–648, 1974.
33. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.
34. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, ed., *Proc. 3rd Workshop on Logics of Programs*, Yorktown Heights, New York, LNCS 131, pp. 52–71. Springer-Verlag, 1982.
35. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
36. E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In C. Courcoubetis, ed., *Proc. 5th Conference on Computer Aided Verification (CAV'93)*, Elounda, Greece, LNCS 697, pp. 450–462. Springer-Verlag, 1993.
37. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.
38. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
39. P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In E. Brinksma, ed., *Proc. 3rd Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, Enschede, The Netherlands, LNCS 1217, pp. 416–431. Springer-Verlag, 1997.
40. H. Davenport. *The Higher Arithmetic*. Cambridge University Press, 1952.
41. P.F.G. Dechering and I.A. van Langevelde. On the verification of coordination. In A. Porto and C. Roman, eds, *Proc. 4th Conference on Coordination Models and Languages (COORDINATION'00)*, Limassol, Cyprus, LNCS 1906, pp. 335–340. Springer-Verlag, 2000.
42. R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, ed., *Proc. Spring School on Semantics of Systems of Concurrent Processes*, La Roche Posay, France, LNCS 469, pp. 407–419. Springer-Verlag, 1990.
43. M.C.A. Devillers, W.O.D. Griffioen, J.M.T. Romijn, and F.W. Vaandrager. Verification of a leader election protocol – Formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, 2000.
44. E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proc. 1st IEEE Symposium on Logic in Computer Science (LICS'86)*, Cambridge, Massachusetts, pp. 267–278. IEEE Computer Society Press, 1986.
45. E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

46. W.J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2000.
47. W.J. Fokkink, J. Pang, and J.C. van de Pol. Cones and foci: A mechanical framework for protocol verification. *Formal Methods in System Design*, 29(1):1–31, 2006.
48. J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In R. Alur and T.A. Henzinger, eds, *Proc. 8th Conference on Computer Aided Verification (CAV'96)*, New Brunswick, New Jersey, LNCS 1102, pp. 437–440. Springer-Verlag, 1996.
49. H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In B. Steffen, ed., *Proc. 4th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, Lisbon, Portugal, LNCS 1384, pp. 68–84. Springer-Verlag, 1998.
50. R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
51. J.F. Groote. *Process Algebra and Structured Operational Semantics*. PhD thesis, University of Amsterdam, 1991.
52. J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. In M. Leuschel and U. Ultes-Nitsche, eds, *Proc. 3rd Workshop on Verification and Computational Logic (VCL'02)*, Pittsburgh, Pennsylvania, *ACM SIGPLAN Notices*, 37(12):98–107, 2002.
53. J.F. Groote and R. Mateescu. Verification of temporal properties of processes in a setting with data. In A.M. Haeberer, ed., *Proc. 7th Conference on Algebraic Methodology and Software Technology (AMAST'98)*, Amazonia, Brazil, LNCS 1548, pp. 74–90. Springer-Verlag, 1999.
54. J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1-2):21–56, 2003.
55. J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets: A case study in computer checked verification. In M. Wirsing and M. Nivat, eds, *Proc. 5th Conference on Algebraic Methodology and Software Technology (AMAST'96)*, Munich, Germany, LNCS 1101, pp. 536–550. Springer-Verlag, 1996.
56. J.F. Groote and J.C. van de Pol. State space reduction using partial τ-confluence. In M. Nielsen and B. Rovan, eds, *Proc. 25th Symposium on Mathematical Foundations of Computer Science (MFCS'00)*, Bratislava, Slovakia, LNCS 1893, pp. 383–393. Springer-Verlag, 2000.
57. J.F. Groote and J.C. van de Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, eds, *Proc. 7th Conference on Logic for Programming and Automated Reasoning (LPAR'00)*, Reunion Island, France, LNAI 1955, pp. 161–178. Springer-Verlag, 2000.
58. J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, eds, *Semantics of Specification Languages*, Utrecht, The Netherlands, Workshops in Computing, pp. 231–250. Springer-Verlag, 1993.
59. J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, eds, *Algebra of Communicating Processes (ACP'94)*, Utrecht, The Netherlands, Workshops in Computing, pp. 26–62. Springer-Verlag, 1995.

