Modelling Distributed Systems

Wan Fokkink
Specification and Analysis of Embedded Systems
Department of Software Engineering
CWI
Email: wan@cwi.nl

Jan Friso Groote & Michel Reniers
Systems Engineering
Department of Computer Science
Eindhoven University of Technology, Eindhoven
Email: J.F.Groote@tue.nl, M.A.Reniers@tue.nl

Contents

Chapter 1

Introduction

A distributed system is driven by separate components that are being executed in parallel. In today's world of wireless and mobile networking, protocols for distributed systems form a major aspect of system design. Verifying the correctness of such protocols is usually a formidable task, as even simple behaviours become wildly complicated when they are executed in parallel. In order to study distributed systems in detail, it is imperative that they are dissected into their concurrent components.

Process algebra focuses on the specification and manipulation of process terms as induced by a collection of operator symbols. Process algebras such as CCS [24, 84, 86], CSP [67, 68, 95] and ACP [11, 5, 43] have proven to be nice basic languages for the description of elementary parallel systems and they are well equipped for the study of behavioural properties of distributed systems. Fundamental to process algebra is a parallel operator, to break down systems into their concurrent components. Most process algebras contain basic operators to build finite processes, communication operators to express concurrency, and some notion of recursion to capture infinite behaviour. Moreover, a special hiding operator allows one to abstract away from internal computations. In process algebras, each operator in the language is given meaning through a characterising set of equations called axioms. If two process terms can be equated by means of the axioms, then they constitute equivalent processes. Thus the axioms form an elementary basis for equational reasoning about processes.

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, 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 explicitly denoted, and henceforth it is clear which assumptions can be used when proving properties about data or processes. Term rewriting [4] 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 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 becomes 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 study of processes, the language *micro* CRL[59] has been developed,[1] denoted μCRL (or mCRL, if Greek letters are unavailable). Basically, μCRL is based on the process algebra ACP, extended with equational abstract data types. In order to intertwine processes with data, actions and recursion variables can be parametrised with data types. Moreover, a conditional (if-then-else construct) can be used to have data elements influence the course of a process, and alternative quantification is added to sum over possibly infinite data domains.

Despite its lack of 'advanced features', μCRL has shown to be remarkably apt for the description of large distributed systems. It has been the basis for the development of a proof theory [58], based in part on the axiomatic semantics of the process algebra ACP and of some basic algebraic data types. This proof theory in combination with proof methods that were developed in [14, 62] has enabled the verification of large distributed systems in a precise and logical way (allowing the proof to be checked by proof checkers), which is slowly turning into a routine. After having mastered the skill of process verification, one will experience that virtually any protocol for a distributed system that has not been proven correct contains flaws of a more or less serious nature. And you will hopefully agree that μCRL and its tool support are well-suited to help detect these problems.

In μCRL we strive for precision in proofs. Therefore, an important research area is to use theorem provers such as PVS [87], HOL [48], Isabelle [90] and Nqthm [21] to help in finding and checking derivations in μCRL. A large number of distributed systems have been verified in μCRL [15, 16, 45, 50, 73, 74, 97, 104], often with the help of a proof checker or theorem prover [13, 53, 55, 75]. See [52] for an overview of such case studies. Typically, these verifications lead to the detection of a number of mistakes in the specification of the system under scrutiny, and the support of proof checkers helps to detect flaws in the correctness proof or even in the statement of correctness.

To each μCRL specification there belongs a directed graph, in which the states are process terms, and the edges are labelled with actions. In this

[1] This language started off as a restricted variant of the so-called Common Representation Language (CRL), which was developed within the research project SPECS, funded by the European Community. CRL was intended to serve as a platform to which a family of specification languages could be translated.

state space, an edge $t \overset{a(d)}{\rightarrow} t'$ means that process term t can perform action a, parametrised with data element d, to evolve into process term t'. If the state space belonging to a μCRL specification is finite, then the μCRL tool set in combination with the CADP toolset [44] can generate and visualise this state space. While the process algebraic proofs that were discussed earlier can cope with unspecified data types, the generation of a state space belonging to a distributed system requires that all data domains are fully specified. This means that 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. Interestingly, if such a data type does not influence the behaviour of a process, then proofs of properties of this process with respect to an ad hoc finite instantiation of the data type may generalise to the case where this data type is left unspecified; see [78] and [95, Section 15.2].

A severe complication in the generation of state spaces is that in real life, a finite-state 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. On-the-fly analysis [69] 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 [31]. Scenario-based verification [38] takes as 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 [51]. The ATerm library [22] 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. The SVC file format [77] targets optimal representation of state spaces using standard compression techniques, typically reducing a state space to about 2% of its original size. Model checking [32] provides a framework to efficiently prove interesting properties of large state spaces. Finally, algorithms have been developed to reduce state spaces after abstracting away from internal computations.

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 explains the basics of the specification language μCRL. In Chapter 5 it is explained how one can abstract away from the internal computation steps of a process. Chapter 6 contains a number of μCRL specifications of protocols from the literature, together with extensive explanations to guide the reader through these specifications. In Chapter 8 a number of standard process algebraic techniques are described that can be used in the verification of μCRL specifications. In Chapter 10, these techniques are applied in the verification of the tree identify protocol and a sliding window protocol. Chapter 7 describes algorithms on graphs. In Chapter 9, techniques are presented to analyse and adapt μCRL specifications on a symbolic level. Chapter 11 gives an overview and some applications of an extension of the language μCRL with time. Finally, Chapter 12 contains a short overview of the μCRL toolset.

Chapter 2

Abstract Data Types

In this chapter we give an introduction to the algebraic specification of abstract data types by means of a set of equations. See [10, 80] for lucid overviews of this field.

2.1 Algebraic specification

We start with an example.

Example 2.1 *As a standard example we specify the natural numbers with addition and multiplication. The* signature *consists of the function 0 of arity zero, the unary successor function S, and the binary functions addition plus and multiplication mul. The equality relation on terms is specified by four* axioms*:*

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

The initial model *of this axiomatisation 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 Figure 2.1.

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

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

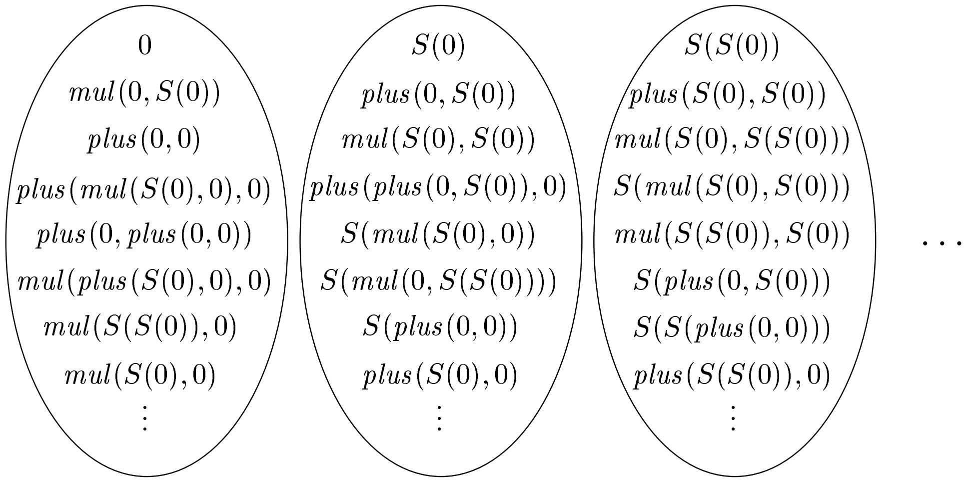


Figure 2.1: Initial model for the natural numbers

and

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

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

1. A *signature*, consisting of function symbols, from which one can build terms. Each function symbol is assigned an *arity*, denoting the number of arguments of the function symbol.

2. A set of *axioms*, i.e., equations between terms (possibly containing variables), which induces an equality relation on terms.

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

- if $s = t$ is an axiom, then $\sigma(s) = \sigma(t)$ holds for all possible substitutions σ from variables to terms;

- if $s = t$ holds, then $C[s] = C[t]$ holds for all possible contexts $C[]$;
- $t = t$ holds for all terms t;
- if $s = t$ holds, then $t = s$ holds;
- if $s = t$ and $t = u$ hold, then $s = u$ holds.

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.

Each data type is declared using the keyword **sort**. Each declared sort represents a non-empty set of data elements. Declaring the sort of the booleans is simply done by:

sort *Bool*

Elements of a data type are declared by using the keywords **func** and **map**. Using **func** one can declare constructors with as target sort the data type in question; these constructors define the structure of the data type. E.g. by

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

one 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 are used in the if-then-else construct in the process language, the sort *Bool* must be declared in every μCRL specification. Besides the declaration of sort *Bool*, it is also obligatory that T and F are declared in every specification. Moreover, it is assumed that T and F are distinct, and that they are the only two elements in *Bool*. This is expressed by the axioms Bool1 and Bool2 in Table 2.1. In axiom Bool2 and elsewhere we use a variable b that can only be instantiated with data terms of sort *Bool*. If in a specification T and F can be proven equal, for instance if the specification contains an equation $\mathtt{T} = \mathtt{F}$, we say that the specification is inconsistent and it looses any meaning.

Bool1	$\neg(\mathtt{T} = \mathtt{F})$
Bool2	$\neg(b = \mathtt{T}) \Rightarrow b = \mathtt{F}$

Table 2.1: Basic axioms for *Bool*

It is now easy to declare the natural numbers, in the logician's style, using the constructors zero 0 and successor S.

sort $Bool, Nat$
func $\mathtt{T}, \mathtt{F} :\to Bool$
$0 :\to Nat$
$S : Nat \to Nat$

This says that each natural number can be written as 0 or the application of a number of successors to 0.

When declaring a sort, it is required that this data type is not empty. For example, the following declaration is invalid.

sort D
func $f : D \to D$

It declares that D is a domain in which all the terms have the form $f(f(f(\ldots)))$, i.e., an infinite number of applications of f. As terms are finite constructs, such terms do not exist. Fortunately it is easy to detect such problems, and therefore it is a static semantic constraint that such empty sorts must not occur (see [59]).

If for a sort D there is no constructor with target sort D, then it is assumed that D may be arbitrarily large. In particular D may contain elements that cannot be denoted by terms. This can be extremely 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 in μCRL:

sort D

The keyword **map** is used to declare additional functions for a data type of which the structure is already given; in other words, in **map** one declares functions symbols that are not constructors.[1] 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 $\wedge : Bool \times Bool \to Bool$
$plus : Nat \times Nat \to Nat$

By adding plain equations, of the form $term = term$, assumptions about the functions can be added. For the two functions declared above, we could add the equations:

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

[1]The distinction between constructors and non-constructors in μCRL is essential for the implementation of summation over a data type; see Section 3.7. Namely, $\sum_{d:D}$ - sums over all terms d that can be built from the functions that have been declared in the **func** section of the specification of sort D.

The keyword **rew** refers to the fact that the equations are applied as rewrite rules from left to right; see Section 2.2. Note that before each group of equations starting with the keyword **rew** one must declare the variables that are used.

Exercise 2.2 *Declare disjunction* $\vee$*, negation* $\neg$*, implication* $\Rightarrow$ *and bi-implication* $\Leftrightarrow$ *on the booleans, and the monus function* $\dot{-}$ *on the naturals (with* $n \dot{-} m = 0$ *if* $n \leq m$*), and provide equations for these functions.*

Exercise 2.3 *Give algebraic specifications of 'greater than or equal'* $\geq$*, 'smaller than'* $<$ *and 'greater than'* $>$ *on the natural numbers.*

Exercise 2.4 *Give algebraic specifications of* $even : Nat \to Bool$ *and* $power : Nat \times Nat \to Nat$*, such that* $even(n) = \mathtt{T}$ *if and only if* n *is an even number, and* $power(m, n)$ *equals* m^n*.*

Exercise 2.5 *Define a sort* $List$ *on an arbitrary non-empty domain* D*, with as constructors the empty list* $[] :\to List$ *and* $in : D \times List \to List$ *to insert an element of* D *at the beginning of a list. Extend this with the following non-constructor functions:* $append : D \times List \to List$ *to insert an element of* D *at the end of a list;* $top : List \to D$ *and* $toe : List \to D$ *to obtain the first and the last element of a list, respectively;* $tail : List \to List$ *and* $untoe : List \to List$ *to remove the first and the last element from a list, respectively;* $nonempty : List \to Bool$ *to check whether a list is empty,* $length : List \to Nat$ *to compute the length of a list, and* $++ : List \times List \to List$ *to concatenate two lists.*

Note that in Exercise 2.5 the function $\top$ is not well-defined on the empty list, meaning that $\top([])$ is not equal to an element of D. Likewise, as the empty list does not contain a first element, preferably one leaves $tail([])$ unspecified, meaning that this term is not equal to an element of $List$. This has the advantage that a verification using this algebraic specification of lists is independent of the fact whether or not it is possible to remove the (non-existent) first element of an empty list. However, if some process verification does use in an essential way that one is allowed to remove the first element of the empty list, one could consider adding the equation $tail([]) = []$ to the specification of lists.

The machine-readable syntax of μCRL only allows prefix functions in ASCII characters. For example, in this machine-readable syntax conjunction could read `and(b,b')` (instead of $b \wedge b'$). However, in this text we use them freely infix or even postfix, and not necessarily restricted to ASCII characters, if we believe that this increases readability. Moreover, we use common mathematical symbols such as $\wedge$, which are also not allowed by the syntax of μCRL, for the same reason.

Functions may be overloaded, as long as every term has a unique sort. This means that the name of the function together with the sorts of its arguments must be unique. E.g. it is possible to declare $max : Nat \times Nat \to Nat$ and $max : Bool \times Bool \to Bool$, but it is not allowed to declare functions $f : Bool \to Bool$ and $f : Bool \to Nat$.

2.2 Term rewriting

Term rewriting [4] 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 is not a single variable and the second term does not contain fresh variables. (If one of these two restrictions is violated, the term rewriting system would 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. An up-to-date overview of term rewriting can be found in [4]. We give an example.

Example 2.6 *As an example of a term rewriting system, we direct the four equations for natural numbers (see Example 2.1) from left to right:*

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

Using this term rewriting system we can reduce the 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 reduced is underlined.*

$$\begin{array}{rcl} \underline{mul(S(0),S(S(0)))} & \stackrel{(4)}{\to} & plus(\underline{mul(S(0),S(0))},S(0)) \\ & \stackrel{(4)}{\to} & plus(plus(\underline{mul(S(0),0)},S(0)),S(0)) \\ & \stackrel{(3)}{\to} & plus(\underline{plus(0,S(0))},S(0)) \\ & \stackrel{(2)}{\to} & plus(S(\underline{plus(0,0)}),S(0)) \\ & \stackrel{(1)}{\to} & \underline{plus(S(0),S(0))} \\ & \stackrel{(2)}{\to} & S(\underline{plus(S(0),0)}) \\ & \stackrel{(1)}{\to} & S(S(0)). \end{array}$$

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 an axiomatisation, one can try to derive an equation $s = t$ by giving a direction to each of the axioms, to obtain a term rewriting system, and attempting to reduce s and t to the same normal form. If the resulting term rewriting system is terminating and confluent, then this procedure to try and equate s and t is guaranteed to return a derivation if $s = t$.

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 layout of the term rewriting system). 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 usually involves only little run-time overhead compared to outermost rewriting.

2.3 Equality functions

In μCRL one needs to specify an *equality* function $eq : D \times D \to Bool$, reflecting equality between terms.[2] That is, $eq(d,e) = \mathtt{T}$ if $d = e$ and $eq(d,e) = \mathtt{F}$ if $d \neq e$, for all $d, e{:}D$. For example, for the booleans one could define an equality function as follows.

$$
\begin{array}{ll}
\textbf{rew} & eq(\mathtt{T},\mathtt{T}) = \mathtt{T} \\
 & eq(\mathtt{F},\mathtt{F}) = \mathtt{T} \\
 & eq(\mathtt{T},\mathtt{F}) = \mathtt{F} \\
 & eq(\mathtt{F},\mathtt{T}) = \mathtt{F}
\end{array}
$$

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 $n = m$.

$$
\begin{array}{ll}
\textbf{var} & n, m{:}Nat \\
\textbf{rew} & eq(0,0) = \mathtt{T} \\
 & eq(S(n),0) = \mathtt{F} \\
 & eq(0,S(n)) = \mathtt{F} \\
 & eq(S(n),S(m)) = eq(n,m)
\end{array}
$$

Exercise 2.7 *Assuming an equality function eq on the data type D, specify an equality function eq on the data type List of lists over D.*

Exercise 2.8 *Assume a data type D, an if-then-else function if and an equality function eq.*

$$
\begin{array}{ll}
\textbf{map} & eq : D \times D \to Bool \\
 & if : Bool \times D \times D \to D \\
\textbf{var} & d, e{:}D \\
\textbf{rew} & eq(d,d) = \mathtt{T} \\
 & if(\mathtt{T},d,e) = d \\
 & if(eq(d,e),d,e) = e
\end{array}
$$

Show that these equations imply for all $d, e{:}D$ that:

$$eq(d,e) = \mathtt{T} \quad \Leftrightarrow \quad d = e.$$

[2] Actually, such an equality function is only needed for data types that are used as parameters of actions that occur in a communication; see Section 3.3. For example, if $a\,|\,b = c$, then in the μCRL tool set an expression $a(d)\,|\,b(e)$ is transformed into $c(d) \lhd eq(d,e) = \mathtt{T} \rhd \delta$.

We note that the specification of the *eq* function in Exercise 2.8 has poor term rewriting properties. For example, the specification of the *eq* function on *Nat* above reduces the term $eq(S(0), 0)$ to its desired normal form F, while with respect to the specification in Exercise 2.8 $eq(S(0), 0)$ is a normal form.

2.4 Induction

We explain how one can use induction to derive equality between data terms, and also how one can prove data terms to be non-equal.

The division between constructors and mappings gives us general induction principles. If there are constructor symbols with target sort D, we may assume that every term of sort D can be written as the application of a constructor to a number of arguments. So if we want to prove a property $p(d)$ for all terms d of sort D, we only need to provide proofs of $p(f(d_1, \ldots, d_n))$ for all constructors $f : S_1 \times \cdots S_n \to D$, where each term d_i is of sort S_i. If any of the arguments of f, say argument d_j, is of sort D, then as d_j is smaller than $f(d_1, \ldots, d_n)$ we may assume that $p(d_j)$ holds. If we apply this line of argumentation, we say that we apply *induction* on D.

Recall that the axiom Bool2 says that if a boolean term b is not equal to T then it must be equal to F. In other words, there are at most two boolean values. Applying this axiom boils down to a case distinction, proving a statement for the values T and F, and concluding that the property must then universally hold. We refer to this style of proof by the phrase 'induction on booleans'. Note that the sort *Bool* is the only sort for which we explicitly describe that the constructors T and F are different. For other sorts, like *Nat*, there are no such axioms.

A typical example of induction on booleans is the following proof of $b \wedge b = b$. By induction it suffices to prove that this equation holds 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 trivial instances of the defining axioms for $\wedge$ mentioned before.

Exercise 2.9 *Prove by induction that* $x \vee x = x$ *and* $\neg\neg x = x$ *for all booleans* x.

Suppose we have declared the natural numbers with constructors 0 and S, as in Example 2.1. We can for instance derive by induction that $plus(0, n) = n$ for all natural numbers n. First we must show that $plus(0, 0) = 0$, considering the case where $n = 0$. This is a trivial instance of the first axiom on addition. Second we must show $plus(0, S(n')) = S(n')$, assuming that n has the form $S(n')$. As n' is smaller than n, in this case we may assume that the property to be proven holds for n', i.e., $plus(0, n') = n'$. Then we obtain:

$$plus(0, S(n')) = S(plus(0, n')) = S(n').$$

Exercise 2.10 *Prove by induction that:*

(1) $\mathtt{F} \wedge x = \mathtt{F}$*;*

(2) $x \Rightarrow \mathtt{T} = \mathtt{T}$*;*

(3) $x \Rightarrow \mathtt{F} = \neg x$*;*

(4) $x \Rightarrow y = \neg y \Rightarrow \neg x$*;*

(5) $x \Leftrightarrow \mathtt{T} = x$*;*

(6) $x \Leftrightarrow x = \mathtt{T}$*;*

(7) $x \Leftrightarrow \neg y = \neg(x \Leftrightarrow y)$*;*

(8) $(x \vee y) \Leftrightarrow x = x \vee \neg y$*;*

(9) $even(plus(m,n)) = even(m) \Leftrightarrow even(n)$*;*

(10) $even(mul(m,n)) = even(m) \vee even(n)$*.*

Exercise 2.11 *Prove by induction that:*

(1) $mul(0,n) = 0$*;*

(2) $plus(plus(k,\ell),m) = plus(k, plus(\ell,m))$*;*

(3) $mul(k, plus(\ell,m)) = plus(mul(k,\ell), mul(k,m))$*;*

(4) $mul(mul(k,\ell),m) = mul(k, mul(\ell,m))$*;*

(5) $mul(power(m,k), power(m,\ell)) = power(m, plus(k,\ell))$*;*

Exercise 2.12 *Describe the concatenation of a list q from which the last element has been removed to a list q' into which the last element of q has been inserted. Prove, using your equations from Exercise 2.5 and induction, that if q is non-empty, then this concatenation is equal to the concatenation of q and q'.*

In μCRL it is possible to establish when two data terms are not equal, by assuming that they are equal, and showing that this implies $\mathtt{T} = \mathtt{F}$, contradicting axiom Bool1. We give an example showing that the natural numbers zero and one are not equal.

Example 2.13 *Let the natural numbers with a 0 and successor function S be appropriately declared. In order to show that 0 and $S(0)$ are different, we need a function that relates Nat to Bool, in order to be able to apply axiom Bool1. For this function we choose 'less than or equal to', notation $\leq$, defined as follows:*

map $\leq: Nat \times Nat \to Bool$
var $n, m{:}Nat$
rew $0 \leq n = \mathtt{T}$
$S(n) \leq 0 = \mathtt{F}$
$S(n) \leq S(m) = n \leq m$

Now assume $0 = S(0)$. *Clearly,* $0 \leq 0 = \mathtt{T}$. *But, using the assumption, we also find* $0 \leq 0 = S(0) \leq 0 = \mathtt{F}$. *So, we can prove* $\mathtt{T} = \mathtt{F}$. *Hence, we may conclude* $0 \neq S(0)$.

Exercise 2.14 *Prove, using Exercise 2.5, that the empty list and a nonempty list must always be different.*

Chapter 3

Process Algebra

This chapter presents the basics of the specification language μCRL. Its framework consists of process algebra, for describing system behaviour, enhanced with abstract data types.

In the μCRL specifications presented in this text we use LaTeX type setting features at will. Actually, 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. In this text, however, we try to optimise readability.

The meaning of each operator in the language is captured by means of a number of axioms, following the process algebraic tradition. Basically, they assist in interpreting the language correctly. Often the semantics of the language is given in terms of a data algebra and an operational semantics; these are not treated here. The axioms provide an alternative semantics of the language. Furthermore, the axioms form an elementary basis for equational reasoning about processes.

3.1 Actions

Actions represent events in the real world. For instance, sending the number 3 can be described by *send*(3), and boiling food can be described by *boil*(*food*), where 3 and *food* are terms declared by a data type specification. In general, an action consists of an action name possibly followed by zero or more data terms within brackets. Intuitively, an action $a(d_1,\ldots,d_n)$ can execute itself, after which it terminates successfully:

$$\begin{array}{c} a(d_1,\ldots,d_n) \\ \Big\downarrow{\scriptstyle a(d_1,\ldots,d_n)} \\ \surd \end{array}$$

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 the data with which it is parametrised. 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 parameters, an action a that is parametrised with booleans, and an action b that is parametrised with pairs of natural numbers and data elements.

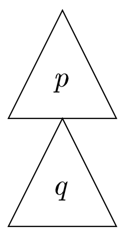
act *time-out*
$a{:}Bool$
$b{:}Nat \times D$

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 sort Nat and with a pair of sorts $D \times Bool$.

In accordance with process algebras such a CCS, CSP and ACP, actions in μCRL are considered to be atomic. If an event has a certain positive duration, such as boiling food, then it is most appropriate to consider the action as the beginning of the event. If the duration of the event is important, separate actions for the beginning and termination of the event can be used.

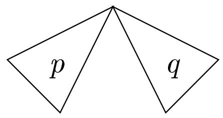
3.2 Alternative and sequential composition

Two elementary operators to construct processes are the *sequential composition* operator, written $p{\cdot}q$ and the *alternative composition* operator, written $p + q$. The process $p{\cdot}q$ 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 $r \xrightarrow{a} q$:



It is common to omit the sequential composition operator in process expressions. That is, pq denotes $p{\cdot}q$.

The process $p+q$ behaves as p or q, depending on which of the two processes 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 roots:



p and q are called *summands* of $p+q$; moreover, all summands of p and q are also summands of $p+q$.

Exercise 3.1 *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.*

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	$(x{\cdot}y){\cdot}z=x{\cdot}(y{\cdot}z)$

Table 3.1: Basic axioms for μCRL

In Table 3.1 axioms A1-A5 are listed, describing the elementary properties of the sequential and alternative composition operators. In these axioms we use variables x, y and z that can be instantiated by process terms. 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. As binding convention we assume that the $\cdot$ binds stronger than the $+$. For example, $a \cdot b + a \cdot c$ represents $(a \cdot b) + (a \cdot c)$.

Exercise 3.2 *Derive the following three equations from A1-A5:*

(1) $((a+a){\cdot}(b+b)){\cdot}(c+c)=a{\cdot}(b{\cdot}c)$;

(2) $(a+a){\cdot}(b{\cdot}c)+(a{\cdot}b){\cdot}(c+c)=(a{\cdot}(b+b)){\cdot}(c+c)$;

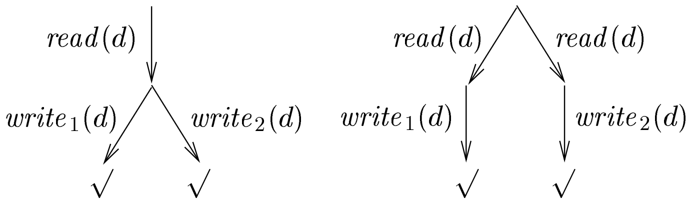
(3) $((a+b){\cdot}c+a{\cdot}c){\cdot}d=(b+a){\cdot}(c{\cdot}d)$.

We use the shorthand $x \subseteq y$ for $x+y=y$, and write $x \supseteq y$ for $y \subseteq x$. This notation is called *summand inclusion*. It is possible to divide the proof of an equation into proving two inclusions, as the following exercise shows.

Exercise 3.3 *Prove for all μCRL processes p and q: if $p \subseteq q$ and $q \subseteq p$, then $p=q$.*

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

Example 3.4 *Consider the two processes below:*



The first process reads datum d, and then decides whether it writes d on disc 1 or on disc 2. The second process makes a choice for disc 1 or disc 2 before it reads datum d. Both processes 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 processes, which becomes apparent if for instance disc 1 crashes. In this case the first process always saves datum d on disc 2, while the second process may get into a deadlock (i.e., may get stuck); see Section 3.4 for a formal definition of such deadlocks.*

Exercise 3.5 *Prove that the axioms A1-3 are equivalent to axiom A3 together with*

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

3.3 Parallel processes

The parallel operator can be used to put processes in parallel. The behaviour of $p \parallel q$ is the arbitrary interleaving of actions of the processes p and q, assuming for the moment that there is no communication between p and q. For example, if there is no communication possible between the actions a and b, then the process $a \parallel b$ behaves as $a\cdot b + b\cdot a$.

It is possible to let processes p and q in $p \parallel q$ communicate. This can be done to declare in a communication section that certain action names can synchronise. In μCRL this is done as follows:

comm $a\,|\,b = c$

Suppose that two actions $a(\vec{d})$ and $b(\vec{d})$ can happen in parallel, where $\vec{d}$ abbreviates a sequence $d_1, \ldots, d_n$. Then the communication declaration implies that they may synchronise; this communication is denoted by $c(\vec{d})$. For example, the process $a \parallel b$ 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.

Communication between actions is commutative and associative:

$$\begin{array}{rcl} a\,|\,b & = & b\,|\,a \\ (a\,|\,b)\,|\,c & = & a\,|\,(b\,|\,c) \end{array}$$

The commutative counterpart $b \,|\, a = c$ 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 3.6 *Let the communication of two actions from $\{a, b, c\}$ always result to c. The state space of the process term $(a{\cdot}b) \parallel (b{\cdot}a)$ is depicted in Figure 3.1.*

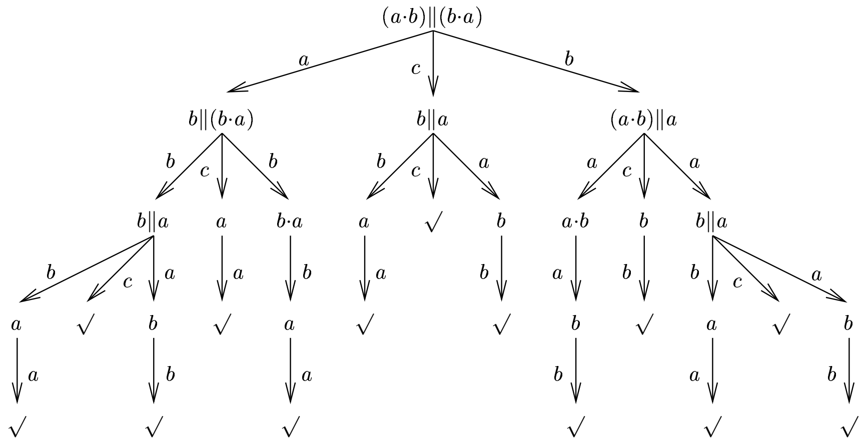


Figure 3.1: State space of $(a{\cdot}b) \parallel (b{\cdot}a)$

Example 3.6 shows that the parallel composition of two simple processes produces a relatively large state space. This partly explains the strength of a theory of communicating processes, as this theory makes it possible to draw conclusions about the full system by studying its separate concurrent components.

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

The core law for the parallel operator is CM1 in Table 3.2. It says that $x \parallel y$ either x performs the first step, represented by the summand $x \mathbin{\|\!\_}\, y$, or y can do the first step, represented by $y \mathbin{\|\!\_}\, x$, or the first step of $x \parallel y$ is a communication between x and y, represented by $x \,|\, y$. 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. In the axioms, $\vec{d}$ and $\vec{e}$ denote lists of data parameters.

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

Table 3.2: Axioms for parallellism in μCRL

Axiom CF′ features the special constant δ, which does not display any behaviour. This constant will be explained in the next section. CF′ expresses that $a(\vec{d}) \mid b(\vec{e})$ does not display any behaviour if a and b do not communicate or if $\vec{d}$ and $\vec{e}$ are distinct.

Exercise 3.7 *Suppose that a and b communicate to b', while a and c communicate to c'. Derive the equation $a \parallel (b + c) = (b + c) \parallel a$.*

3.4 Deadlock and encapsulation

If two actions are able to communicate, then often we only want these actions 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 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 constant δ called *deadlock*, which does not display any behaviour. The deadlock does not carry any data parameters. 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. Generally, this is only done when we want to force this action into a communication. The unary *encapsulation* operators ∂_H, for subsets H of Act, are especially designed for this task. A process $\partial_H(p)$ 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 3.8. The axioms for deadlock and encapsulation are listed in Table 3.3.

A6	$x + \delta = x$	
A7	$\delta \cdot x = \delta$	
DD	$\partial_H(\delta) = \delta$	
D1	$\partial_H(a(\vec{d})) = a(\vec{d})$	if $a \notin H$
D2	$\partial_H(a(\vec{d})) = \delta$	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	$\delta \mathbin{\|\!\_} x = \delta$	
CD2	$\delta \mid x = \delta$	
CD3	$x \mid \delta = \delta$	

Table 3.3: Axioms for deadlock

Example 3.8 *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 to $comm(d)$ for $d \in \{0, 1\}$, while all other communications between actions result to δ. The behaviour of the channel is described by the process term*

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

The encapsulation operator enforces that the action $send(d)$ can only occur in communication with the action $read(d)$, for $d \in \{0, 1\}$.

Exercise 3.9 *Suppose action a cannot communicate with itself. Derive the equation $(b \cdot a) \parallel a = (b \parallel a) \cdot a$.*

Exercise 3.10 *Let the communication of two actions from* $\{a, b, c\}$ *always result to* c. *Derive* $\partial_{\{a,b\}}((a{\cdot}b) \parallel (b{\cdot}a)) = c{\cdot}c$. *(Cf. Example 3.6).*

Exercise 3.11 *Use the axioms to equate the process*

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

from Example 3.8 and $comm(0) + comm(1)$.

Exercise 3.12 *Give an example to show that processes* $\partial_H(p \parallel q)$ *and* $\partial_H(p) \parallel \partial_H(q)$ *can display distinct behaviour.*

Exercise 3.13 *Suppose* $p + q = \delta$ *can be derived for certain processes* p *and* q. *Derive* $p = \delta$.

Beware not to confuse a transition of the form $t \stackrel{a}{\to} \delta$ with a transition of the form $t \stackrel{a}{\to} \surd$; intuitively, the first transition expresses that t gets stuck after the execution of a, while the second transition expresses that t terminates successfully after the execution of a. A process p is said to contain a deadlock if there is a sequence of transitions $p \stackrel{a_1}{\to} p_1 \stackrel{a_2}{\to} \cdots \stackrel{a_n}{\to} p_n$ such that the process p_n cannot perform any actions. In general it is undesirable that a process contains a deadlock, because it represents that the process gets stuck without producing any output. Experience learns that non-trivial specifications of system behaviour often contain a deadlock. For example, the third sliding window protocol in [100] contains a deadlock; see [49, *Stelling 7*]. It can, however, be very difficult to detect such a deadlock, even if one has a good insight into such a protocol.

If one process term contains a deadlock while the other does not, then we do not want these terms to be equal. Namely, we only want to equate process terms that exhibit equivalent behaviour. It is for this reason that we have to distinguish terms such as $a{\cdot}b + a{\cdot}c$ and $a{\cdot}(b + c)$. Namely, as $a{\cdot}\delta + a{\cdot}c$ contains a deadlock, it must not be equal to the deadlock-free term $a{\cdot}c$. In other words, $\partial_{\{b\}}(a{\cdot}b + a{\cdot}c))$ must not be equal to $\partial_{\{b\}}(a{\cdot}(b + c))$. Clearly this implies that $a{\cdot}b + a{\cdot}c$ and $a{\cdot}(b + c)$ must not be equal.

Exercise 3.14 *Let the communication of* b *and* c *result to* 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) $\partial_{\{b,c\}}((a{\cdot}b) \parallel c)$.

3.5 Process declarations

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

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

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

The equations in a process declaration are to be considered as equations in the ordinary mathematical sense. This means that with a declaration such as the one above, an occurrence of $X(e_1, \ldots, e_n)$ can be replaced by $t[x_1{:=}e_1, \ldots, x_n{:=}e_n]$, or vice versa. The equations are to be *guarded*, meaning that occurrences of expressions $Y(d_1, \ldots, d_m)$ are to be proceeded by actions. In other words, such an expressions always occurs in a context of the form $C_1[a{\cdot}C_2[]]$. 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 process name Z; as a result, such an equation 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 $X(d_1, ..., d_n)$

$X(d_1, ..., d_n)$ 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 variables. The **init** section may be omitted, in which case the initial behaviour of the system is left unspecified.

An example of a process declaration is the following clock process, which repeatedly performs the action *tick* or displays the current time. In this example and also in later examples we assume the existence of a sort *Nat* with additional operators, representing the natural numbers.

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

Exercise 3.15 *Derive* $\partial_{\{tick\}}(\mathit{Clock}(0)) = \mathit{display}(0){\cdot}\partial_{\{tick\}}(\mathit{Clock}(0))$.

3.6 Conditionals

The process expression $p \lhd b \rhd q$ where p and q are processes, and b is a data term of sort *Bool*, behaves as p if b is equal to $\mathtt{T}$ (true) and behaves as q if b is equal to $\mathtt{F}$ (false). This operator is called the *conditional* operator, and operates as an *then_if_else* construct. This operator binds stronger than $+$ and weaker than $\cdot$.

Using the conditional operator, data can influence process behaviour. For instance, a counter that counts the number of a actions that occur, issuing a c action and resetting the internal counter after ten a's, can be described by (we omit declaring the data types):

act a, c
proc $Counter(n{:}Nat) = a{\cdot}Counter(S(n)) \lhd n < 10 \rhd c{\cdot}Counter(0)$
init $Counter(0)$

The conditional operator is characterised by axioms C1 and C2 in Table 3.4. There are no further axioms needed, because all essential properties of conditionals are provable using axioms Bool1 and Bool2.

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

Table 3.4: Axioms for conditionals

Exercise 3.16 *Specify a process that adds the elements of a list of natural numbers and prints the final result.*

Exercise 3.17 *Derive the following three equations:*

(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* $x \lhd b \rhd z = y \lhd b \rhd z.$

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 $p_1 + p_2 + p_3$. The *sum* operator $\sum_{d:D} X(d)$, with $X(d)$ a mapping from the data type D to processes, behaves as $X(d_1) + X(d_2) + \cdots$, i.e., as the possibly infinite choice between $X(d)$ for any data term d taken from D. This operator is generally used to describe a process that is reading some input over a data type. E.g.

in the following example we describe a single-place buffer, repeatedly reading a natural number n using action name r, and then delivering that value via action name s.

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

Exercise 3.18 *Specify a stack and a queue process. A stack (resp. queue) process is similar to the buffer above, 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.*

SUM1	$\sum_{d:D} x = x$
SUM3	$\sum_{d:D} X(d) = \sum_{d:D} X(d) + X(d_0) \quad (d_0 \in D)$
SUM4	$\sum_{d:D}(X(d) + Y(d)) = \sum_{d:D} X(d) + \sum_{d:D} Y(d)$
SUM5	$(\sum_{d:D} X(d))\cdot x = \sum_{d:D}(X(d)\cdot x)$
SUM6	$(\sum_{d:D} X(d)) \mathbin{\lfloor\!\lfloor} x = \sum_{d:D}(X(d) \mathbin{\lfloor\!\lfloor} x)$
SUM7	$(\sum_{d:D} X(d)) \mid x = \sum_{d:D}(X(d) \mid x)$
SUM7′	$x \mid (\sum_{d:D} X(d)) = \sum_{d:D}(x \mid X(d))$
SUM8	$\partial_H(\sum_{d:D} X(d)) = \sum_{d:D} \partial_H(X(d))$
SUM11	$(\forall d_0{\in}D\ X(d_0) = Y(d_0)) \;\Rightarrow\; \sum_{d:D} X(d) = \sum_{d:D} Y(d)$

Table 3.5: Axioms for summation

In Table 3.5 axioms for the sum operator are listed. The sum operator $\sum_{d:D} X(d)$ is a conceptually difficult operator, because it acts as a binder, just like the λ in the λ-calculus [7]. As before the variable x in the axioms may be instantiated with processes, while the process names X and Y represent functions from some data type to processes. Conforming the λ-calculus, we allow α-conversion (i.e., renaming of bound variables) in the sum operator, and do not state this explicitly. Hence, we consider the expressions $\sum_{d:D} X(d)$ and $\sum_{e:D} X(e)$ as equal (under the assumption that e and d do not occur free in $X(d)$ and $X(e)$, respectively).

When substituting a process p for a variable x, then this p does not contain free variables, so in particular it cannot contain free occurrences of the data parameter d. For example, we may not substitute the action $a(d)$ for x in the left-hand side of SUM1 in Table 3.5. So, SUM1 is a concise way of saying that if d does not appear in p, then we may omit the sum operator from $\sum_{d:D} p$. SUM3 allows one to split a summand instance from a sum. For instance the processes $\sum_{n:Nat} a(n)$ and $\sum_{n:Nat} a(n) + a(2)$ are obviously the same, as they allow an $a(n)$ action for every natural number n. This equation is a direct consequence of SUM3. SUM4 says that one may distribute the sum operator over an alternative composition, even if the sum binds a variable. This can be seen by substituting $a(d)$ for X and $b(d)$ for Y. Then the left-hand side of

SUM4 becomes $\sum_{d:D}(a(d)+b(d))$, and the right-hand side becomes $\sum_{d:D} a(d)+\sum_{d:D} b(d)$. SUM5 expresses that a process without variables can be moved outside the scope of any sum, while SUM6-8 deal with the interplay of the sum operator with the left merge, the communication merge and the encapsulation operator. SUM11 expresses that two sums are equal if all instantiations of their arguments are equal.

We show how we can eliminate a finite sum operator in favour of a finite number of sum operators. Such results always depend on the fact that a data type is defined using constructors. Therefore, we need to use induction, which makes the proof appear to be quite complex, due to the use of axioms SUM3 and SUM11. The equation that we want to derive reads:

$$\sum_{n:Nat} r(n) \lhd n \leq 2 \rhd \delta = r(0) + r(1) + r(2). \tag{3.1}$$

It is assumed that the natural numbers together with the $\leq$ relation have appropriately been defined. The result follows in a straightforward fashion using the following lemma that we prove first.

Lemma 3.19 *Let S denote the successor function on Nat. For all m:Nat:*

$$\sum_{n:Nat} X(n) = X(0) + \sum_{m:Nat} X(S(m)).$$

Proof. Using Exercise 3.3 we can split the proof into two summand inclusions.

($\subseteq$) We first prove the following statement by a case distinction on n:

$$X(n) \subseteq X(0) + \sum_{m:Nat} X(S(m)). \tag{3.2}$$

- ($n = 0$) Trivial using A3.
- ($n = S(n')$)

$$\begin{array}{l} X(0) + \sum_{m:Nat} X(S(m)) \stackrel{\text{SUM3}}{=} \\ X(0) + \sum_{m:Nat} X(S(m)) + X(S(n')) \supseteq \\ X(S(n')). \end{array}$$

As (3.2) has been proven for all natural numbers n, application of SUM11, SUM4 and SUM1 yields

$$\sum_{n:Nat} X(n) \subseteq X(0) + \sum_{m:Nat} X(S(m))$$

as had to be shown.

($\supseteq$) Using SUM3 it immediately follows for all natural numbers m that:

$$\sum_{n:Nat} X(n) \supseteq X(0) + X(S(m)).$$

So SUM11, SUM4 and SUM1 yield:

$$\sum_{n:Nat} X(n) \supseteq X(0) + \sum_{m:Nat} X(S(m)).$$

$\boxtimes$

After α-conversion, namely renaming the bound variable m into n, Lemma 3.19 takes the form

$$\sum_{n:Nat} X(n) = X(0) + \sum_{n:Nat} X(S(n)).$$

Equation (3.1) can now be derived as follows:

$$\begin{array}{rl}
 & \sum_{n:Nat} r(n) \lhd n \leq S(S(0)) \rhd \delta \\
= & (r(0) \lhd 0 \leq S(S(0)) \rhd \delta) + \sum_{n:Nat} r(S(n)) \lhd S(n) \leq S(S(0)) \rhd \delta \\
= & r(0) + (r(S(0)) \lhd S(0) \leq S(S(0)) \rhd \delta) + \sum_{n:Nat} r(S(S(n))) \lhd S(S(n)) \leq S(S(0)) \rhd \delta \\
= & r(0) + r(S(0)) + (r(S(S(0))) \lhd S(S(0)) \leq S(S(0)) \rhd \delta) \\
 & + \sum_{n:Nat} r(S(S(S(n)))) \lhd S(S(S(n))) \leq S(S(0)) \rhd \delta \\
= & r(0) + r(S(0)) + r(S(S(0))) + \delta \\
= & r(0) + r(S(0)) + r(S(S(0))).
\end{array}$$

Using the axioms on natural numbers we can prove all identities on data that we have used in the proof above; see the next exercise.

Exercise 3.20 *Derive* $0 \leq S(S(0)) = \mathtt{T}$, $S(0) \leq S(S(0)) = \mathtt{T}$ *and* $S(S(0)) \leq S(S(0)) = \mathtt{T}$. *Moreover, prove* $S(S(S(n))) \leq S(S(0)) = \mathtt{F}$ *for all natural numbers* n.