60. J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization of parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1/2):39–72, 2001.
61. J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, eds., *Handbook of Process Algebra*, pp. 1151–1208. Elsevier, 2001.
62. J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1/2):47–81, 1996.
63. J.F. Groote and J. Springintveld. Focus points and convergent process operators: A proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49(1/2):31–60, 2001.
64. J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M.S. Paterson, ed., *Proc. 17th Colloquium on Automata, Languages and Programming (ICALP'90)*, Warwick, UK, LNCS 443, pp. 626–638. Springer-Verlag, 1990.
65. J.F. Groote and T.A.C. Willemse. Parameterised boolean equation systems. *Theoretical Computer Science*, 343(3):332–369, 2005.
66. M. Hammer and M. Weber. 'To store or not to store' reloaded: Reclaiming memory on demand. In L. Brim, B. Haverkort and M. Leucker, eds, *Proc. 11th Workshop on Formal Methods for Industrial Critical Systems (FMICS'06)*, Bonn, Germany, LNCS 4346, pp. 51–66. Springer-Verlag, 2006.
67. L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In H.P. Barendregt and T. Nipkow, eds, *Selected Papers 1st Workshop on Types for Proofs and Programs (TYPES'93)*, Nijmegen, The Netherlands, LNCS 806, pp. 127–165. Springer-Verlag, 1994.
68. M.C.B. Hennessy and H. Lin. Unique fixpoint induction for message-passing process calculi. In *Proc. 3rd Australasian Theory Symposium on Computing (CATS'97)*, Sydney, Australia, pp. 122–131. Australia Computer Science Communications, 1997.
69. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
70. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
71. G.J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305. Kluwer, 1998
72. IEEE Computer Society. *IEEE Standard for a High Performance Serial Bus*. Std. 1394-1995, August 1996.
73. ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*. ITU-T, Geneva, June 1994.
74. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
75. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Conference Record 12th ACM Symposium on Principles of Programming Languages (POPL'85)*, New Orleans, Louisiana, pp. 97–107. ACM, 1985.
76. H. Lin. Symbolic transition graph with assignment. In V. Sassone, ed., *Proc. 7th Conference on Concurrency Theory (CONCUR'96)*, Pisa, Italy, LNCS 1119, pp. 50–65. Springer-Verlag, 1996.
77. J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley/Teubner, 1996.

78. S.P. Luttik. Description and formal specification of the link layer of P1394. In I. Lovrek, ed., *Proc. 2nd Workshop on Applied Formal Methods in System Design*, Zagreb, Croatia, 1997.
79. S.P. Luttik. *Choice Quantification in Process Algebra*. PhD thesis, University of Amsterdam, 2002.
80. N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
81. W.C. Lynch. Reliable full duplex file transmission over half-duplex telephone lines. *Communications of the ACM*, 11(6):407–410, 1968.
82. A.H. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, Technical University of Munich, 1997.
83. R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, 2003.
84. K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer, 1993.
85. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.
86. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
87. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer-Verlag, 2002.
88. S. Owre, J.M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, ed., *Proc. 11th Conference on Automated Deduction (CADE'92)*, Saratoga Springs, New York, LNCS 607, pp. 748–752. Springer-Verlag, 1992.
89. J. Pang, W.J. Fokkink, R.F.H. Hofman, and R. Veldema. Model checking a cache coherence protocol for a Java DSM implementation. *Journal of Logic and Algebraic Programming*, 71(1):1–43, 2007.
90. D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, ed., *Proc. 5th GI (Gesellschaft für Informatik) Conference*, Karlsruhe, Germany, LNCS 104, pp. 167–183. Springer-Verlag, 1981.
91. D.A. Peled. *Software Reliability Methods*. Springer-Verlag, 2001.
92. G.D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
93. A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, Providence, Rhode Island, pp. 46–57. IEEE Computer Society Press, 1977.
94. J.C. van de Pol. A prover for the μCRL toolset with applications – version 1. Technical Report SEN-R0106, CWI, 2001.
95. J.C. van de Pol and M. Valero Espada. An abstract interpretation toolkit for μCRL. *Formal Methods in System Design*, 30(3):249–273, 2007.
96. J. Rathke. Unique fixpoint induction for value-passing processes. In *Proc. 12th IEEE Symposium on Logic in Computer Science (LICS'97)*, Warsaw, Poland, pp. 140–148. IEEE Computer Society Press, 1997.
97. J.M.T. Romijn. A timed verification of the IEEE 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, 2001.
98. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
99. C. Shankland and M.B. van der Zwaag. The tree identify protocol of IEEE 1394 in μCRL. *Formal Aspects of Computing*, 10(5/6):509–531, 1998.
100. M. Sighireanu and R. Mateescu. Verification of the link layer protocol of the IEEE-1394 serial bus (FireWire): An experiment with E-LOTOS. *Software Tools for Technology Transfer*, 2(1):68–88, 1998.

101. M.I.A. Stoelinga and F.W. Vaandrager. Root contention in IEEE 1394. In J.-P. Katoen, ed., *Proc. 5th AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, Bamberg, Germany, LNCS 1601, pp. 53–74. Springer-Verlag, 1999.
102. A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.
103. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
104. Terese. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science 55. Cambridge University Press, 2003.
105. A.J. Wijs, J.C. van de Pol, and E. Bortnik. Solving scheduling problems by untimed model checking: The clinical chemical analyser case study. In T. Margaria and M. Massink, eds, *Proc. 10th Workshop on Formal Methods for Industrial Critical Systems (FMICS'05)*, Lisbon, Portugal, pp. 54–61. ACM Press, 2005.
106. A.G. Wouters. Manual for the μCRL toolset (version 2.8.2). Technical Report SEN-R0130, CWI, 2001.
107. M. Ying. Weak confluence and τ-inertness. *Theoretical Computer Science*, 238(1/2):465–475, 2000.

Index