Exercise 3.21 *Derive*

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

The axiom SUM11 is one of the most tricky laws of μCRL, and it is easily abused. The universal quantifier in SUM11 expresses that this axiom may only be applied when there are no assumptions made on d. We show what can go wrong if this requirement is not respected. We saw in Exercise 3.21 that $\sum_{b:Bool} x \lhd b \rhd y = x+y$. An easy corollary of this fact is that $\sum_{b:Bool} x \lhd b \rhd y = \sum_{b:Bool} y \lhd b \rhd x$. So, as $x \lhd \mathtt{T} \rhd y = x$ and $y \lhd \mathtt{F} \rhd x = x$, it is tempting to conclude on the basis of SUM11 that $\sum_{b:Bool} x \lhd b \rhd y = x$. However, such an application of SUM11, where the data domain is partitioned and different sums are used for the different parts of the data domain, is not allowed; the example above shows that it would lead to the derivation of invalid equations.

Despite this pitfall, we quite often use sequences $\sum \ldots = \sum \ldots = \sum \ldots$, where we transform the process terms at $\ldots$ using axioms and lemmas. In such cases we carefully check that the transformation is valid for every variable bound by the sum operator, and we silently apply SUM11.

An important law is *sum elimination.* It states that the sum over a data type from which only one element can be selected can be removed. This lemma occurred for the first time in [50]. Note that we assume that we have a function eq available, reflecting equality between terms. That is, $eq(d,e) = \mathtt{T}$ if $d = e$ and $eq(d,e) = \mathtt{F}$ if $d \neq e$, for all $d, e{:}D$ (see Section 2.3).

Lemma 3.22 (Sum elimination) *Let D be a sort and $eq : D \times D \to Bool$ a function such that for all $d, e{:}D$ it holds that $eq(d,e) = \mathtt{T}$ if and only if $d = e$. Then*

$$\sum_{d:D} X(d) \triangleleft eq(d,e) \triangleright \delta = X(e).$$

Proof. According to Exercise 3.3 it suffices to prove summand inclusion in both directions.

($\subseteq$) As for each $d{:}D$ either $eq(d,e) = \mathtt{T}$ or $\neg eq(d,e) = \mathtt{T}$, it follows that:

$$X(e) = X(e) \triangleleft eq(d,e) \triangleright \delta + X(e) \triangleleft \neg eq(d,e) \triangleright \delta.$$

As we did not make any assumptions about d (except that it is of type D), we may use SUM11 in combination with SUM4, Exercise 3.17(3) and the assumption $eq(d,e) = \mathtt{T} \to d = e$ to derive:

$$\begin{array}{rcl}
\sum_{d:D} X(e) & = & \sum_{d:D}(X(e) \triangleleft eq(d,e) \triangleright \delta + X(e) \triangleleft \neg eq(d,e) \triangleright \delta) \\
& = & \sum_{d:D} X(e) \triangleleft eq(d,e) \triangleright \delta + \sum_{d:D} X(e) \triangleleft \neg eq(d,e) \triangleright \delta \\
& = & \sum_{d:D} X(d) \triangleleft eq(d,e) \triangleright \delta + \sum_{d:D} X(e) \triangleleft \neg eq(d,e) \triangleright \delta \\
& \supseteq & \sum_{d:D} X(d) \triangleleft eq(d,e) \triangleright \delta.
\end{array}$$

So using SUM1 we obtain:

$$\sum_{d:D} X(d) \triangleleft eq(d,e) \triangleright \delta \subseteq \sum_{d:D} X(e) = X(e).$$

($\supseteq$) By applying SUM3 and the assumption $eq(e,e) = \mathtt{T}$ we find:

$$\sum_{d:D} X(d) \triangleleft eq(d,e) \triangleright \delta \supseteq X(e) \triangleleft eq(e,e) \triangleright \delta = X(e).$$

$\boxtimes$

Exercise 3.23 *Show that if there is some $e{:}D$ such that $b(e) = \mathtt{T}$, then*

$$x = \sum_{d:D} x \triangleleft b(d) \triangleright \delta.$$

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 [12]. 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 to δ. Initially the bag is empty, so that one can only put a datum into the bag. The state space in Figure 3.2 depicts the behaviour of the bag over $\{0, 1\}$, with the root state placed in the leftmost uppermost corner.

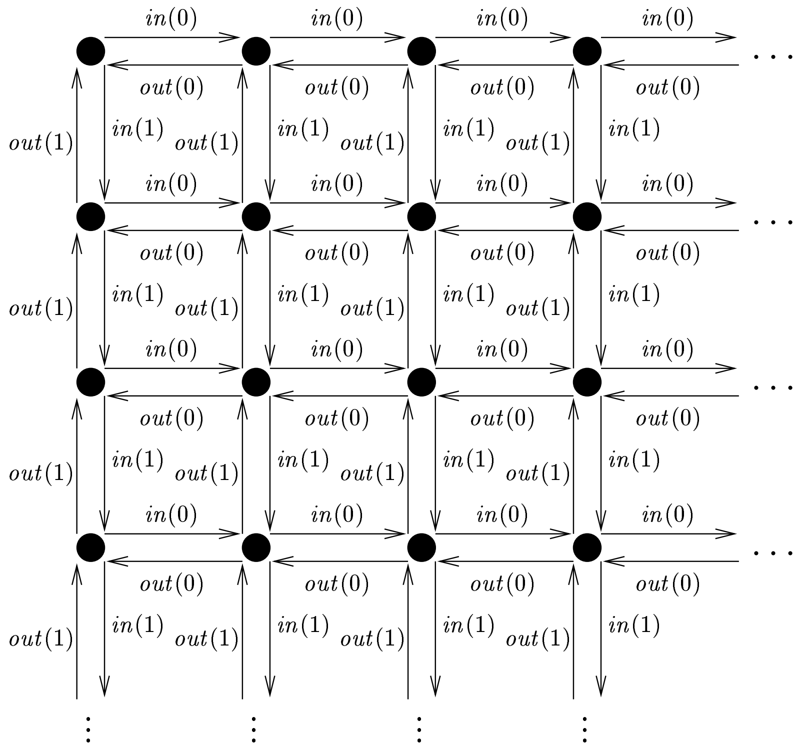


Figure 3.2: State space of the bag over $\{0, 1\}$

The bag over a data type D can be specified by the following one-liner, using the merge $\parallel$:

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

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

Exercise 3.24 *Give a process declaration of the bag over $\{d_1, d_2\}$ that does not include the three parallel operators.*

3.9 Renaming

Sometimes it is efficient to reuse a given specification with different action names. This for instance allows the definition of generic components that can be used in different configurations. The renaming operator ρ_f, with $f : \mathsf{Act} \to \mathsf{Act}$, is suited for this purpose. The subscript f signifies that the action a must be renamed to $f(a)$. The process $\rho_f(p)$ 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.

R1	$\rho_f(\delta) = \delta$
R3	$\rho_f(a(\vec{d})) = f(a)(\vec{d})$
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	$\rho_f(\sum_{d:D} X(d)) = \sum_{d:D} \rho_f(X(d))$

Table 3.6: Axioms for renaming

Chapter 4

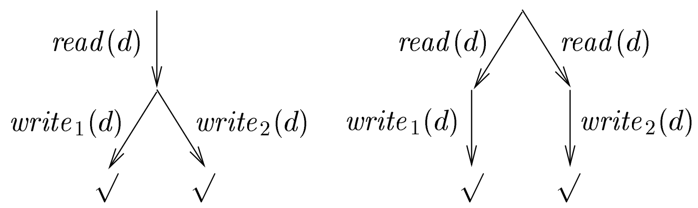
Operational Semantics

4.1 Transition rules

4.2 Bisimulation equivalence

In the process algebraic framework defined in the previous sections one can distinguish two levels. On the one hand there are the process terms, which can be manipulated by means of the axioms. Techniques from equational logic and tools such as proof checkers and 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 state spaces.

Processes have been studied since the early 60's, 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 processes are said to be equivalent if they can execute exactly the same strings of actions. However, for system behaviour this equivalence is not always satisfactory, which was shown in Example 3.4. The state spaces in this example are depicted below.



Both processes display the same strings of actions, $read(d)write_1(d)$ and $read(d)write_2(d)$, so they are trace equivalent. Still, there is a crucial distinction between the two processes, which becomes apparent if for instance disc 1 crashes. In this case the first process always saves datum d on disc 2, while the second process may get into a deadlock.

Bisimulation equivalence [6, 85, 89] discriminates more processes than trace equivalence. Namely, if two processes are *bisimilar*, then not only they can execute exactly the same strings of actions, but also they have the same branching structure. We write $\vec{d} = \vec{e}$ if they have equal length n and $d_i = e_i$ for $i = 1, \ldots, n$.

Definition 4.1 (Bisimulation) *A* bisimulation relation $\mathcal{B}$ *is a symmetric binary relation on processes such that:*

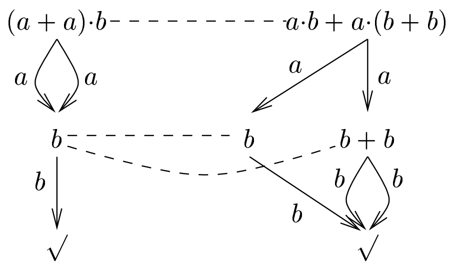
1. *if* $p\,\mathcal{B}\,q$ *and* $p \stackrel{a(\vec{d})}{\to} p'$, *then* $q \stackrel{a(\vec{e})}{\to} q'$ *with* $p'\,\mathcal{B}\,q'$ *and* $\vec{d} = \vec{e}$;

2. *if* $p\,\mathcal{B}\,q$ *and* $p \stackrel{a(\vec{d})}{\to} \surd$, *then* $q \stackrel{a(\vec{e})}{\to} \surd$ *with* $\vec{d} = \vec{e}$;

Two processes p *and* q *are* bisimilar, *denoted by* $p \,\underline{\leftrightarrow}\, q$, *if there is a bisimulation relation* $\mathcal{B}$ *such that* $p\,\mathcal{B}\,q$.

Bisimulation agrees with the equality relation on process terms in μCRL, in the sense that if two terms can be equated then their state spaces are bisimulation equivalent. In particular, similar to the equality relation, bisimulation is an equivalence relation, and if $p \,\underline{\leftrightarrow}\, q$ then $C[p] \,\underline{\leftrightarrow}\, C[q]$ for all contexts $C[]$.

Example 4.2 $(a+a)\cdot b \,\underline{\leftrightarrow}\, a\cdot b + a\cdot(b+b)$.
A bisimulation relation that relates these two basic 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:*



Exercise 4.3 *Say for each of the following pairs of basic 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 4.4 *Show that the basic 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 4.5 *Let a^1 denote a, and let a^{k+1} denote $a \cdot a^k$ for $k > 0$. Prove that $a^k \not\underline{\leftrightarrow} a^{k+1}$ for $k > 0$.*

Exercise 4.6 *Prove that $\partial_H(\rho_f(t)) \underline{\leftrightarrow} \rho_f(\partial_{f^{-1}(H)}(t))$ for all process terms t.*

Chapter 5

Hiding Internal Behaviour

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

5.1 Hiding of actions

If a customer asks a programmer to implement a product, ideally this customer is able to provide the external behaviour of the desired program. That is, he or she is able to tell what should be 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 computation steps of the program.

Hiding is an important means to analyse communicating systems. It means that certain actions are made invisible, such that the relationship between the remaining actions becomes more clear. The hidden action is denoted τ. It represents an action that can take place in a system, but that cannot be observed directly. The τ does not carry any data parameters. The hidden action is meant for analysis purposes, and hardly ever used in specifications, as it is very uncommon to specify that something unobservable must happen.

A typical identity characterising τ is $a{\cdot}\tau{\cdot}p = a{\cdot}p$, with a an action and p a process term. It says that it is by observation impossible to tell whether or not hidden actions happen after the a. Sometimes, the presence of hidden actions can be observed, due to the context in which they appear. E.g. $a + \tau{\cdot}b \neq a + b$, as the left-hand side can silently execute the τ, after which it only offers a b action, whereas the right-hand side can always do an a. The difference between the two processes can be observed by insisting in both cases that the a happens. This is always successful in $a + b$, but may lead to a deadlock in $a + \tau{\cdot}b$.

Axioms B1,2 in Table 5.1 are the characterising laws for the τ. They express that a τ is invisible if it does not lose any possible behaviours (cf. Section 5.4). The hidden action τ does not communicate with any actions. In order to make

B1	$x \cdot \tau = x$	
B2	$x \cdot (\tau \cdot (y+z) + y) = x \cdot (y+z)$	
TID	$\tau_I(\delta) = \delta$	
TI1	$\tau_I(a(\vec{d})) = a(\vec{d})$	if $a \notin I$
TI2	$\tau_I(a(\vec{d})) = \tau$	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	$\tau_I(\sum_{d:D} X(d)) = \sum_{d:D} \tau_I(X(d))$	
R2	$\rho_f(\tau) = \tau$	

Table 5.1: Axioms for hidden actions and hiding

actions hidden, the hiding operator τ_I, for subsets I of Act, is introduced, where I is a set of action names. The process $\tau_I(p)$ behaves as the process p, except that all actions with action names in I are renamed to τ. This is characterised by axioms TID and TI1-4.

Exercise 5.1 *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* $\tau \cdot (\tau \cdot x + y) = \tau \cdot x$.

An equation such as $X = \tau \cdot X$ does not specify a unique process; each process term $\tau \cdot t$ constitutes a 'solution' for X. The equations in a process declaration are *guarded* if each expression $Y(d_1, \ldots, d_m)$ occurs in a context of the form $C_1[a \cdot C_2[]]$ with $a \in \mathsf{Act}$ (i.e., $a \neq \tau$).

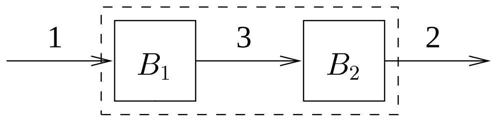
5.2 Summary

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, communication operators ($\|$, $\lfloor\!\lfloor$, $|$) to express parallelism, deadlock and encapsulation to force actions into communication, τ and hiding to make internal computations invisible, process declarations to express recursion, conditional and summation to mix data with processes, and action renaming to support reuse of specifications.

In particular, the framework is suitable for the specification and verification of network protocols. For such a verification, the desired external behaviour of the protocol is represented in the form of a process term that is built from process declarations 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 actions of the implementation are forced into communication using an encapsulation operator, and the internal communication actions are made invisible using a hiding operator, so that only the input/output relation of the implementation remains. Finally, the two process terms representing the specification and the implementation are equated by means of the axioms.

5.3 An example: two one-bit buffers in sequence

To give a more extensive 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 3.8, $s_3(d) \mid r_3(d) = c_3(d)$, while all other communications between actions result to δ. The buffers B_1 and B_2 are defined by the process declaration

act $r_1, r_2, r_3, s_1, s_2, s_3, c_3 : \Delta$
proc $B_1 = \sum_{d:\Delta} r_1(d) \cdot s_3(d) \cdot B_1$
$B_2 = \sum_{d:\Delta} r_3(d) \cdot s_2(d) \cdot B_2$

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

init $\tau_{\{c_3(d) \mid d \in \Delta\}}(\partial_{\{s_3(d), r_3(d) \mid d \in \Delta\}}(B_2 \parallel B_1))$

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

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

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 d' from channel 1 and proceed to the state $Z(d,d')$ where the queue contains d and d', or send datum d into channel 2 and proceed to the state X where the queue is empty. Finally, in state $Z(d,d')$ the queue is full, so that it can only send datum d into channel 2 and proceed to the state $Y(d')$ where it contains d'.

Exercise 5.2 *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.*

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 atomic actions are abbreviated by omitting the suffix (0). First we expand $\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)$; in each derivation step, the subterms that are reduced are underlined. Since actions r_3 and r_1 do not communicate, the axioms for the parallel operators together with the process declarations yield:

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

So the axioms for deadlock and encapsulation yield:

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

Summarising, we have derived

$$
\partial_{\{s_3,r_3\}}(B_2 \parallel B_1) = r_1\cdot\partial_{\{s_3,r_3\}}((s_3\cdot B_1) \parallel B_2). \tag{5.1}
$$

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 process declarations 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

$$\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2) = c_3{\cdot}\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2)). \tag{5.2}$$

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 process declarations,

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

So by the axioms for encapsulation,

$$\begin{array}{rl} & \partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2)) \\ = & r_1{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2)) + s_2{\cdot}\partial_{\{s_3,r_3\}}(B_2 \parallel B_1). \end{array} \tag{5.3}$$

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 process declarations,

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

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 yields $B_2 \parallel (s_3{\cdot}B_1) = (s_3{\cdot}B_1) \parallel B_2$, so

$$\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2)) = s_2{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2). \tag{5.4}$$

Summarising, we have algebraically derived the following relations:

$$\begin{tikzcd}
\partial_{\{s_3,r_3\}}(B_2 \| B_1) \arrow[d, "r_1"] \\
\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \| B_2) \arrow[d, "c_3"] \\
\partial_{\{s_3,r_3\}}(B_1 \| (s_2{\cdot}B_2)) \arrow[d, "r_1"] \arrow[uu, bend left, "s_2"] \\
\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \| (s_2{\cdot}B_2)) \arrow[uu, bend right, "s_2"']
\end{tikzcd}$$

Equations (5.1) and (5.2) together with the axioms for τ and hiding yield:

$$
\begin{array}{rcl}
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)) & \stackrel{(5.1)}{=} & \tau_{\{c_3\}}(r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2)) \\
& \stackrel{\mathrm{TI1,4}}{=} & r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2)) \\
& \stackrel{(5.2)}{=} & r_1 \cdot \tau_{\{c_3\}}(c_3 \cdot \partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2))) \\
& \stackrel{\mathrm{TI2,4}}{=} & r_1 \cdot \tau \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2))) \\
& \stackrel{\mathrm{B1}}{=} & r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2))).
\end{array}
$$

Moreover, equation (5.3) together with the axioms for hiding yield:

$$
\begin{array}{rl}
& \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2))) \\
\stackrel{(5.3)}{=} & \tau_{\{c_3\}}(r_1 \cdot \partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2)) + s_2 \cdot \partial_{\{s_3,r_3\}}(B_2 \parallel B_1)) \\
\stackrel{\mathrm{TI1,3,4}}{=} & r_1 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2))) + s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)).
\end{array}
$$

Finally, equations (5.2) and (5.4) together with the axioms for τ and hiding yield:

$$
\begin{array}{rcl}
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel (s_2 \cdot B_2))) & \stackrel{(5.4)}{=} & \tau_{\{c_3\}}(s_2 \cdot \partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2)) \\
& \stackrel{\mathrm{TI1,4}}{=} & s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3 \cdot B_1) \parallel B_2)) \\
& \stackrel{(5.2)}{=} & s_2 \cdot \tau_{\{c_3\}}(c_3 \cdot \partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2))) \\
& \stackrel{\mathrm{TI2,4}}{=} & s_2 \cdot \tau \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2))) \\
& \stackrel{\mathrm{B1}}{=} & s_2 \cdot \tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 \cdot B_2))).
\end{array}
$$

The last three derivations together show that

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

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

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

Exercise 5.3 *Use the renaming operator to extract the buffer B_2 in Exercise 5.2 from the process declaration of B_1.*

5.4 Branching bisimulation equivalence

While on the level of process terms with hidden actions we have defined an equality relation, we have not yet introduced a way to relate state spaces. In this section we define the notion of *rooted branching bisimulation equivalence* on state spaces. This equivalence agrees with the equality relation on process terms, in the sense that if two terms can be equated then their state spaces are rooted branching bisimulation equivalent. Furthermore, rooted branching bisimulation equivalence agrees with bisimulation equivalence, if the τ is taken to be a concrete visible action.

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 s and t are equivalent, then $C[s]$ and $C[t]$ are equivalent for all contexts $C[]$.

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

which τ's are inert?

The obvious answer to this question, "all τ's are inert", turns out to be incorrect. Namely, this answer would produce an equivalence relation that, on the level of process terms, is not preserved under contexts.

As an example of an action τ that is not inert, consider the process terms $a + \tau{\cdot}\delta$ and a. If the τ 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 in the first term is not inert. In order to describe this case more vividly, we give an example.

Example 5.4 *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 actions via channels 2 and 3, because this would cover up an important problem of the protocol.*

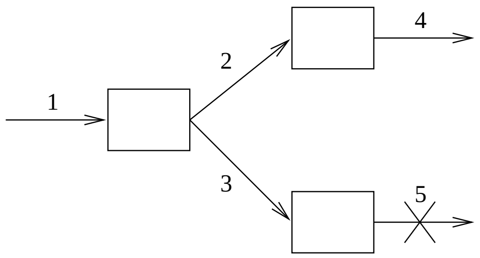


Figure 5.1: Protocol with a malfunctioning channel

The system, which is depicted in Figure 5.1, is described by the process term

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

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. Abstracting away from the internal 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 τ's in this 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 steps. 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 in the first process term is inert: after execution of this τ it is still possible to execute a. In general, process terms $s + \tau\cdot(s + t)$ and $s + t$ are equivalent for all process terms s and t. By contrast, in a process term such as $a + \tau\cdot b$ the τ is not inert, since execution of this τ means losing the option to execute a.

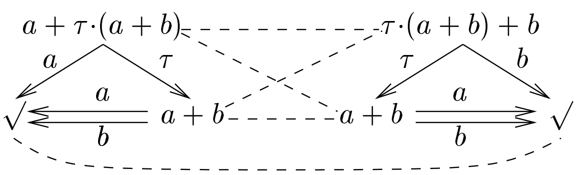
The intuition above is formalised in the notion of *branching bisimulation equivalence* [47]. Let the processes p and q be branching bisimilar. If $p \xrightarrow{\tau} p'$, then q does not have to simulate this τ-transition if it is inert, meaning that p' and q are branching bisimilar. Moreover, a non-inert transition $p \xrightarrow{a(\vec{d})} p'$ need not be simulated by q immediately, but only after a number of inert τ-transitions: $q \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_0 \xrightarrow{a(\vec{d})} q'$, where p and q_0 are branching bisimilar (to ensure that the τ-transitions are inert) and p' and q' are branching bisimilar (so that $p \xrightarrow{a(\vec{d})} p'$ is simulated by $q_0 \xrightarrow{a(\vec{d})} q'$). A special termination predicate $\downarrow$ is needed in order to relate branching bisimilar process terms such as $a\tau$ and a. These intuitions are formalised in the following definition.

Definition 5.5 (Branching bisimulation) *Assume a special termination predicate $\downarrow$, and let $\surd$ represent a state with $\surd\downarrow$.* A branching bisimulation relation *$\mathcal{B}$ is a symmetric binary relation on the collection of processes such that*

1. *if* $p\,\mathcal{B}\,q$ *and* $p \stackrel{a(\vec{d})}{\to} p'$, *then*
   - *either* $a = \tau$ *and* $p'\,\mathcal{B}\,q$;
   - *or there is a sequence of (zero or more)* τ*-transitions* $q \stackrel{\tau}{\to} \cdots \stackrel{\tau}{\to} q_0$ *such that* $p\,\mathcal{B}\,q_0$ *and* $q_0 \stackrel{a(\vec{e})}{\to} q'$ *with* $p'\,\mathcal{B}\,q'$ *and* $\vec{d} = \vec{e}$;
2. *if* $p\,\mathcal{B}\,q$ *and* $p\downarrow$, *then there is a sequence of (zero or more)* τ*-transitions* $q \stackrel{\tau}{\to} \cdots \stackrel{\tau}{\to} q_0$ *such that* $q_0\downarrow$.

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

Example 5.6 $a + \tau{\cdot}(a + b) \underline{\leftrightarrow}_b\, \tau{\cdot}(a + b) + b$.
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:*

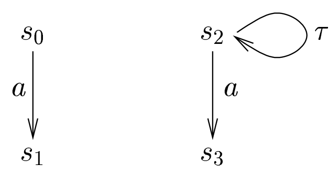


It is left to the reader to verify that this relation satisfies the requirements of a branching bisimulation.

Exercise 5.7 *Give branching bisimulation relations to prove that the process terms* a, $a{\cdot}\tau$, *and* $\tau{\cdot}a$ *are branching bisimilar.*

Exercise 5.8 *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.*

Branching bisimulation equivalence 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 the chance that no exit from the τ-loop is ever chosen is zero. For example, it is not hard to see that the states s_0 and s_2 in the two process graphs below are branching bisimilar.



Exercise 5.9 *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* $s \stackrel{\tau}{\to} \cdots \stackrel{\tau}{\to} s'$ *and* $s' \stackrel{\tau}{\to} \cdots \stackrel{\tau}{\to} s$. *Prove that* s *and* s' *are branching bisimilar.*

Groote and Vaandrager [63] presented an algorithm to decide whether two finite-state processes are branching bisimilar, which has worst-case time complexity $O(mn)$, where n is the number of states and m the number of transitions in the input graph; see Section 7.1.

Branching bisimilarity is an equivalence relation; see [9]. Branching bisimulation equivalence, however, is not yet a congruence. For example, $\tau{\cdot}a$ and a are branching bisimilar (see Exercise 5.7), 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 processes are considered equivalent if they can simulate each other's initial transitions, such that the resulting processes are branching bisimilar. This leads to the notion of *rooted branching bisimulation equivalence*, which is presented below.

Definition 5.10 (Rooted branching bisimulation) *Assume the termination predicate $\downarrow$, and let $\surd$ represent a state with $\surd\downarrow$. A* rooted branching bisimulation relation *$\mathcal{B}$ is a symmetric binary relation on processes such that:*

1. *if $p\,\mathcal{B}\,q$ and $p \stackrel{a(\vec{d})}{\rightarrow} p'$, then $q \stackrel{a(\vec{e})}{\rightarrow} q'$ with $p' \underline{\leftrightarrow}_b\, q'$ and $\vec{d} = \vec{e}$;*
2. *if $p\,\mathcal{B}\,q$ and $p\downarrow$, then $q\downarrow$;*

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

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

Exercise 5.11 *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* $(b + a){\cdot}(d + c) + a{\cdot}(c + d)$;

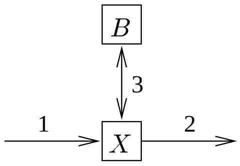
(3) $\tau{\cdot}(b + a) + \tau{\cdot}(a + b)$ *and* $a + b$;

(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 5.12 *Data elements (from a collection Δ) can be received by a one-place buffer X via channel 1. An incoming datum is either sent on via channel 2, or stored in a one-place buffer B via channel 3.*



X and B are defined by the following μCRL specification:

act $r_1, s_2, s_3, r_3, c_3{:}\Delta$
comm $s_3 \,|\, r_3 = c_3$
proc $X = \sum_{d:\Delta}(r_1(d) + r_3(d))\cdot(s_2(d) + s_3(d))\cdot X$
$B = \sum_{d:\Delta} r_3(d)\cdot s_3(d)\cdot B$

Let t denote $\partial_{\{r_3(d), s_3(d) | d \in \Delta\}}(X \| B)$, and let Δ consist of $\{d_1, d_2\}$.

(1) *Draw the state space of t.*

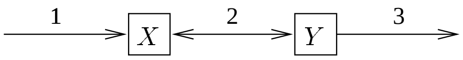
(2) *Are data elements read via channel 1 and sent via channel 2 in the same order?*

(3) *Does $\partial_{\{s_2(d_1)\}}(t)$ contain a deadlock? If yes, then give an execution trace to a deadlock state.*

(4) *Draw the state space of $\tau_{\{c_3(d_1), c_3(d_2)\}}(t)$. Distinguish the inert τ-transitions from the non-inert ones.*

Draw the state space of $\tau_{\{c_3(d_1), c_3(d_2)\}}(t)$ after minimisation modulo branching bisimulation equivalence.

Exercise 5.13 *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 $X = \sum_{d:\Delta}(r_1(d) + r_2(d))\cdot s_2(d)\cdot X$
$Y = \sum_{d:\Delta} r_2(d)\cdot(s_3(d) + s_2(d))\cdot Y$

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

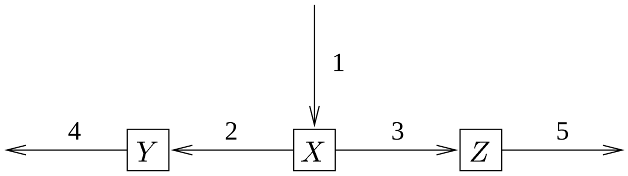
(1) *Draw the state space of t.*

(2) Are data elements read via channel 1 and sent via channel 3 in the same order?

(3) Does $\partial_{\{s_3(d_1)\}}(t)$ *contain a deadlock? If yes, then give an execution trace to a deadlock state.*

(4) Draw the state space of $\tau_{\{c_2(d_1),c_2(d_2)\}}(t)$ *after minimisation modulo branching bisimulation equivalence.*

Exercise 5.14 *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 dataum 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 *en* Z *in* μ*CRL, together with de action declaration* **act***, the communication declaration* **comm** *and the initial declaration* **init***.*

(2) Let Δ *consist of a single element. Draw the state space that belongs to the initial declaration.*

Chapter 6

Protocol Specifications

Much time and effort is expended in the development of new techniques for the description and analysis of distributed systems; however, many of these techniques are never widely used, due to a sharp learning curve required to adopt them; many verification techniques have complex theoretical underpinnings, and require sophisticated mathematical skills to apply them. Case studies therefore have a valuable role to play both in promoting and demonstrating particular verification techniques, and providing practical examples of their application. In this chapter we present specifications of a number of protocols in μCRL. In Chapter 10 we will verify the correctness of some of these protocols.

6.1 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, army one has to acknowledge the reception of this acknowledgement, et cetera.

The alternating bit protocol (ABP) [8, 82] ensures successful transmission of data through a lossy 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. The protocol is depicted in Figure 6.1.

Data elements $d_1, d_2, d_3, \ldots$ from a nonempty set Δ are communicated between a Sender and a Receiver. If the Sender reads a datum from channel A, then this datum is communicated to the Receiver, which sends the datum into channel D. However, the channels between the Sender and the Receiver are lossy,

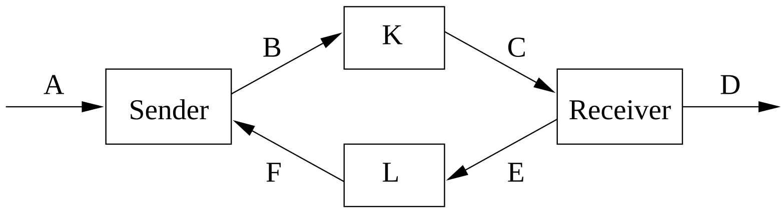


Figure 6.1: Alternating bit protocol

so that a message that is communicated through these channels can be turned into an error message $\perp$. Therefore, every time the Receiver receives a message, it sends an acknowledgement to the Sender, which can also be corrupted.

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 $\perp$. 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 $1-b$, et cetera. 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 it acknowledges reception of a datum or of an error message.

We give a μCRL specification of the ABP. This specification displays the desired external behaviour; that is, 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 process term is a solution for 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".

First, we specify the Sender in the state that it is going to send out a datum with the bit b attached to it, represented by the process name $S(b)$ for $b \in \{0, 1\}$:

$$\begin{array}{lcl} S(b) & = & \sum_{d:\Delta} r_{\mathrm{A}}(d){\cdot}s_{\mathrm{B}}(d,b){\cdot}T(d,b) \\ T(d,b) & = & r_{\mathrm{F}}(b){\cdot}S(1-b) \;+\; (r_{\mathrm{F}}(1-b) + r_{\mathrm{F}}(\perp)){\cdot}s_{\mathrm{B}}(d,b){\cdot}T(d,b) \end{array}$$

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 $\perp$, 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 name $R(b)$ for $b \in \{0,1\}$:

$$R(b) \quad = \quad \sum_{d:\Delta} r_{\mathrm{C}}(d,b)\cdot s_{\mathrm{D}}(d)\cdot s_{\mathrm{E}}(b)\cdot R(1-b) \;+\; (\textstyle\sum_{d:\Delta} r_{\mathrm{C}}(d,1-b) + r_{\mathrm{C}}(\perp))\cdot s_{\mathrm{E}}(1-b)\cdot R(b)$$

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 $\perp$ 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 $R(b)$.

The processes K and L express that messages between the Sender and the Receiver, and vice versa, may be corrupted.

$$\begin{aligned} K &= \sum_{d:\Delta}\sum_{b:\{0,1\}} r_{\mathrm{B}}(d,b)\cdot(j\cdot s_{\mathrm{C}}(d,b) + j\cdot s_{\mathrm{C}}(\perp))\cdot K \\ L &= \sum_{b:\{0,1\}} r_{\mathrm{E}}(b)\cdot(j\cdot s_{\mathrm{F}}(b) + j\cdot s_{\mathrm{F}}(\perp))\cdot L \end{aligned}$$

The action j expresses the nondeterministic choice whether or not a message is corrupted.

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

$$\begin{aligned} s_{\mathrm{B}}(d,b)\,|\,r_{\mathrm{B}}(d,b) &= c_{\mathrm{B}}(d,b) \\ s_{\mathrm{B}}(\perp)\,|\,r_{\mathrm{B}}(\perp) &= c_{\mathrm{B}}(\perp) \\ s_{\mathrm{D}}(b)\,|\,r_{\mathrm{D}}(b) &= c_{\mathrm{D}}(b) \\ s_{\mathrm{D}}(\perp)\,|\,r_{\mathrm{D}}(\perp) &= c_{\mathrm{D}}(\perp) \end{aligned}$$

for $d \in \Delta$ and $b \in \{0,1\}$. All other communications between atomic actions result to δ.

The desired concurrent system is obtained by putting $S(0)$, $R(0)$, K and L in parallel, encapsulating send and read actions over internal channels, and abstracting away from communication actions over these channels and from the action j. That is, the ABP is expressed by the process term

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

with H consisting of all send and read actions over B, C, E and F, and I consisting of all communication actions over B, C, E and F together with j. The state space of $\partial_H(S(0) \parallel R(0) \parallel K \parallel L)$ is depicted in Figure 6.2.

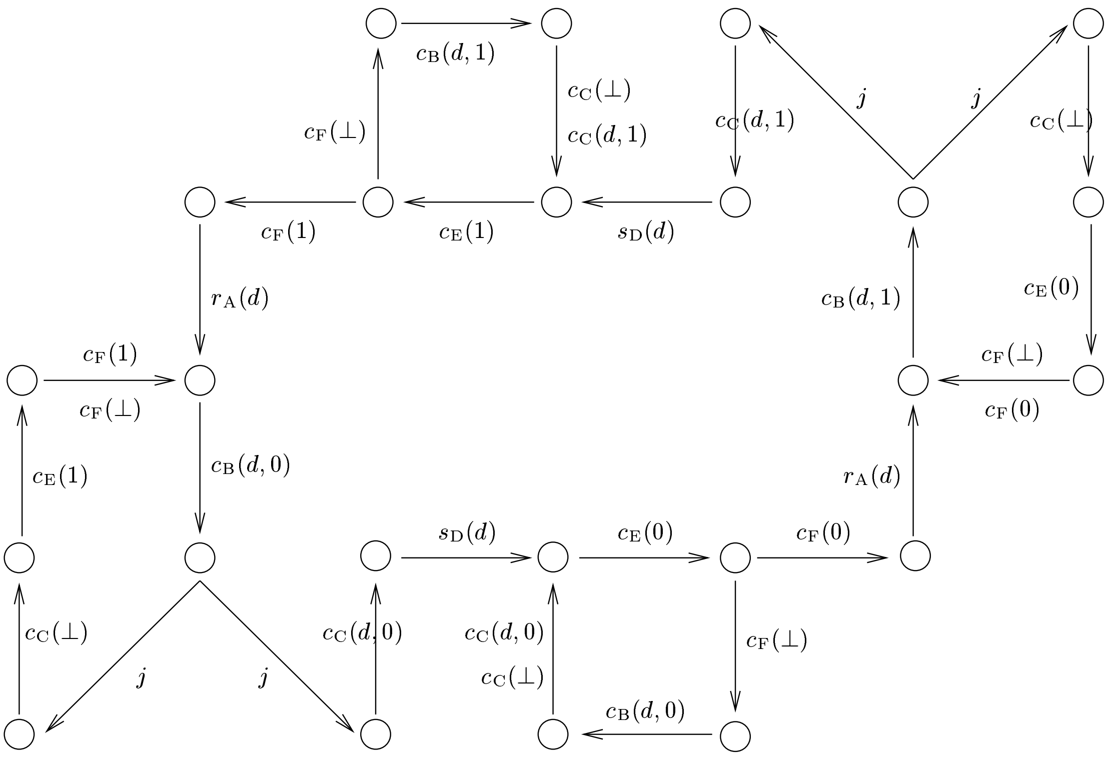


Figure 6.2: State space of $\partial_H(S(0) \parallel R(0) \parallel K \parallel L)$.

Exercise 6.1 *Give a μCRL specification of the ABP together with its data types. Use the μCRL toolset to analyse this specification, for $\Delta = \{d_1, d_2\}$.*

Exercise 6.2 *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.*

6.2 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. 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 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 implicitly sets the timer T_1; if the RC receives an acknowledgement, then it implicitly 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, the 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.

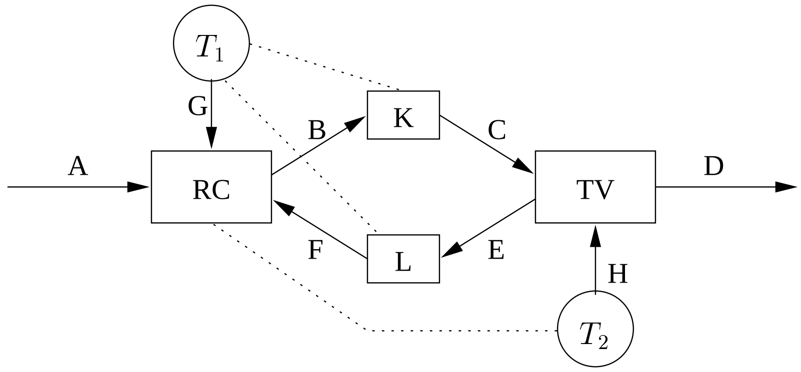


Figure 6.3: Bounded retransmission protocol

The BRP is depicted in Figure 6.3. Note that the medium between the RC and the TV is represented by two separate entities K and L, which can pass on a datum or lose it at random. The dotted lines between these entities and the 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.

Groote and van de Pol [55] specified the BRP in process algebra, and verified that the protocol exhibits the required external behaviour. Alternative specifications and verifications of the BRP can be found in [1, 36, 55, 65].

First, we give an informal description of the process algebra specification for the BRP, and explain its required external behaviour. Next, we present the formal specification, and derive algebraically its actual external behaviour. Our specification 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 implicitly sets the timer T_1, and moreover it sets a counter at zero to keep track of the number of failed 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 the timer T_1, and gives the counter the value zero.

2. The RC receives a time-out from the timer T_1 via channel G. Then it sends out the pair (d_i, b) again, sets the timer T_1, 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 and 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 the timer T_2 via channel H, signalling 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.

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, the 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, the 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 Figure 6.4. 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.

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 the process declaration that formally specify the BRP in process algebra. This process declaration exhibits the external behaviour depicted in Figure 6.4, intertwined with non-invisible τ-transitions.

In order to simplify the specification, we assume that the data packets that reach the RC via channel A have length ≥ 2, and that $max \geq 2$. The process declaration uses the following data parameters and functions.

- d ranges over a nonempty data set Δ, and ℓ ranges over the set Λ of lists of data of length ≥ 2. $head(\ell)$ represents the first element of the list ℓ, and $tail(\ell)$ represents the remaining list: $head(d_1, \ldots, d_N) = d_1$ and $tail(d_1, \ldots, d_N) = (d_2, \ldots, d_N)$.

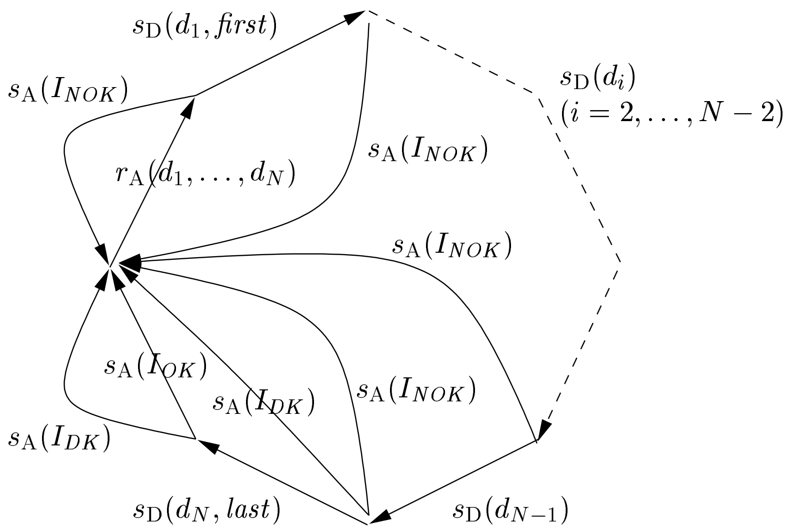


Figure 6.4: External behaviour of the BRP

- b ranges over $\{0, 1\}$, while n ranges over $\{0, \ldots, \mathit{max}\}$, where max is the maximum number of attempts that the RC is allowed to undertake to transmit a datum to the TV.

- 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 process name X:

$$
\begin{array}{llcl}
 & X & = & \sum_{\ell:\Lambda} r_{\rm A}(\ell) \cdot Y(\ell, 0, 0) \\
 & Y(\ell, b, n) & = & s_{\rm B}(\mathit{head}(\ell), b) \cdot Z(\ell, b, n) \\
 & Y(d, b, n) & = & s_{\rm B}(d, b, \mathit{last}) \cdot Z(d, b, n) \\
(n < \mathit{max}) & Z(\ell, b, n) & = & r_{\rm F}(\mathit{ack}) \cdot Y(\mathit{tail}(\ell), 1-b, 0) \\
 & & + & r_{\rm G}(\mathit{to}) \cdot Y(\ell, b, S(n)) \\
 & Z(\ell, b, \mathit{max}) & = & r_{\rm F}(\mathit{ack}) \cdot Y(\mathit{tail}(\ell), 1-b, 0) \\
 & & + & r_{\rm G}(\mathit{to}) \cdot s_{\rm A}(I_{NOK}) \cdot s_{\rm H}(\mathit{to}) \cdot X \\
(n < \mathit{max}) & Z(d, b, n) & = & r_{\rm F}(\mathit{ack}) \cdot s_{\rm A}(I_{OK}) \cdot X \\
 & & + & r_{\rm G}(\mathit{to}) \cdot Y(d, b, S(n)) \\
 & Z(d, b, \mathit{max}) & = & r_{\rm F}(\mathit{ack}) \cdot s_{\rm A}(I_{OK}) \cdot X \\
 & & + & r_{\rm G}(\mathit{to}) \cdot s_{\rm A}(I_{DK}) \cdot s_{\rm H}(\mathit{to}) \cdot X
\end{array}
$$

The intuition behind these process declarations is as follows. Let l range over lists of data of length > 0.

- In state X, the RC waits until it receives a data packet ℓ via channel A, after which it proceeds to $Y(\ell, 0, 0)$. The first zero represents the bit that is going to be attached to $head(\ell)$, while the second zero represents the counter.

- In state $Y(l,b,n)$, the RC attempts to send the head of list l to the TV via channel B, with bit b attached to it. If l consists of a single datum, then moreover a label *last* is attached to this message. The counter n registers the number of unsuccessful attempts to send the head of l to the TV.
- In state $Z(l,b,n)$, 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 l consists of two or more data elements, then it proceeds to send the head of $tail(l)$ to the TV, with bit $1-b$ attached to it and the counter starting at zero. If l 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 the timer T_1. If $n < max$, then it sends the pair $(head(l), b)$ to the TV again, with the counter increased by one. If $n = max$, then it concludes that transmission of the data packet was unsuccessful (if l consists of two or more elements) or may have been unsuccessful (if l 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 the 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 root state is represented by the process name V:

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

The intuition behind these process declarations 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 such a message, then it sends the datum 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.

Finally, the TV may receive a time-out from the 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 both the datum d and the message I_{OK} 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 the timer T_2 via channel H, which signals that the RC has given up transmission of the data packet. Then the TV sends the error message I_{NOK} into channel D and proceeds to state V.

Finally, we specify the mediums K and L:

$$\begin{array}{rcl} K & = & \sum_{d:\Delta}\sum_{b:\{0,1\}}\{r_{\mathrm{B}}(d,b)\cdot(s_{\mathrm{C}}(d,b)+s_{\mathrm{G}}(to))\cdot K \\ & & \qquad\qquad + r_{\mathrm{B}}(d,b,last)\cdot(s_{\mathrm{C}}(d,b,last)+s_{\mathrm{G}}(to))\cdot K\} \\ L & = & r_{\mathrm{E}}(ack)\cdot(s_{\mathrm{F}}(ack)+s_{\mathrm{G}}(to))\cdot L \end{array}$$

The intuition behind these process declarations 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 the 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 the 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.

The process term $\tau_I(\partial_H(V \parallel X \parallel K \parallel L))$ exhibits the required external behaviour (see Figure 6.4), intertwined with non-intert τ-transitions.

Exercise 6.3 *Give a μCRL specification of the BRP and its data types, where the lists of data that are being transmitted contain three elements, and the maximum number of retries is four.*

6.3 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 [28] (see also [100]), a Sender can send out data elements without waiting for acknowledgements. See [15, 23] for specifications and verifications of sliding window protocols in process algebra.

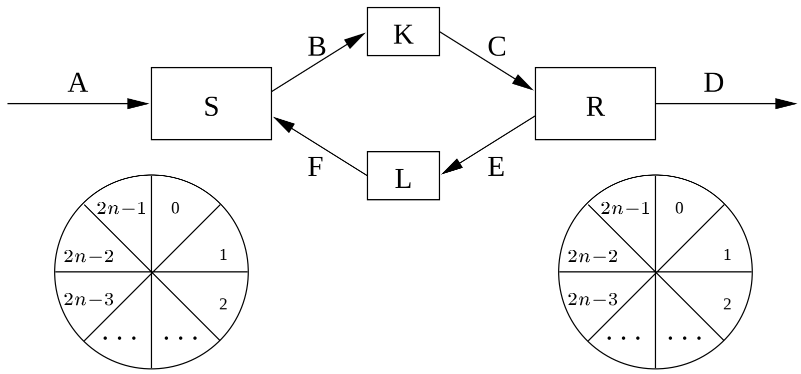


Figure 6.5: Sliding window protocol

Again we assume that the elements from a nonempty data domain Δ that are sent from 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 $n > 0$; see Figure 6.5. 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*.

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:

$$[] :\rightarrow \mathit{Buffer}$$
$$\mathit{in} : \Delta \times \mathit{Nat} \times \mathit{Buffer} \rightarrow \mathit{Buffer}$$

The nature of positions in buffers is firmly linked with so-called *modulo arithmetic* (see, e.g., [37]). Two natural numbers are considered equal *modulo m* if their difference is divisible by m. It is not hard to see that this defines an equivalence relation on natural numbers, with equivalence classes $0, \ldots, m-1$.

In the remainder of this section, $2n$ is a special parameter representing the size of the buffer. In the μCRL specification of the SWP, one can declare $2n :\rightarrow Nat$, and this function symbol of arity zero can be instantiated with its desired value in the algebraic specification of the natural numbers.

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, where $if : Bool \times Nat \times Nat \rightarrow Nat$ is the if-then-else function from Exercise 2.8 and $eq : Nat \times Nat \rightarrow Bool$ is the equality function on natural numbers from Section 2.3:

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

In the SWP, the Sender reads data elements from channel A and stores them in the window of 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 $2n$. At any time, the Sender can send a datum from its window into channel B, paired with its position number. 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 (d, k) is the first pair in its window, then the Receiver can send datum d into channel D, remove the pair (d, k) from its buffer, and slide its window beyond position k. The Receiver can also 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; if previously the Receiver sent an acknowledgement ℓ, and it received data elements for positions ℓ up to $\ell+m$ (modulo $2n$), then it can send the acknowledgement $\ell+m+1$. Upon reception of such an acknowledgement via channel F, the Sender eliminates all acknowledged pairs from its buffer, and slides 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.

We proceed to specify the SWP in process algebra. We assume that the sending and receiving buffers have predefined size $2n$, and that the sliding windows in these buffers are restricted to n positions.

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, $retrieve(k, t)$ produces the datum that resides at position k in buffer t (if this position is occupied), and $test(k, t)$ produces `T` if and only if position k in t is occupied. These four functions are defined by:

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

The second equation of *remove* implicitly assumes that there is at most one datum at each position in a buffer. For example, a buffer must not be of the form $\mathit{in}(d,k,\mathit{in}(d',k,[]))$, because $\mathit{remove}(k,\mathit{in}(d,k,\mathit{in}(d',k,[])))$ would produce the erroneous result $\mathit{in}(d',k,[])$. In order to support the assumption that there is no overloading of positions in buffers, position k in buffer t needs to be emptied in the right-hand side of the equation for $\mathit{add}(d,k,t)$. The use of the function *add*, instead of *in*, in the specification of the Receiver will be essential to obtain a finite state space. Namely, otherwise some pair (d,ℓ) could be added to the receiving window an unbounded number of times.

$\mathit{release}(k,\ell,t)$ 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:

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

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 $\mathit{release}(\mathit{succmod}(k),\ell,\mathit{remove}(k,t))$ in its right-hand side. This problem can be solved by adapting the equation for *release* to:

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

In the specification of the SWP below, for clarity of presentation, two of the conditions $\lhd\_\rhd$ are formulated in natural language. For algebraic formulations of these conditions, the reader is referred to the exercises. If k and ℓ are in $\{0,\ldots,2n-1\}$, then 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 $X(\mathit{first\text{-}in},\mathit{first\text{-}empty},\mathit{buffer})$, where *buffer* represents the sending buffer of size $2n$, *first-in* the first position in the window of *buffer*, and *first-empty* the first empty position in (or just outside) the window of *buffer*.

$$\begin{array}{ll}
& X(\mathit{first\text{-}in},\mathit{first\text{-}empty},\mathit{buffer}) \\
= & \sum_{d:\Delta} r_{\mathrm{A}}(d)\cdot X(\mathit{first\text{-}in},\mathit{succmod}(\mathit{first\text{-}empty}),\mathit{add}(d,\mathit{first\text{-}empty},\mathit{buffer})) \\
& \qquad \lhd \text{“the range from } \mathit{first\text{-}in} \text{ to } \mathit{first\text{-}empty} \text{ does not exceed } n\text{”} \rhd \delta \\
+ & \sum_{k:\mathit{Nat}} s_{\mathrm{B}}(\mathit{retrieve}(k,\mathit{buffer}),k)\cdot X(\mathit{first\text{-}in},\mathit{first\text{-}empty},\mathit{buffer}) \\
& \qquad \lhd \mathit{test}(k,\mathit{buffer}) \rhd \delta \\
+ & \sum_{k:\mathit{Nat}} r_{\mathrm{F}}(k)\cdot X(k,\mathit{first\text{-}empty},\mathit{release}(\mathit{first\text{-}in},k,\mathit{buffer}))
\end{array}$$

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 $2n$:

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

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:

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

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

$$\begin{array}{cl} & Y(\mathit{first\text{-}in}, \mathit{buffer}) \\ = & \sum_{d:\Delta}\sum_{k:\mathit{Nat}} r_{\mathrm{C}}(d,k)\cdot(Y(\mathit{first\text{-}in}, \mathit{add}(d,k,\mathit{buffer})) \\ & \qquad \lhd \text{“the range from } \mathit{first\text{-}in} \text{ to } k \text{ does not exceed } n\text{”} \rhd Y(\mathit{first\text{-}in}, \mathit{buffer})) \\ + & s_{\mathrm{D}}(\mathit{retrieve}(\mathit{first\text{-}in}, \mathit{buffer}))\cdot Y(\mathit{succmod}(\mathit{first\text{-}in}), \mathit{remove}(\mathit{first\text{-}in}, \mathit{buffer})) \\ & \qquad \lhd \mathit{test}(\mathit{first\text{-}in}, \mathit{buffer}) \rhd \delta \\ + & s_{\mathrm{E}}(\mathit{next\text{-}empty}(\mathit{first\text{-}in}, \mathit{buffer}))\cdot Y(\mathit{first\text{-}in}, \mathit{buffer}) \end{array}$$

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

$$\begin{array}{lcl} K & = & \sum_{d:\Delta}\sum_{k:\mathit{Nat}} r_{\mathrm{B}}(d,k)\cdot(j\cdot s_{\mathrm{C}}(d,k)+j)\cdot K \\ L & = & \sum_{k:\mathit{Nat}} r_{\mathrm{E}}(k)\cdot(j\cdot s_{\mathrm{F}}(k)+j)\cdot L \end{array}$$

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.

Data elements that are read from channel A by the Sender should be sent into channel D by the Receiver in the same order, and no data elements should be lost. In other words, the SWP is intended to be a solution for the process declaration

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

See Exercise 2.5 for specifications of the data type *List* and for the functions *append*, *length*, *top*, *tail* and *nonempty*. Note that the action $r_{\mathrm{A}}(d)$ can be performed until the list q 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 the special case that the window size n is restricted to one, the external behaviour above can be captured by the following more elegant process declaration (cf. the external behaviour of the ABP):

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

It can be shown, using the μCRL toolset, that this property holds for specific data sets Δ, and for small buffer sizes $2n$. However, it is an open question whether the SWP as specified above displays the desired external behaviour for general data sets and buffer sizes.

Exercise 6.4 *Define a function plusmod* : $Nat \times Nat \to Nat$*, where for* $k, \ell \in \{0, \ldots, 2n-1\}$ *plusmod*(k, ℓ) *produces the equivalence class of* $k + \ell$ *modulo* $2n$ *in* $\{0, \ldots, 2n-1\}$.

Also define a function ordered : $Nat \times Nat \times Nat \to Bool$*, where 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 range from k *to* ℓ *does not exceed* m*".*

Exercise 6.5 *Give a* μ*CRL specification of the SWP and its data types. Use the* μ*CRL toolset to analyse this specification, where* $\Delta = \{d_1, d_2\}$ *and the buffer size is four.*

Exercise 6.6 *Suppose the buffer size is three and the 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.*

In the *two-way* SWP [20, 27], 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; see Figure 6.6. 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.

Exercise 6.7 *Give a* μ*CRL specification of the two-way SWP. Use renaming to extract the Receiver and L from the process declarations of the Sender and K, respectively.*

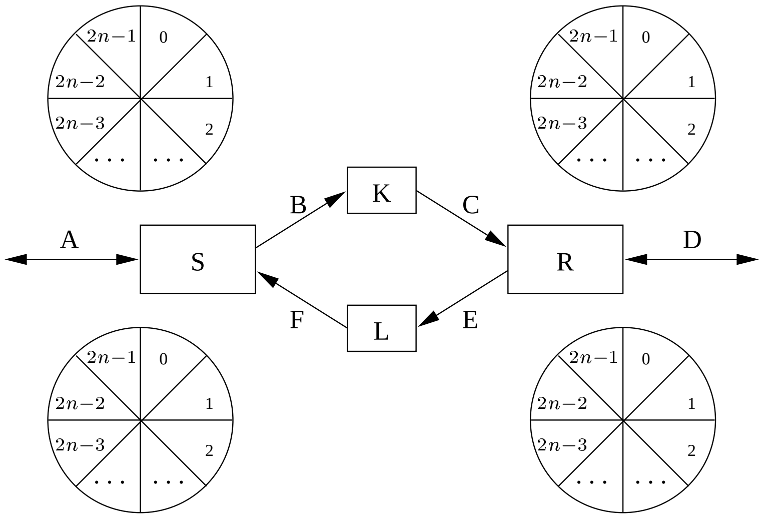


Figure 6.6: Two-way SWP

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 packets. This technique, which is commonly known as *piggybacking*, promotes a better use of available channel bandwidth.

Exercise 6.8 *Give a μCRL specification of the Sender in the two-way SWP with piggybacking.*

Piggybacking as described in Exercise 6.8 slows down the two-way SWP, since an acknowledgement may have to wait for a long time before it can be attached to a data packet. 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 [100] for more details.

Exercise 6.9 *Give a μCRL specification of the two-way SWP with piggybacking supplied with timers.*

Bezem and Groote [15] gave a μCRL specification of the *one-bit* two-way SWP with piggybacking and timers, where the window size is limited to one, and proved that their specification exhibits the desired external behaviour.

6.4 Tree identify protocol

IEEE 1394 (FireWire) connects a collection of systems and 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", so a designer or user can add or remove systems and peripherals easily at any time. The only requirement is that the form of the network should be a tree (other configurations lead to errors).

The protocol is subdivided into layers, in the manner of OSI, and further into phases, corresponding to particular tasks, e.g. data transmission or bus master identification. Much effort has been expended on the description and verification of various parts of the standard, using several different formalisms and proof techniques. For example, the operation of sending packets of information across the network is described using μCRL in [81] and using E-LOTOS in [98]. The former is essentially a description only, with five correctness properties stated informally, but not formalised or proved. The exercise of [98] is based on the μCRL description, adding another layer of the protocol and carrying out the verification suggested, using the tool CADP [44].

In this section we concentrate on the tree identify phase of the physical layer, which occurs after a bus reset in the system, e.g. 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, the protocol consists of a set of negotiations between nodes to establish the direction of the parent-child relationship. Thus, from a general graph a spanning tree is created (where possible). Potentially, a node can be a parent to many nodes, but a child of at most one node. A node with no parent (after the negotiations are complete) is the root. The TIP must ensure that a root is chosen, and that it is the only root chosen.

We present two specifications of the TIP in μCRL; one with synchronous and one with asynchronous communication. In the case of asynchronous communication matters can become more complicated, as two nodes can simultaneously send a parent request 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. The two μCRL specifications of the TIP in this section, which originate from [97], do not take into account timing aspects and probabilities. They were derived with reference to the transition diagram in Section 4.4.2.2 of the standard [71].

If the network is connected, and there are no cycles, then specifications of both the synchronous and the asynchronous version of the TIP ultimately produce one root. A verification of the synchronous version of the protocol is presented in Section 10.1.

For a formal specification of the TIP using I/O automata, and for an analysis of timing aspects and probabilities, see [40, 94, 99].

Implementation A: Synchronous communication We assume a network, consisting of a collection of nodes and connections between nodes. The aim of the TIP is to establish parent-child relations between connected nodes. The final structure should be a tree, meaning that each node has at most one parent, and exactly one node has no parent at all; this last node is called 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 $c(i,j)$. In the first implementation of the TIP, which is presented here, 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. So, as channels are supposed to be secure, 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 of some 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 root of the network.

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 name $Node(i,p,s)$ represents node i in state s, with as list of possible parents p.

$$\begin{array}{rcl} Node(i,p,0) & = & \sum_{j:Nlist} r(j,i)\cdot Node(i,p\backslash\{j\},0) \lhd j\in p \rhd \delta \\ & & +\ \sum_{j:Nlist} s(i,j)\cdot Node(i,p,1) \lhd p=\{j\} \rhd \delta \\ Node(i,[],0) & = & leader(i)\cdot Node(i,[],1) \end{array}$$

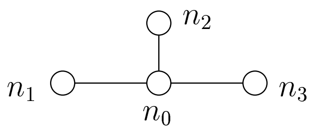
The initial state of Implementation A consists of the parallel composition of the node processes for the nodes $n_1,\ldots,n_k$ in state 0, with as possible parents the lists of all neighbours $p_1,\ldots,p_k$:

$$\tau_I(\partial_H(Node(n_1,p_1,0) \parallel \cdots \parallel Node(n_k,p_k,0)))$$

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

In Section 10.1 it is proved formally that if a network is connected and free of cycles, then the specification above produces one root.

Example 6.10 *The network*



is captured by

$$\tau_I(\partial_H(X(n_0,\{n_1,n_2,n_3\},0) \parallel X(n_1,\{n_0\},0) \parallel X(n_2,\{n_0\},0) \parallel X(n_3,\{n_0\},0)))$$

where H consists of all read and send actions between n_0 and the other three nodes, while I consists of all communication actions between n_0 and the other three nodes. The external behaviour is:

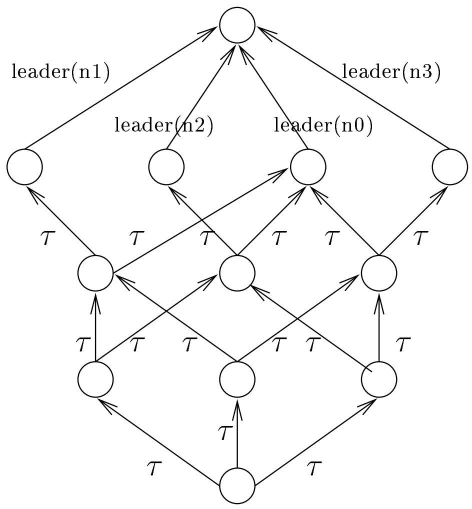
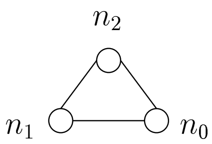


Figure 6.7: External behaviour of the TIP

Exercise 6.11 *Explain why the τ-transitions in the external behaviour depicted in Figure 6.7 are not intert.*

Exercise 6.12 *Consider the network*



Give the μCRL specification of the initial state of this network. Explain why in this case no root will be elected.

Implementation B: Asynchronous communication Implementation A assumes synchronous communication between nodes. In reality, messages are sent by variations in voltage along wires of various lengths. Hence, these messages are not received instantaneously, so communication is asynchronous. This means that a node may ask to be a child of its neighbour, while that neighbour already sent a message asking to be *its* child (but the messages have crossed in transmission). That situation, called *root contention*, needs to be resolved.

In Implementation B, unidirectional one-place buffers are introduced to model asynchronous communication between nodes, there are two buffers of each pair of neighbours. As 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 read, send and communication messages from nodes to buffers, denoted by r', s' and c', and from buffers to nodes, denoted by r, s and c.

Again, individual nodes of the network are specified as separate processes, which are put in parallel with the one-place buffers. 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 Figure 6.8. 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 are collapsed. If 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 it either 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 ultimately 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.

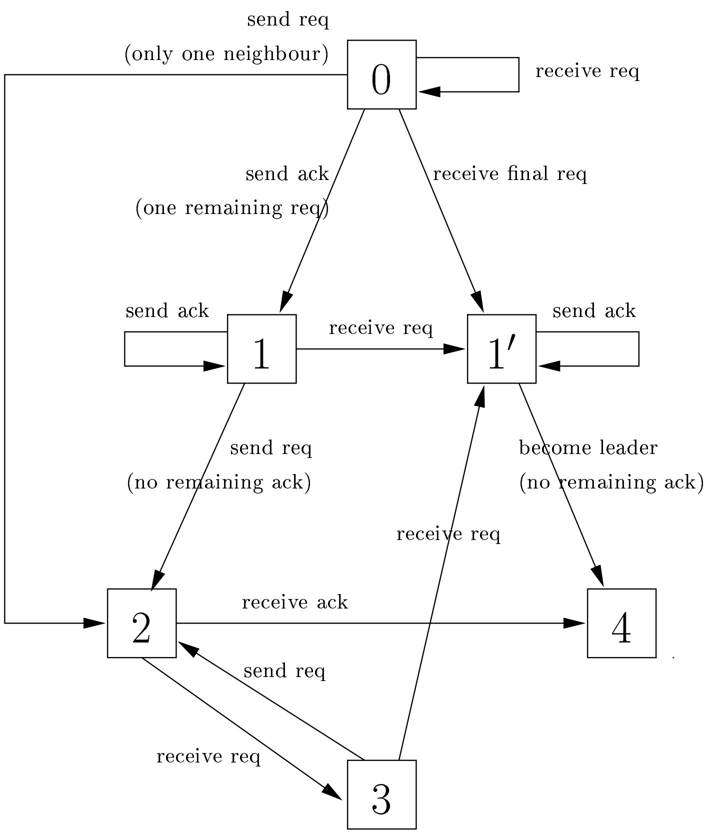


Figure 6.8: Relations between the five states

In the μCRL specification of Implementation B, the node i in state s is represented by the process name $Node(i, p, q, s)$, with p as list of possible parents, and with q as list of neighbours to which it must still send an acknowledgement. Lists of nodes are built from the standard constructors [] and in. Moreover, the specification includes the following mappings:

- $rem(j, p)$ 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;

- $occur(j, p)$ tests whether node j occurs in list p;

- $empty(p)$ tests whether list p is empty.

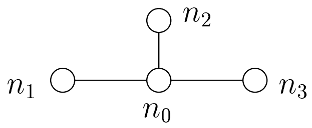
$$
\begin{array}{rcl}
Node(i,p,q,0) & = & \sum_{j:N} r(j,i,req)\cdot Node(i,rem(j,p),in(j,q),if(single(p),1,0)) \lhd occur(j,p) \rhd \delta \\
& + & \sum_{j:N} s'(i,j,ack)\cdot Node(i,p,rem(j,q),1) \lhd single(p) \wedge occur(j,q) \rhd \delta \\
& + & \sum_{j:N} s'(i,j,req)\cdot Node(i,p,q,2) \lhd single(p,j) \wedge empty(q) \rhd \delta \\
\\
Node(i,p,q,1) & = & \sum_{j:N} s'(i,j,ack)\cdot Node(i,p,rem(j,q),1) \lhd occur(j,q) \rhd \delta \\
& + & \sum_{j:N} s'(i,j,req)\cdot Node(i,p,q,2) \lhd single(p,j) \wedge empty(q) \rhd \delta \\
& + & leader(i)\cdot Node(i,p,q,4) \lhd empty(p) \wedge empty(q) \rhd \delta \\
& + & \sum_{j:N} r(j,i,req)\cdot Node(i,[],in(j,q),1) \lhd single(p,j) \rhd \delta \\
\\
Node(i,p,q,2) & = & \sum_{j:N} r(j,i,ack)\cdot Node(i,p,q,4) \lhd single(p,j) \rhd \delta \\
& + & \sum_{j:N} r(j,i,req)\cdot Node(i,p,q,3) \lhd single(p,j) \rhd \delta \\
\\
Node(i,p,q,3) & = & \sum_{j:N} r(j,i,req)\cdot Node(i,[],p,1) \lhd single(p,j) \rhd \delta \\
& + & \sum_{j:N} s'(i,j,req)\cdot Node(i,p,q,2) \lhd single(p,j) \rhd \delta \\
\\
B(i,j) & = & r'(i,j,req)\cdot s(i,j,req)\cdot B(i,j) + r'(i,j,ack)\cdot s(i,j,ack)\cdot B(i,j)
\end{array}
$$

The initial state of Implementation B consists of the parallel composition of the one-place buffers and the node processes for the nodes $n_1,\ldots,n_k$ in state 0, with as possible parents the lists of all neighbours $p_1,\ldots,p_k$ and with as neighbours that need to be acknowledged the empty list:

$$\tau_I(\partial_H(Node(n_1,p_1,[],0) \parallel \cdots \parallel Node(n_k,p_k,[],0) \parallel B(n_\ell,n_{\ell'}) \parallel \cdots \parallel B(n_m,n_{m'})))$$

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

Example 6.13 *The network*



is captured by

$$\tau_I(\partial_H(X(n_0,\{n_1,n_2,n_3\},[],0) \parallel X(n_1,\{n_0\},[],0) \parallel X(n_2,\{n_0\},[],0)$$
$$\parallel X(n_3,\{n_0\},[],0) \parallel B(n_0,n_1)||B(n_0,n_2)||B(n_0,n_3)||B(n_1,n_0)||B(n_2,n_0)||B(n_3,n_0)))$$

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 is depicted in Figure 6.7.

Exercise 6.14 *Five philosophers sit around a plate of noodles. Between each philosopher 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.*

Chapter 7

Graph Algorithms

In this chapter we present some of the ideas and algorithms that underlie the transformation and analysis of state spaces that are generated from μCRL specifications. These algorithms are supported by the μCRL and CADP tool sets.

7.1 Minimisation modulo branching bisimulation

We sketch an algorithm by Groote and Vaandrager [63] to decide whether two finite-state processes are branching bisimilar (see Definition 5.5). The basic idea of this algorithm is to partition the set of states of the input graph into subsets of states that may be branching bisimilar; if two states are in distinct sets, 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 state space containing finitely many states. In order to explain the minimisation algorithm, we need to introduce some preliminary terminology. 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 a sequence of transitions $s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_{n-1} \xrightarrow{a} s_n$ for some $n > 0$ such that $s_i \in P$ for $i = 0, \ldots, n-1$ and $s_n \in P'$.

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 $s_0 \in \mathit{split}_a(P, P')$ and a state $s_1 \in P \backslash \mathit{split}_a(P, P')$ cannot be branching bisimilar if $a \neq \tau$ or $P \neq P'$. Namely, $s_0 \in \mathit{split}_a(P, P')$ implies that there exists an execution sequence $s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s \xrightarrow{a} s'$ with $s \in P$ and $s' \in P'$, and since $s_1 \notin \mathit{split}_a(P, P')$ this execution sequence cannot be mimicked by s_1.

The minimisation algorithm works as follows. Suppose that at some point we have constructed a partition $P_1, \ldots, P_k$ of disjoint non-empty sets of states, where $P_1 \cup \cdots \cup P_k$ is the set of states in the input graph. (Remember that initially this partition consists of a single set.) If 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 that $\emptyset \subset \mathit{split}_b(P_i, P_j) \subset P_i$ (where $\subset$

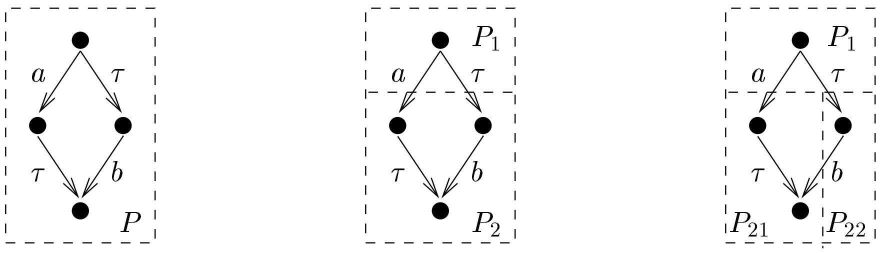
denotes strict set inclusion), then P_i can be replaced by the two disjoint non-empty sets $\mathit{split}_b(P_i, P_j)$ and $P_i \backslash \mathit{split}_b(P_i, P_j)$. This procedure is repeated until no set in the partition can be split in this fashion any further.

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 input graph. Thus, the states in a set Q_i can be collapsed, producing the desired minimisation of the input graph modulo branching bisimulation equivalence.

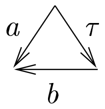
Example 7.1 *We show how the minimisation algorithm minimises the process term* $(a\cdot\tau + \tau\cdot b)\cdot\delta$*. Initially, the set* P *contains all four states in the corresponding state space.*

$\mathit{split}_a(P, P)$ *consists of the root node only, so that the minimisation algorithm separates the root node from the other nodes.*

Next, $\mathit{split}_b(P_2, P_2)$ *only contains the node belonging to the subterm* $b\cdot\delta$*, so that the minimisation algorithm separates this node from the other two nodes 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:*



Exercise 7.2 *Apply the minimisation algorithm 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 7.3 *Explain how the minimisation algorithm can be adapted to take into account the successful termination predicate* $\xrightarrow{\tau} \surd$.

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 input graph. 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, also requires $O(m)$. Since there can be no more than $n - 1$ subsequent splits of sets in the partition, the worst-case time complexity is $O(mn)$.

Paige and Tarjan [88] presented an efficient algorithm to minimise a finite state space modulo bisimulation equivalence (see Definition 4.1), which has worst-case time complexity $O(m \log n)$. In the case of minimisation of a large finite-state process modulo branching bisimulation equivalence, it can be wise to first minimise the process modulo bisimulation equivalence, as this preprocessing step can be performed more efficiently.

Weak bisimulation [84] is a popular alternative to branching bisimulation, if one wants to abstract away from internal behaviour. Weak bisimulation is coarser than branching bisimulation, in the sense that two branching bisimilar processes are by default also weakly bisimilar. Kanellakis and Smolka [72] presented a minimisation algorithm for weak bisimulation equivalence. Basically it consists of saturating the input graph with τ-transitions, after which the minimisation algorithm of bisimulation equivalence can be applied, interpreting the τ as a concrete visible action. This algorithm is less efficient than the minimisation algorithm modulo branching bisimulation equivalence.

7.2 Confluence

In real-life systems, often two different concurrent components can perform transitions that are independent, meaning that executing the two actions consecutively leads to the same state, irrespective of the fact which of the two transitions is executed first. This phenomenon is generally referred to as *confluence* (cf. Section 2.2). Milner [84, 86] was the first to recognise the usefulness of this notion of confluence 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 [17, 56, 61, 106]. In [61] several notions of confluence were studied on their practical applicability.

Here we use the following 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 $s \xrightarrow{\tau}_T s''$ in the state space, the picture

$$\begin{tikzcd} & s \arrow[dl, "a"'] \arrow[dr, "\tau", "T"'] & \\ s' & & s'' \end{tikzcd}$$

can be completed in the following fashion:

$$\begin{tikzcd} & s \arrow[dl, "a"'] \arrow[dr, "\tau", "T"'] & \\ s' \arrow[dr, "\tau"', "T"] & & s'' \arrow[dl, "a"] \\ & s''' & \end{tikzcd}$$

Definition 7.4 (Confluence) *Assume a state space L. A collection T of τ-transitions in L is* confluent *if for all transitions $s \xrightarrow{a} s' \in L$ and $s \xrightarrow{\tau} s'' \in T$:*

(1) either $s' \xrightarrow{\tau} s''' \in T$ and $s'' \xrightarrow{a} s''' \in L$, for some state s''';

(2) or $a = \tau$ and $s' = s''$.

The union of confluent collections of τ-transitions is again confluent, so there is a maximal confluent collection of τ-transitions.

Groote and van de Pol [56] presented an efficient algorithm to compute, for a given finite state space, the maximal set T of τ-transitions that is confluent with respect to this state space. As a preprocessing step, first all states that are on a τ-loop are collapsed to a single state. That is, if there are sequences of τ-transitions $s \xrightarrow{\tau} \cdots \xrightarrow{\tau} s'$ 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 5.9. The τ-loops in a state space can be detected using a well-known linear-time algorithm by Tarjan [101] for finding the *strongly connected components* in a graph. Two states s and s' are in the same strongly connected component of a state space if there exist execution sequences from s to s' and vice versa.

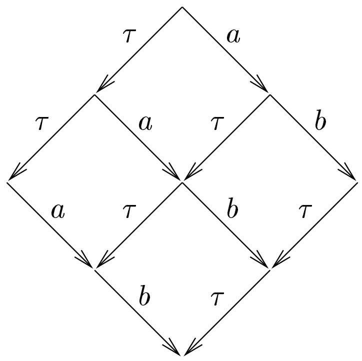
In the algorithm from [56], initially it is assumed that all τ-transitions of the state space are in T, and subsequently transitions are eliminated from T if they are found to obstruct confluence. That is, all transitions of the input graph are placed on a stack. In each processing step, a transition $s \xrightarrow{a} s'$ 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 7.4 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.

A confluent set T of τ-transitions can be used to trim the corresponding state space L. Namely, if $s \xrightarrow{\tau} s' \in T$, then s and s' are branching bisimilar states in L, so that they can be identified. Even more so, if L 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 L) 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 L. The next example shows that the absence of τ-loops in L is essential for the soundness of priorisation of confluent τ-transitions.

Example 7.5 *Consider the state space defined by the process declaration $X = (\tau + a)\cdot X$; note that it contains a τ-loop. 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 process declaration $Y = \tau\cdot Y$. Clearly the state belonging to X and the state belonging to Y are not branching bisimilar.*

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

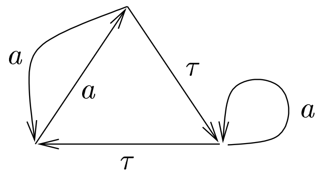
Example 7.6 *Consider the state space*



The maximal confluent set of τ-transitions contains all six τ-transitions in this state space. 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 in general the maximal confluent set of τ-transitions may be a proper subset of the collection of inert τ-transitions of a state space.

Example 7.7 *Consider the state space*

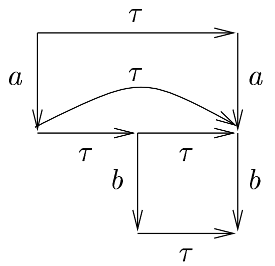


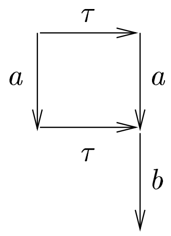
This state space is minimal modulo bisimulation equivalence, and its maximal confluent set of τ-transitions is empty. However, the two τ-transitions are both inert.

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 $O(m)$. 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 $O(m)$ (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 bisimulation equivalence; see Section 7.1. 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 input graph 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 bisimulation equivalence. Computation of the maximal confluent set of τ-transitions is supported by the μCRL toolset.

Finally, we note that 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.

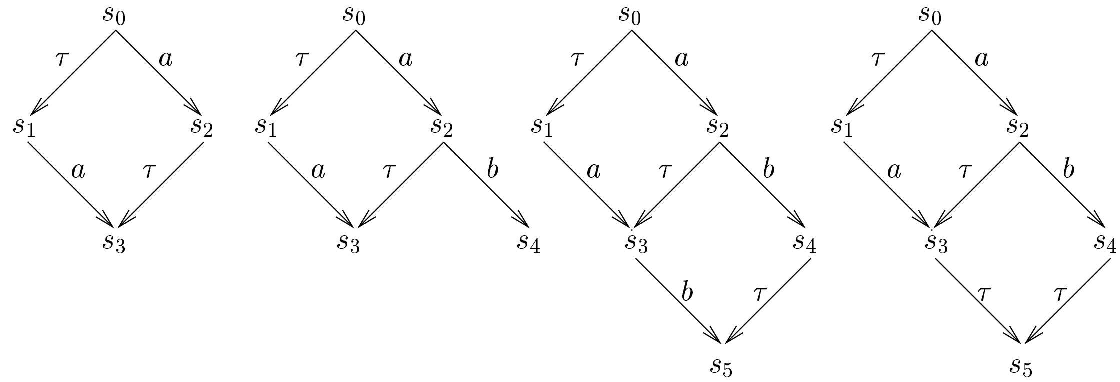
Example 7.8 *Below are presented a state space before and after compression with respect to the confluent τ-transitions, respectively:*



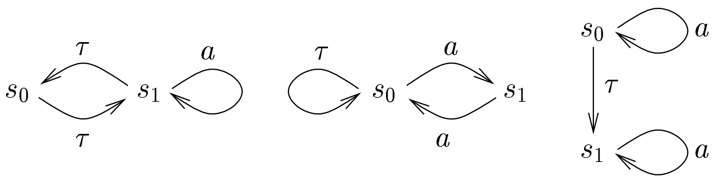


Compression with respect to the confluent τ-transitions in the latter state space produces the state space belonging to $a{\cdot}b{\cdot}\delta$.

Exercise 7.9 *Give the maximal confluent sets of τ-transitions of the following four state spaces:*



Exercise 7.10 *Give the maximal confluent sets of τ-transitions of the following three state spaces:*



7.3 Model checking

Model checking is a method for verifying properties of finite state spaces. 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 three temporal logics: computation tree logic, the μ-calculus, and a value-based extension of the μ-calculus.

Computation tree logic CTL [29] is a temporal logic to express properties of state spaces that do not carry actions on their transitions. ACTL [39] is an extension of CTL to state spaces with actions. ACTL consists of formulas on states, defined by the following BNF grammar:

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

where a ranges over Act. Here, $\mathtt{T}$ is the universal predicate that holds for all states.[1] 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 intuitions behind the three temporal constructs are as follows. (The '$\mathtt{E}$' stands for existential quantification.)

[1] Actually, CTL assumes a collection of predicates, which hold in part of the states.

- $\langle a \rangle \phi$ 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 an 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:

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

CTL contains some more temporal operators, which can all be expresses in terms of the aforementioned constructs. (In the operators defined below, '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'. That is,

$$\neg\,\langle a \rangle\,\neg\phi.$$

- $\mathtt{A}\,\phi\,\mathtt{U}\,\phi'$ 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. That is,

$$(\neg\,\mathtt{E}\,\neg\phi'\,\mathtt{U}\,(\neg\phi \wedge \neg\phi'))\ \wedge\ (\neg\,\mathtt{EG}\,\neg\phi').$$

- $\mathtt{AG}_a\,\phi$ holds in a state s if each execution sequence, starting in s only visits states in which ϕ holds. That is,

$$\neg\,\mathtt{E}\,\mathtt{T}\,\mathtt{U}\,\neg\phi$$

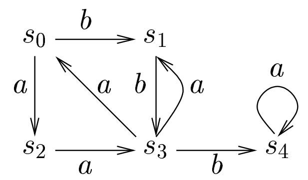
Assume a finite state space. The verification in which states a given temporal logic formula holds is known as *model checking*. Clarke, Emerson, and Sistla

[30] presented an efficient model checking algorithm for CTL formulas, which extends to ACTL without any complications [39]. 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 input graph it is true.

- The three basic boolean operators are straightforward: $\mathtt{T}$ holds for 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 converse of the transition relation, to find all states that can be reached by an execution sequence in which each state satisfies ϕ.
- To compute the states in which $\mathtt{EG}\,\phi$ holds, first we eliminate all states from the input graph where ϕ does not hold. A state satisfies $\mathtt{EG}\,\phi$ if and only if there is an execution sequence in the resulting state space to a state that has no outgoing transitions in the original graph, or to a strongly connected component that includes at least one transition.

This algorithm has worst-case time complexity $O(km)$, where k is the size of the given ACTL formula and m is the number of transitions of the input graph. A crucial factor for this nice complexity is that strongly connected components can be computed in time $O(m)$, using Tarjan's algorithm [101] (cf. Section 7.1).

Exercise 7.11 *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) $\mathtt{EG}\, \langle a \rangle \mathtt{T}$*;*

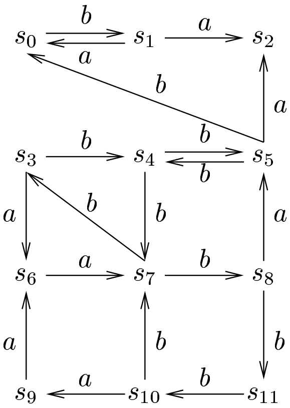
(4) $\mathtt{EG}\, \langle b \rangle \mathtt{T}$*;*

(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 7.12 *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) $\neg\,\mathtt{EG}\,\neg\,\langle a\rangle\,\langle b\rangle\,\mathtt{T}$;

(3) $\langle a\rangle\,\langle a\rangle\,\neg\,(\langle a\rangle\,\mathtt{T} \vee \langle b\rangle\,\mathtt{T})$;

(4) $\langle b\rangle\,\langle b\rangle\,\langle b\rangle\,\langle a\rangle\,\langle b\rangle\,\langle a\rangle\,\neg\,(\langle a\rangle\,\mathtt{T} \vee \langle b\rangle\,\mathtt{T})$;

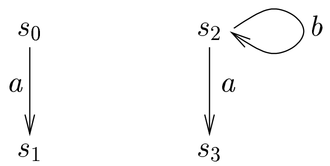
(5) $\mathtt{E}\,(\neg\,\langle a\rangle\,\langle b\rangle\,\mathtt{T})\,\mathtt{U}\,((\langle a\rangle\,\langle a\rangle\,\neg\,(\langle a\rangle\,\mathtt{T} \vee \langle b\rangle\,\mathtt{T})) \vee (\langle b\rangle\,\langle b\rangle\,\langle b\rangle\,\langle a\rangle\,\langle b\rangle\,\langle a\rangle\,\neg\,(\langle a\rangle\,\mathtt{T} \vee \langle b\rangle\,\mathtt{T})))$.

Exercise 7.13 *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.

Fairness Branching bisimulation equivalence satisfies a notion of *fairness*; see Section 5.4. This means that 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 process graphs.



Due to the b-loop, state s_2 satisfies $\texttt{EG}\,\langle a\rangle\texttt{T}$. 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 $\langle a\rangle\texttt{T}$ 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 $\texttt{E}\,\phi\,\texttt{U}\,\phi'$ remains unchanged, because these formulas deal with the existence of some finite execution sequence. However, the semantics of $\texttt{EG}\,\phi$ 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 any exits. The interpretation of $\texttt{EG}\,\phi$ 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 input graph 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 graph.*

Linear temporal logic LTL [91] 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 [79] 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 [42] for a comparison of branching-time and linear-time temporal logics.

μ-calculus The μ-calculus [76] is based on fixpoint computations [102]. Let D be a set with a partial ordering $\leq$, and consider a mapping $\mathcal{S} : D \to D$. An element d of 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 $\mu X.S(X)$ and $\nu X.S(X)$, respectively. The mapping $\mathcal{S} : D \to D$ is called *monotonic* if $d \leq e$ implies $\mathcal{S}(d) \leq \mathcal{S}(e)$. Each monotonic mapping $\mathcal{S}$ has a least and a greatest fixpoint.

Let $\mathcal{S} : D \to D$ be monotonic. Furthermore, let D be finite, and suppose 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 the least and greatest fixpoints of $\mathcal{S}$ can be computed in an iterative way. Put $d_{i+1} = \mathcal{S}(d_i)$ and $e_{i+1} = \mathcal{S}(e_i)$ for $i \geq 0$. By the monotonicity of $\mathcal{S}$ it follows that $d_i \leq d_{i+1}$ and $e_{i+1} \leq e_i$ for all $i \geq 0$. Since 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 $\mu X.S(X) = d_j$ and $\nu X.S(X) = e_k$, respectively.

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

$$\phi \quad ::= \quad \mathtt{T} \mid \neg\phi \mid \phi \wedge \phi' \mid \langle a \rangle \phi \mid X \mid \mu X.\phi \mid \nu X.\phi$$

where a ranges over Act 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.

As before, $\langle a \rangle \phi$ holds in those states where an a-transition is possible to a state where ϕ holds. The formulas $\mu X.\phi$ and $\nu X.\phi$ represent minimal and maximal fixpoints, respectively. Here, 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 is true in all states in S and false in all other states. 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 and $\langle a \rangle \phi$ are indeed monotonic, since $\phi \Rightarrow \phi'$ yields $\phi \wedge \phi'' \Rightarrow \phi' \wedge \phi''$, $\phi'' \wedge \phi \Rightarrow \phi'' \wedge \phi'$ and $\langle a \rangle \phi \Rightarrow \langle a \rangle \phi'$. However, negation is not monotonic. We give an example.

Example 7.14 *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 terminate at a fixpoint.

Since negation is not monotonic, a subclass of the μ-calculus needs to be determined where monotonicity is guaranteed. We restrict to closed μ-calculus formulas where each occurrence of a recursion variable is guarded by an *even* number of negation symbols. For example, the formulas $\mu X.\langle a \rangle X$ and $\mu X.\neg\langle a \rangle \neg X$ are within this class, while the formula $\mu X.\neg\langle a\rangle X$ (from Example 7.14) is not; in the first two formulas X is guarded by an even number of negation symbols (zero and two, respectively), while in the third formula X is guarded by an odd number of negation symbols (namely, one).

If all occurrences of recursion variables in a closed μ-calculus formula ϕ are guarded by an even number of negation symbols, then the mapping from sets of states to sets of states represented by ϕ is monotonic. Hence, on this subset of the μ-calculus, minimal and maximal fixpoints are well-defined.

$\mu X.\phi$ and $\nu X.\phi$ can be computed as explained before. For the case of $\mu X.\phi$, initially we take as a solution S_0 for X the empty set of states. Next, we repeatedly compute $S_{i+1} = \mathit{FIX}(\phi, X{=}S_i)$ 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 of $\nu X.\phi$, initially we take as a solution S_0 for X the set of all states. Next, we repeatedly compute $S_{i+1} = \mathit{FIX}(\phi, X{=}S_i)$ 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 7.15 *Consider the state space*

$$\begin{tikzcd} s_0 \arrow[r, shift left, "a"] & s_1 \arrow[l, shift left, "b"] \arrow[r, "a"] & s_2 \arrow[r, "b"] & s_3 \end{tikzcd}$$

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.

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.

Consider a formula $\mu X.\phi$. By monotonicity, computing $\mathit{FIX}(\phi, X{=}\emptyset)$ 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 $\nu Y.\phi'$, 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 $O(n^{N(\phi)})$, where $N(\phi)$ is the length of the longest chain of nexted fixpoints in ϕ:

$$\begin{array}{lcllcl} N(\mathtt{T}) & = & 0 & N(X) & = & 0 \\ N(\neg\phi) & = & N(\phi) & N(\mu X.\phi) & = & N(\phi)+1 \\ N(\phi\wedge\phi') & = & \max\{N(\phi),N(\phi')\} & N(\nu X.\phi) & = & N(\phi)+1 \\ N(\langle a\rangle\phi) & = & N(\phi) & & & \end{array}$$

Example 7.16 *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
$\{s_0,\ldots,s_{2n-2}\}$	$\{s_0,\ldots,s_{2n-3}\}$
$\{s_0,\ldots,s_{2n-4}\}$	$\{s_0,\ldots,s_{2n-5}\}$
$\vdots$	$\vdots$
$\emptyset$	$\emptyset$

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

Emerson and Lei [41] observed that 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. 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 newly computed solution for Y is guaranteed to be included in the previous solution for Y (cf. Example 7.16). Hence, the worst-case time-complexity for model checking the μ-calculus reduces to $O(n^{A(\phi)})$, 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 a newly computed solution for Y is not guaranteed to include the previous solution for Y.

Example 7.17 *Consider the state space from Example 7.16. We compute the solutions for* X *and* Y *in the formula* $\nu X.\mu Y.(\langle a\rangle X\vee\langle b\rangle Y)$*. 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
$\{s_0,\ldots,s_{2n-2}\}$	$\{s_0,\ldots,s_{2n-3}\}$
$\{s_0,\ldots,s_{2n-4}\}$	$\{s_0,\ldots,s_{2n-5}\}$
$\vdots$	$\vdots$
$\emptyset$	$\emptyset$

The empty set is the ultimate solution for both X and Y. Note that the successive intermediate solutions for the minimal fixpoint Y are decreasing sets.

It is not a coincidence that in Examples 7.16 and 7.17, X and Y have the same solution. One can prove that if two fixpoints are direct neighbours (e.g., $\nu X.\mu Y.\phi$), then their solutions coincide. The following example features nested fixpoints that are not direct neighbours.

Example 7.18 *Consider the state space*

$$\begin{tikzcd} s_0 \arrow[r, "a"] & s_1 \arrow[r, "b"] & s_2 \arrow[r, "a"] & s_3 \arrow[ll, bend left, "c"] \end{tikzcd}$$

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

Y	X
$\emptyset$	$\{s_0, s_1, s_2, s_3\}$
$\{s_0, s_1, s_2\}$	$\{s_3\}$
$\{s_1, s_2\}$	$\{s_3\}$

Exercise 7.19 *Consider the state space*

$$\begin{tikzcd} s_0 \arrow[r, "a"] & s_1 \arrow[r, "b"] & s_2 \arrow[r, "a"] & s_3 \arrow[lll, bend left, "c"] \end{tikzcd}$$

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

Distinguishing logics HML CTL (TO BE ADDED)

μ-calculus with data (TO BE ADDED)

7.4 Bitstate hashing

Hashing (see, e.g., [33]) 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 the indices of the table to a much smaller 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 indices i and j of two occupied positions in the table have the same hash value, meaning that $h(i) = h(j)$, then the two items at positions i and j in the table are kept as a linked list at position $h(i)$ in the resulting hash table.

Holzmann [70] introduced hashing as a method for storing state spaces. The idea is that usually the amount of available memory is known, while the size of the state space to be generated is unknown, and often in the end turns out to be much too large to be stored in memory. By defining a (random) hash function from states to the index set of memory positions, it is guaranteed that the number of stored states does not exceed the number of available positions. A crucial new idea is that, unlike standard hash tables, no linked lists are maintained; that is, if two generated states happen to have the same hash value, then they are basically collapsed in the generated state space. Thus, in the hash table, a single bit per stored state suffices. (However, during state space generation, states that were detected but not yet explored usually have to be kept in memory, which can be expensive.)

A disadvantage of this approach is that states that are not branching bisimilar may be collapsed, if their hash values happen to coincide. Due to this fact, propserties derived for the generated state space may no longer be robust. For example, present deadlocks may not be detected, if these deadlock states are collapsed with states that are not deadlocks.

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 collapsed. 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 underlies the model checker SPIN [69], and is available for the μCRL toolset via a connection to OPEN/CÆSAR [46].

Chapter 8

Linear Process Equations

A *linear process equation* (LPE) [14] is a one-line process declaration that consists of atomic actions, summations, sequential compositions and conditionals. In particular, an LPE does not contain any parallel operators, encapsulations or hidings. In essence an LPE is a vector of data parameters together with a list of summands consisting of a condition, action and effect triple, describing when an action may happen and what is its effect on the vector of data parameters.

Each μCRL specification can be transformed into an LPE (see Section 8.2). 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 parallel components typically gives rise to an LPE of only several hundreds of summands, while the corresponding state space may consist of billions of states. Even in cases where the state space is too large to allow generation, the LPE can generally be obtained without much effort. We will explain proof techniques for LPEs, algorithms to prove properties of a state space on the level of an LPE, and transformations of LPEs to arrive at a simplified LPE or at a smaller state space.

Definition 8.1 (Linear process equation) *A* linear process equation *is a process declaration of the form*

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

where for each $a \in \mathsf{Act} \cup \{\tau\}$ we have $f_a : D \times E \to D_a$, $g_a : D \times E \to D$ and $h_a : D \times E \to Bool$.

Intuitively, in an LPE written as in Definition 8.1, the different states of the process are represented by the data parameter $d{:}D$. Note that D may be a Carthesian product of n data types, meaning that d may actually be a tuple $(d_1, \ldots, d_n)$. The LPE expresses that in state d one can perform actions a, carrying a data parameter $f_a(d,e)$, under the condition that $h_a(d,e)$ is true; in this case the resulting state is $g_a(d,e)$.

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 action a, the condition h_a and the resulting state g_a. Typically, a data parameter e in an LPE is used to let a read or send action range over a data domain. See for example the process declaration of a queue of capacity two in Section 5.3.

8.1 CL-RSP

In this section we introduce the proof principle CL-RSP, which states that the solutions of a convergent LPE (i.e., an LPE without an infinite sequence of τ-transitions) are rooted branching bisimilar. Convergence is essential for the soundness of CL-RSP.

Definition 8.2 (Convergent LPE) *An LPE is* convergent *if it cannot exhibit an infinite sequence of τ-transitions from any state.*

In general it is undecidable whether or not an LPE is convergent. The following result provides a syntactic criterion that is often useful in showing that an LPE is convergent.

Lemma 8.3 *An LPE written as in Definition 8.1 is convergent if there exists a well-founded ordering $<$ on D such that for all $e{:}E$ and $d{:}D$, $h_\tau(d,e)$ implies $g_\tau(d,e) < d$.*

Proof. The well-founded ordering guarantees that for each τ-transition, the original state d is strictly greater than the resulting state $g_\tau(d,e)$, with respect to the ordering $<$. Since this ordering is well-founded, there is no infinite decreasing sequence $d_1 > d_2 > d_3 > \cdots$ of data elements of type D. ⊠

The derivation rule CL-RSP (*Convergent Linear Recursive Specification Principle*) of Bezem and Groote [14] says that every convergent LPE has at most one solution. Hennessy and Lin [66] introduced a similar derivation rule called UFI-O.

Definition 8.4 (CL-RSP) *Let the LPE*

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

be convergent. If $t(d)$ denotes a process term for each $d{:}D$, such that

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

then $t(d) = X(d)$ for all $d{:}D$.

Convergence 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 5.1 and 3.4, respectively) we have, 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 branching bisimilar if $a \neq b$.

From now on, for notational convenience, expressions $X(\_) \lhd \mathtt{T} \rhd \delta$ in LPEs are abbreviated to $X(\_)$.

Example 8.5 *Consider the following two LPEs:*

$$X(m{:}Nat) \quad = \quad a(2m){\cdot}X(m+1)$$

$$Y(n{:}Nat) \quad = \quad a(n){\cdot}Y(n+2)$$

Substituting $Y(2m)$ for $X(m)$ in the first LPE, for $m{:}Nat$, yields $Y(2m) = a(2m){\cdot}Y(2(m+1))$. This equality follows from the second LPE by substituting $2m$ for n. Since the LPE for X is convergent, by CL-RSP

$$Y(2m) \;=\; X(m)$$

for $m{:}Nat$.

8.2 Linearisation

Groote, Ponse and Usenko [60] described how each process declaration in *parallel pCRL* (the “p” stands for “pico”) can be transformed into an LPE. Parallel pCRL is a proper subset of μCRL. Basically, in parallel pCRL, atomic actions, summations, sequential compositions and conditionals are used to build basic processes, to which parallel composition, encapsulation and hiding can be applied. The linearisation algorithm from [60] underlies the `mcrl` command of the μCRL toolset, which transforms a parallel pCRL specification into an LPE; see [105].[1] Usenko [?] showed how any process declaration can be transformed into an LPE.

Groote and van Wamel [64] showed how a special class of μCRL specifications, consisting of the parallel composition of the same LPE, with different data parameters, can be quite easily transformed to a single LPE. Given a convergent LPE X over a data type D, terms

$$X(d_0) \parallel \cdots \parallel X(d_n)$$

can be equated to $Y(d_0, \ldots, d_n)$, where Y is a convergent LPE.

In this section, we give an example of the latter transformation, to sketch its basic idea. In this example, $Y(n_i := n_i + 1)$ abbreviates $Y(n_1, \ldots, n_i + 1, \ldots, n_k)$, while $Y(n_j := n_j + 2)$ and $Y(n_i := n_i + 1, n_j := n_j + 2)$ denote similar abbreviations. Similar abbreviations will be used in the remainder of this text without further mention.

[1] Currently, applying `mcrl` to a μCRL specification that lies outside parallel pCRL produces an error message.

Example 8.6 *Let $a\,|\,b = c$, and consider the LPE*

$$X(n{:}Nat) = a(n)\cdot X(n+1) \lhd n < 10 \rhd \delta + b(n)\cdot X(n+2) \lhd n > 5 \rhd \delta$$

For natural numbers k, the parallel composition $X(n_1) \parallel \cdots \parallel X(n_k)$ is equal to $Y(n_1, \ldots, n_k)$, where

$$\begin{array}{l} Y(n_1{:}Nat, \ldots, n_k{:}Nat) \;=\; \\ \sum_{i\in\{1,\ldots,k\}} a(n_i)\cdot Y(n_i := n_i + 1) \lhd n_i < 10 \rhd \delta \\ +\; \sum_{j\in\{1,\ldots,k\}} b(n_j)\cdot Y(n_j := n_j + 2) \lhd n_j > 5 \rhd \delta \\ +\; \sum_{i\in\{1,\ldots,k\}} \sum_{j\in\{1,\ldots,i-1,i+1,\ldots,k\}} c(n_i)\cdot Y(n_i := n_i + 1, n_j := n_j + 2) \\ \qquad\qquad\qquad\qquad \lhd n_i < 10 \wedge n_j > 5 \wedge n_i = n_j \rhd \delta \end{array}$$

Exercise 8.7 *Describe the following process declaration by means of an LPE:*

$$X \;=\; (a + c)\cdot d\cdot X$$

Exercise 8.8 *Let $a\,|\,b = c$, 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 for natural numbers k, the parallel composition $X(n_1) \parallel \cdots \parallel X(n_k)$ is equal to $Y(n_1, \ldots, n_k)$

Exercise 8.9 *Prove with the help of CL-RSP that the parallel composition $X(n_1) \parallel \cdots \parallel X(n_k)$ in Example 8.6 is equal to $Y(n_1, \ldots, n_k)$, for $k = 2$.*

8.3 Invariants

Definition 8.10 (Invariant) *A mapping $\mathcal{I} : D \to Bool$ is an invariant for an LPE written as in Definition 8.1 if, for all $a \in \mathsf{Act} \cup \{\tau\}$, $d{:}D$ and $e{:}E$,*

$$\mathcal{I}(d) \wedge h_a(d, e) \;\Rightarrow\; \mathcal{I}(g_a(d, e)) \tag{8.1}$$

Intuitively, an invariant characterises the set of reachable states of an LPE. That is, if $\mathcal{I}(d) = \mathtt{T}$ and if one can evolve from state d 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 $a(f_a(d, e))$ in this state (meaning that $h_a(d, e) = \mathtt{T}$), then it is ensured by (8.1) that $\mathcal{I}$ holds in the resulting state $g_a(d, e)$.

Invariants tend to play a crucial role in algebraic verifications of system behaviour that involve data. Namely, the properties that are verified generally do not need to hold (and do not hold) in all possible states, but only in the reachable ones. Since invariants are guaranteed to hold in all reachable states, they can be used in the derivation of these properties.

Example 8.11 *Consider the LPE* $X(n{:}Nat) = a(n){\cdot}X(n+2)$. *Invariants for this LPE are*

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

Exercise 8.12 *Show that the mappings* $\mathcal{I}_1$ *and* $\mathcal{I}_2$ *in Example 8.11 are invariants.*

Exercise 8.13 *Show that the mappings* $\mathcal{I}_1(d) = \mathsf{T}$ *for all* $d{:}D$ *and* $\mathcal{I}_2(d) = \mathsf{F}$ *for all* $d{:}D$ *are invariants for all LPEs.*

Exercise 8.14 *Consider the LPE*

$$Y(n{:}Nat) = \sum_{m:Nat} a{\cdot}Y(2m \dot{-} n) \lhd m+n<5 \rhd \delta$$

Give the "optimal" invariant $\mathcal{I}$ *for this LPE with* $\mathcal{I}(0) = \mathsf{T}$. *The same for* $\mathcal{I}(1) = \mathsf{T}$. *(Here, optimal means that* $\mathcal{I}(n) = \mathsf{F}$ *for as many* $n{:}Nat$ *as possible.)*

In Definition 8.4, CL-RSP ranged over the complete data set D. That is, if the LPE $X(d)$ and the processes $t(d)$ agreed for all $d{:}D$, then we could conclude $t(d) = X(d)$ for all $d{:}D$. Actually, it is sufficient to show that $X(d)$ and $t(d)$ agree for all *reachable* data elements, which can be characterised by some invariants. Of course, in this case we can only conclude $t(d) = X(d)$ for data elements d with $\mathcal{I}(d) = \mathtt{T}$. CL-RSP with invariants is similar to the derivation rule UFI-Inv of Rathke [93].

Definition 8.15 (CL-RSP with invariants) *Let the LPE*

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

be convergent, and let $\mathcal{I} : D \to Bool$ *be an invariant for this LPE. If* $t(d)$ *ranges over processes such that, for all* $d{:}D$ *with* $\mathcal{I}(d) = \mathtt{T}$,

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

then $t(d) = X(d)$ *for all* $d{:}D$ *with* $\mathcal{I}(d) = \mathtt{T}$.

Example 8.16 *Let* $even : Nat \to Bool$ *map even numbers to* $\mathtt{T}$ *and odd numbers to* $\mathtt{F}$. *Consider the following two LPEs:*

$$\begin{aligned} X(n{:}Nat) &= a(even(n)){\cdot}X(n+2) \\ Y &= a(\mathsf{T}){\cdot}Y \end{aligned}$$

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

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

is an invariant for the first LPE (cf. Example 8.11), and this LPE is convergent, by Definition 8.15

$$X(n) \;=\; Y$$

for even n.

Exercise 8.17 *Consider the process declarations*

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

and

$$\begin{array}{lcl} Y & = & a(\mathsf{T}) \cdot Z \\ Z & = & a(\mathsf{F}) \cdot Y \end{array}$$

Prove, using CL-RSP with invariants, that $X(n) = Y$ for natural numbers n that can be divided by six.

Chapter 9

Symbolic Methods

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

9.1 Cones and foci

The *cones and foci* technique of [62] aims to eliminate all hidden actions from an LPE. The main idea of this technique is that usually hidden events progress inertly towards a state in which no hidden actions 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. Imagine the state space forming a cone or funnel pointing towards the focus. Figure 9.1 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 technique, as otherwise the hidden actions in the cone would not be inert.

Assume an LPE X such that each of its states belongs to the cone of some focus point. In the cones and foci technique, the states of X are mapped to states of an LPE Y; intuitively, Y 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 between the two LPEs in question, and moreover that all states in a cone of X are mapped to the same state in Y. Informally, 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 $\phi(d) = \phi(d')$;

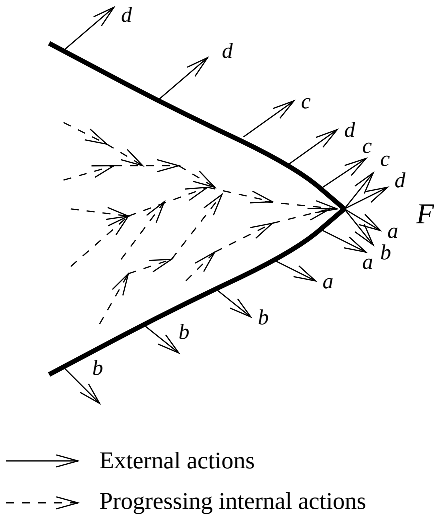


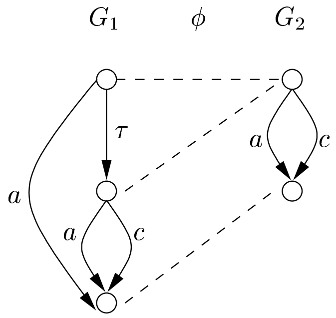
Figure 9.1: A focus point and its cone

- if $d \stackrel{a(\vec{e})}{\to} d'$, then $\phi(d) \stackrel{a(\vec{e})}{\to} \phi(d')$;

- if d is a focus point for X and $\phi(d) \stackrel{a(\vec{e})}{\to} d''$, then $d \stackrel{a(\vec{e})}{\to} d'$ with $\phi(d') = d''$.

Clearly, it is sufficient if the matching criteria are satisfied for the reachable states of X. In other words, the matching criteria above can be weakened by adding a condition that $\mathcal{I}(d) = \mathtt{T}$, where $\mathcal{I}$ is some invariant for X.

Example 9.1 *Below is depicted a mapping ϕ from the nodes in a graph G_1 to nodes in a graph G_2 without τ's. Note that each node in G_1 belongs to the cone of some focus point.*



ϕ satisfies the matching criteria. Note that ϕ establishes a branching bisimulation relation.

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

We proceed to give a more formal exposition on the cones and foci method.

Definition 9.2 (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

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

A state mapping $\phi : D \to D'$ satisfies the matching criteria *for $d \in D$ if for all $b \in \mathsf{Act}$:*

- $\forall e{:}E\,(h_\tau(d,e) \Rightarrow \phi(d) = \phi(g_\tau(d,e)))$;
- $\forall e{:}E\,(h_b(d,e) \Rightarrow (h'_b(\phi(d),e) \wedge f_b(d,e) = f'_b(\phi(d),e) \wedge \phi(g_b(d,e)) = g'_b(\phi(d),e)))$;
- $\forall e{:}E\,(\neg h_\tau(d,e)) \;\Rightarrow\; \forall e{:}E\,(h'_b(\phi(d),e) \Rightarrow h_b(d,e))$.

The first matching criterion in Definition 9.2 requires that all the states in a cone of X are mapped to the same state of Y. The second criterion expresses that each external action in X can be simulated. Finally, the third criterion expresses that if a state in Y can perform an external action, then the corresponding *focus points* in X can simulate this action; the so-called *focus condition* $\forall e{:}E\,(\neg h_\tau(d,e))$ captures that d must be a focus point.

Theorem 9.3 (General Equality Theorem) *Assume a convergent LPE X over D. Let $\mathcal{I}$ be an invariant for X. Furthermore, assume an LPE Y without hidden actions. If $\phi : D \to D'$ satisfies the matching criteria from Definition 9.2 for all $d{:}D$ with $\mathcal{I}(d) = \mathsf{T}$, then for all $d{:}D$ with $\mathcal{I}(d) = \mathsf{T}$:*

$$X(d) \leftrightarroweq_b Y(\phi(d)).$$

We recall that the invariant $\mathcal{I}$ in Theorem 9.3 provides information on the collection of reachable states of X. That is, if $\mathcal{I}(d_0) = \mathsf{T}$ and a state d_1 is reachable from d_0, then we know that $\mathcal{I}(d_1) = \mathsf{T}$.

Exercise 9.4 *Let*

$$\begin{array}{rcl} X(n{:}Nat) & = & \sum_{m:Nat}(a{\cdot}X(n+1) \lhd n = 3m \rhd \delta + d{\cdot}X(n+1) \lhd n = 3m+1 \rhd \delta \\ & & +\, \tau{\cdot}X(n+1) \lhd n = 3m+2 \rhd \delta) \\ Y(b{:}Bool) & = & a{\cdot}Y(\mathtt{F}) \lhd b \rhd \delta + d{\cdot}Y(\mathtt{T}) \lhd \neg b \rhd \delta \end{array}$$

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. Prove that the matching criteria are satisfied indeed.

Exercise 9.5 *The data type Nil consists of the single element nil. Let*

$$\begin{array}{lcl} X(n{:}Nat) & = & \sum_{m:Nat}(a{\cdot}X(n+2) \lhd n=3m \rhd \delta + d{\cdot}X(n+1) \lhd n=3m+1 \rhd \delta \\ & & +\, \tau{\cdot}X(n+1) \lhd n=3m+2 \rhd \delta) \\ Y(\eta{:}Nil) & = & a{\cdot}Y(\eta) \lhd \mathtt{T} \rhd \delta \end{array}$$

Verify that X is convergent. What is the focus condition for X? Give an invariant $\mathcal{I} : Nat \to Bool$ such that the matching criteria are satisfied for all $n{:}Nat$ with $\mathcal{I}(n) = \mathtt{T}$. (Of course the state mapping $\phi : Nat \to Nil$ is defined by $\phi(n) = nil$ for all $n{:}Nat$.)

9.2 Partial order reduction

Much research is devoted to algorithms that generate a reduced, but essentially equivalent, state space. Collectively, these methods are called partial order reduction methods.

Blom and van de Pol [19] showed how it can be detected that a τ-summand in an LPE (see Definition 8.1) 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, which often leads to a considerable reduction of the number of states. In some cases the number of states of this reduced state space can actually grow in a linear fashion with respect to the number of concurrent components in the original system.

Detection of confluent τ-summands Assume an LPE

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

We want to establish which τ-summands give rise to confluent τ-transitions. Suppose that $a_i = \tau$, and that there are transitions $d \xrightarrow{\tau} g_i(d,x)$ and $d \xrightarrow{a_j(f_j(d,y))} g_j(d,y)$ for some $x, y \in E$ (meaning that both $h_i(d,x)$ and $h_j(d,y)$ are true). We want to try and complete the picture of these two transitions of d in a confluent fashion; cf. Section 7.2. In principle, one could try all possible ways to complete this picture with any pair of transitions $g_i(d,x) \xrightarrow{a_j(f_j(g_i(d,x),v))} g_j(g_i(d,x),v)$ and $g_j(d,y) \xrightarrow{\tau} g_i(g_j(d,y),w)$. However, since we want to be able to derive the formulas resulting from this completion by means of an automated theorem prover, we prefer to avoid the existential quantification introduced by the v and the w, and only attempt the completion for $v = y$ and $w = x$. This choice is motivated by the fact that confluence in system behaviour is usually caused by two different concurrent components within a system that can perform independent transitions. Independence of these transitions implies that $v = y$ and $w = x$.

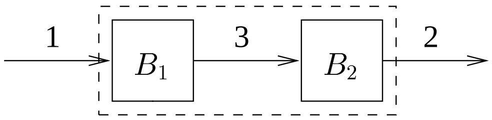
$$\begin{tikzcd}
d \arrow[r, "\tau"] \arrow[d, "a_j(f_j(d,y))"'] & g_i(d,x) \arrow[d, dashed, "a_j(f_j(g_i(d,x),y))"] \\
g_j(d,y) \arrow[r, dashed, "\tau"'] & {}
\end{tikzcd}$$

Thus we obtain the picture above. In case the transitions $d \xrightarrow{\tau} g_i(d,x)$ and $d \xrightarrow{a_j(f_j(d,y))} g_j(d,y)$ are distinct, we want to prove the presence of two more transitions (denoted by dashed arrows in the picture): $g_i(d,x) \xrightarrow{a_j(f_j(g_i(d,x),y))} g_j(g_i(d,x),y)$ and $g_j(d,y) \xrightarrow{\tau} g_i(g_j(d,y),x)$ with $f_j(d,y) = f_j(g_i(d,x),y)$ and $g_j(g_i(d,x),y) = g_i(g_j(d,y),x)$. These requirements are captured by the following formula. Let $a_i = \tau$. If $h_i(d,x) \wedge h_j(d,y)$ always implies

$$\begin{array}{ll}
 & h_j(g_i(d,x),y) \\
\wedge & h_i(g_j(d,y),x) \\
\wedge & f_j(d,y) = f_j(g_i(d,x),y) \\
\wedge & g_j(g_i(d,x),y) = g_i(g_j(d,y),x)
\end{array}
\quad \bigvee \quad
\begin{array}{ll}
 & a_j = \tau \\
\wedge & g_i(d,x) = g_j(d,y)
\end{array}$$

then summand i of the LPE is confluent.

Example 9.6 *We consider two unbounded queues in sequence (cf. the example of two one-bit buffers in sequence in Section 5.3).*



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

$$\begin{array}{rcl}
B_1(q{:}Queue) & = & \sum_{d:\Delta} r_1(d)\cdot B_1(in(d,q)) \\
 & + & s_3(toe(q))\cdot B_1(untoe(q)) \lhd \neg empty(q) \rhd \delta
\end{array}$$

$$\begin{array}{rcl}
B_2(q{:}Queue) & = & \sum_{d:\Delta} r_3(d)\cdot B_2(in(d,q)) \\
 & + & s_2(toe(q))\cdot B_2(untoe(q)) \lhd \neg empty(q) \rhd \delta
\end{array}$$

The initial state of the system is

$$\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1([]) \parallel B_2([])))$$

The specification of the data type Queue of queues over Δ that is used in the process declaration above is as follows. Here, $toe(q)$ denotes the bottom element of the queue q, $untoe(q)$ is the queue that results if the bottom element of q is *removed, and $empty(q)$ tests whether or not q is empty.*

$$\begin{array}{ll}
\textbf{sort} & Queue \\
\textbf{func} & [] :\to Queue \\
 & in : \Delta \times Queue \to Queue \\
\textbf{map} & toe : Queue \to \Delta \\
 & untoe : Queue \to Queue \\
 & empty : Queue \to Bool \\
 & if : Bool \times Queue \times Queue \to Queue \\
\textbf{var} & d : \Delta \\
 & q, q_1, q_2 : Queue \\
\textbf{rew} & toe(in(d,q)) = if(\neg empty(q), toe(q), d) \\
 & untoe(in(d,q)) = \\
 & \qquad if(\neg empty(q), in(d, untoe(q)), []) \\
 & empty([]) = \mathsf{T} \\
 & empty(in(d,q)) = \mathsf{F} \\
 & if(\mathsf{T}, q_1, q_2) = q_1 \\
 & if(\mathsf{F}, q_1, q_2) = q_2
\end{array}$$

The external behaviour of the two parallel unbounded queues is again an unbounded queue. Clearly, the resulting state space is infinite.

The resulting convergent LPE is (cf. Section 8.2):

$$\begin{array}{rcl}
X(q_1, q_2) & = & \sum_{d:\Delta} r_1(d) \cdot X(in(d, q_1), q_2) \\
 & + & \tau \cdot X(untoe(q_1), in(toe(q_1), q_2)) \lhd \neg empty(q_1) \rhd \delta \\
 & + & s_2(toe(q_2)) \cdot X(q_1, untoe(q_2)) \lhd \neg empty(q_2) \rhd \delta
\end{array}$$

We compute the confluence formulas.

1. *Commutation of τ and $r_1(d)$:*

$$\begin{array}{rcl}
\neg empty(q_1) & \Rightarrow & \neg empty(in(d, q_1)) \\
 & \wedge & in(d, untoe(q_1)) = untoe(in(d, q_1)) \\
 & \wedge & in(toe(q_1), q_2) = in(toe(in(d, q_1)), q_2)
\end{array}$$

2. *Commutation of τ and $s_2(toe(q_2))$:*

$$\begin{array}{rcl}
\neg empty(q_1) \wedge \neg empty(q_2) & \Rightarrow & \neg empty(in(toe(q_1), q_2)) \wedge \neg empty(q_1) \\
 & \wedge & toe(in(toe(q_1), q_2)) = toe(q_2) \\
 & \wedge & untoe(in(toe(q_1), q_2)) = in(toe(q_1), untoe(q_2))
\end{array}$$

It is not difficult to see that these confluence formulas can be derived from the equational specification of the data type Queue. Hence, the τ-summand of LPE X is convergent.

We note that it would not be possible to derive the confluence formulas with the automated theorem prover if we had used the standard definitions of toe and *untoe:*

$$\begin{array}{l} toe(in(d,[])) = d \\ toe(in(d,in(e,q))) = toe(in(e,q)) \\ untoe(in(d,[])) = [] \\ untoe(in(d,in(e,q))) = in(d,untoe(in(e,q))) \end{array}$$

Namely, with this definition, the formulas

$$\begin{array}{lcl} \neg empty(q) & \Rightarrow & toe(in(d,q)) = toe(q) \\ \neg empty(q) & \Rightarrow & untoe(in(d,q)) = in(d,untoe(q)) \end{array}$$

can only be proved using induction.

Exercise 9.7 *Consider the following LPEs. In each case, show whether or not the confluence formula for the τ-summand is true. If not, show that the τ-transitions generated by this summand are not always confluent.*

(1)

$$\begin{array}{rcl} X(n:Nat) & = & \tau{\cdot}X(n) \lhd even(n) \rhd \delta \\ & + & a(n){\cdot}X(S(S(n))) \lhd \mathsf{T} \rhd \delta \end{array}$$

(2)

$$\begin{array}{rcl} X(n:Nat) & = & \tau{\cdot}X(S(n)) \lhd even(n) \rhd \delta \\ & + & a(n){\cdot}X(S(S(n))) \lhd \mathsf{T} \rhd \delta \end{array}$$

(3)

$$\begin{array}{rcl} X(n:Nat) & = & \tau{\cdot}X(S(n)) \lhd \mathsf{T} \rhd \delta \\ & + & a{\cdot}X(S(S(n))) \lhd even(n) \rhd \delta \end{array}$$

Automated theorem prover A theorem prover [92] within the μCRL toolset, for Boolean combinations over a user-defined algebraic data type, makes it possible to prove such (large) formulas in an automated fashion. (It is based on an extension of so-called binary decision diagrams with equality [57].) The theorem prover is of course 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 8.3), 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.

Representative states For each reachable state d we compute a *representative* state $repr(d)$, such that:

- if $d \xrightarrow{\tau} d'$ is confluent, then $repr(d) = repr(d')$;
- each state d can evolve to $repr(d)$ 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 is found. (Terminal means that for each state d in the strongly connected component and for each confluent transition $d \xrightarrow{\tau} d'$, we have that d' 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.

State space generation modulo confluence Blom [17] showed how confluence can be exploited during state space generation. In this state space generation algorithm, only representatives of states are generated. That is, if d_0 is the initial state, then we start state space generation from $repr(d)$. Moreover, if a state d is added to the state space, then for each transition $d \xrightarrow{a} d'$ we add the state $repr(d')$ and the transition $d \xrightarrow{a} repr(d')$ to the generated state space.

The state spaces generated by the 'standard' algorithm and by the partial order reduction technique 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 $repr(d') = repr(d'')$.

A prototype of this generation algorithm was implemented, and experience learns that 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 algorithm. In the table below, 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 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

Symbolic priorisation If a τ-summand in a *convergent* LPE is confluent, then it can be given priority over other summands. (In Section 7.2 it was shown that in the presence of τ-loops, such a priorisation of τ's may be unsound.) In other words, the negation of the condition of the confluent τ-summand can be aded 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.

In the case of the two parallel unbounded buffers, the result is

$$\begin{array}{rcl} X(q_1, q_2) & = & \sum_{d:\Delta} r_1(d)\cdot X(in(d, q_1), q_2) \lhd empty(q_1) \rhd \delta \\ & + & \tau\cdot X(untoe(q_1), in(toe(q_1), q_2)) \lhd \neg empty(q_1) \rhd \delta \\ & + & s_2(toe(q_2))\cdot X(q_1, untoe(q_2)) \lhd\ empty(q_1) \wedge \neg empty(q_2) \rhd \delta \end{array}$$

So if the first queue q_1 is not empty, then only the τ-summand can be executed.

9.3 Elimination of parameters and sum variables

The μCRL toolset comprises five tools (`rewr`, `constelm`, `sumelm`, `parelm`, and `structelm`) [51] that target the automated simplification of LPEs by eliminating parameters and sum variables that do not influence the LPE's behaviour. These optimisations can be viewed as special cases of abstract interpretation [34, 35].

At first sight the underlying algorithms may seem somewhat simplistic, because in principle a sensible μCRL specification will be free of parameters and sum variables that are inert or that remain constant throughout any execution sequence. However, keep in mind that LPEs are usually produced by means of automated linearisation algorithms, were 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

$$\begin{array}{rcl} X(x_1{:}D_1, \ldots, x_m{:}D_m) & = & \sum_{i=1}^{k} \sum_{y_1^i:E_1^i} \cdots \sum_{y_{n_i}^i:E_{n_i}^i} a_i(f_i(\vec{x}, \vec{y^i}))\cdot \\ & & \qquad X(g_1^i(\vec{x}, \vec{y^i}), \ldots, g_m^i(\vec{x}, \vec{y^i})) \lhd\ h_i(\vec{x}, \vec{y^i}) \rhd \delta \end{array}$$

Here, we spell out the parameters $x_1, \ldots, x_m$ (abbreviated to $\vec{x}$) of the LPE and the sum variables $y_1^i, \ldots, y_{n_i}^i$ (abbreviated to $\vec{y^i}$) of the i-th 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.

Rewriting data terms The data terms occurring in an LPE can be rewritten using the equations of the data types. If a condition is rewritten to false, then the corresponding summand in the LPE is removed. This tool is called `rewr`.

Elimination of constant parameters A parameter of an LPE can be replaced by its initial value if it can be determined that this parameter remains constant throughout any run of the process. This elimination of constant parameters does not reduce the resulting state space, but it may considerably shorten the time needed to generate a state space from an LPE. Furthermore, this elimination can make other simplification tools more effective.

The algorithm underlying `constelm` works as follows. Throughout the algorithm, some parameters are marked and some are not, where unmarked parameters are candidates for elimination, while marked parameters are no longer candidates. At any time, the substitution σ is defined by $\sigma(x_j) = d_j$ if x_j is unmarked and $\sigma(x_j) = x_j$ if x_j is marked, for $j = 1, \ldots, m$. Initially, all 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 arguments at unmarked parameter positions preserve these initial values. To be more precise, for each unmarked parameter x_j and each summand $i = 1, \ldots, k$ it is checked whether

$$\sigma(h_i(\vec{x}, \vec{y^i}) \Rightarrow eq(g_j^i(\vec{x}, \vec{y^i}), d_j))$$

rewrites to `T` (using `rewr`). If this check fails for some unmarked parameter x_j and some i, then this parameter is marked. Namely, then it cannot be guaranteed that the value of x_j remains constant. Substitution σ is adapted accordingly, i.e., $\sigma(x_j)$ is changed from d_j to x_j.

The above check is repeated until it succeeds for all unmarked parameters and for all summands. (Note that after each fail, the check must be repeated for all unmarked summands, because after a fail the substitution σ is adapted.) When the check succeeds, we conclude that the unmarked parameters remain at their initial values throughout any run 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 bisimulation equivalence, 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 parameters and k the number of summands in the LPE.

Exercise 9.8 *Apply* `constelm` *to the LPE*

$$\begin{array}{rcl} X(w{:}Nat, x{:}Nat, y{:}Nat, z{:}Nat) & = & a(x)\cdot X(x, w, z, y) \lhd eq(y, 0) \rhd \delta \\ & + & b(y)\cdot X(S(0), x, 0, y + z) \end{array}$$

with initial state $X(0, 0, 0, 0)$. *Which parameters are found to be constant? What is the resulting LPE?*

Elimination of inert parameters A parameter of an LPE that has no (direct or indirect) influence on the parameters of actions and on conditions does not influence the LPE's behaviour and can be removed. Elimination of inert parameters may lead to a substantial reduction of the state space underlying an LPE. If the inert parameter ranges over an infinite domain, the number of states can even reduce from infinite to finite by this operation. This typically happens after hiding part of the behaviour of a system.

The algorithm underlying `parelm` works as follows. Throughout the algorithm, some parameters are marked and some are not, where unmarked parameters are candidates for elimination, while marked parameters are no longer candidates. Initially, all parameters in the LPE are unmarked. A parameter is marked if it occurs at one of the following three kinds of places:

1. in a condition $h_i(\vec{x},\vec{y^i})$ for some $i \in \{1,\ldots,k\}$;
2. in an argument $f_i(\vec{x},\vec{y^i})$ for some $i \in \{1,\ldots,k\}$; or
3. in an argument $g^i_j(\vec{x},\vec{y^i})$ for some $i \in \{1,\ldots,k\}$ and $j \in \{1,\ldots,m\}$, where the j-th parameter is marked.

In the first two cases, the parameter under scrutiny has direct influence on the LPE's behaviour. In the third case, it influences the value of parameter x_j, which in turn was found to influence the LPE's behaviour.

When no further parameter can be marked, we conclude that the unmarked parameters are inert. This means that all unmarked parameters can be eliminated from the parameter list of the LPE. Since at the right-hand side of the LPE these unmarked parameters only occur in arguments $g^i_j(\vec{x},\vec{y^i})$ where the j-th parameter is unmarked, this elimination effectively removes all occurrences of unmarked parameters from the LPE.

`parelm` preserves bisimulation equivalence. The worst-case time complexity of the algorithm is $O(mk)$.

Exercise 9.9 *Apply* `parelm` *to the LPE*

$$X(x{:}D,y{:}D,z{:}D) \;=\; a\cdot X(x,z,y) \;+\; \sum_{w:D} b(z)\cdot X(w,y,z)$$

Which parameters are found to be inert? What is the resulting LPE?

Elimination of sum variables If a sum variable is inert, 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 can be replaced by this datum.

If a sum variable only occurs below the summation sign where it is declared, then it 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 parameters and obsolete conditions have been eliminated beforehand. For example, after application of `parelm` to the LPE in Exercise 9.9, the inert sum variable w can be eliminated.

Furthermore, if the data type of a sum variable consists of a single constant, then clearly the range of values of this sum variable is restricted to this constant.

So then the sum variable can be replaced by this constant, and the corresponding summation sign can be eliminated.

The third and most interesting apect of `sumelm` is that it tries to detect for each condition $h_i(\vec{x}, \vec{y^i})$ 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 this 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.

$$\begin{array}{lcl}
\mathit{Values}(y, \mathit{eq}(y,t)) & = & \{t\} \text{ if } y \text{ does not occur in } t\\
\mathit{Values}(y, \mathit{eq}(t,y)) & = & \{t\} \text{ if } y \text{ does not occur in } t\\
\mathit{Values}(y, b_1 \wedge b_2) & = & \mathit{Values}(y, b_1) \cup \mathit{Values}(y, b_2)\\
\mathit{Values}(y, b_1 \vee b_2) & = & \mathit{Values}(y, b_1) \cap \mathit{Values}(y, b_2)\\
\mathit{Values}(y, b) & = & \emptyset \text{ otherwise}
\end{array}$$

Conditions $\mathit{eq}(y,t)$ and $\mathit{eq}(t,y)$ imply that y is equal to t. If both b_1 and b_2 hold, then y is equal to terms that occur either in $\mathit{Values}(y, b_1)$ or in $\mathit{Values}(y, b_2)$. If either b_1 or b_2 holds, then y is guaranteed to be equal to 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 $\mathit{eq}(s,t)$ with $s \in \mathit{Values}(y, b_1)$ and $t \in \mathit{Values}(y, b_2)$, to determine which data terms in these two sets are equal.

If for some $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, n_i\}$ the set $\mathit{Values}(y^i_j, h_i(\vec{x}, \vec{y^i}))$ is not empty, then we conclude that $h_i(\vec{x}, \vec{y^i})$ guarantees that the value of y^i_j is equal to any element in $\mathit{Values}(y^i_j, h_i(\vec{x}, \vec{y^i}))$. So we select an element t from this set, and replace all occurrences of y^i_j in $f_i(\vec{x}, \vec{y^i})$ and $g^i_\ell(\vec{x}, \vec{y^i})$ for $\ell = 1, \ldots, m$ by t. Since y^i_j does not occur in t, the summation sign for y^i_j can then be eliminated from the LPE.

`sumelm` preserves bisimulation equivalence. The worst-case time complexity of the algorithm is $O((n_1 + \cdots + n_k)k)$.

Example 9.10 *We assume the data types Bool with disjunction, Bit consisting of* $\{0, 1\}$*, and* D *consisting of* $\{d_1, d_2\}$*. As usual, the three data types are supplied with an equality function. Consider the LPE*

$$\begin{array}{lcl}
X(d{:}D, b{:}\mathit{Bit}) & = & \sum_{d':D} a{\cdot}X(d', b) \lhd \mathit{eq}(d, d_2) \vee \mathit{eq}(b, 0) \rhd \delta\\
& + & \sum_{b':\mathit{Bit}} c{\cdot}X(d, b') \lhd \mathit{eq}(b', 0) \rhd \delta
\end{array}$$

The initial state is $X(d_1, 0)$.

- `constelm` *does not change the LPE, due to the occurrences of* d' *and* b' *in the first and second argument of* X*, respectively.*

  Neither does `parelm` *change the LPE, because the parameters* d *and* b *both occur in a condition.*

We apply `sumelm` *to the LPE. Owing to the condition* $eq(b', 0)$ *in the second summand, the sum variable* b' *is replaced by* 0 *and the summation sign for* b' *is removed. The resulting LPE is*

$$\begin{array}{rcl} X(d{:}D, b{:}Bit) & = & \sum_{d':D} a{\cdot}X(d', b) \lhd eq(d, d_2) \vee eq(b, 0) \rhd \delta \\ & + & c{\cdot}X(d, 0) \lhd eq(0, 0) \rhd \delta \end{array}$$

- `parelm` *still does not change the LPE, because the parameters* d *and* b *both occur in a condition.*

  We apply `constelm` *to the LPE. It marks the parameter* d *(due to the occurrence of* d' *in the first argument of* X*), but leaves the parameter* b *unmarked. So the latter parameter is replaced by its initial value 0 and removed from the parameter list of* X*. The resulting LPE is*

$$\begin{array}{rcl} X(d{:}D) & = & \sum_{d':D} a{\cdot}X(d') \lhd eq(d, d_2) \vee eq(0, 0) \rhd \delta \\ & + & c{\cdot}X(d) \lhd eq(0, 0) \rhd \delta \end{array}$$

- *We apply* `rewr` *to the LPE. The resulting LPE is*

$$\begin{array}{rcl} X(d{:}D) & = & \sum_{d':D} a{\cdot}X(d') \lhd \mathtt{T} \rhd \delta \\ & + & c{\cdot}X(d) \lhd \mathtt{T} \rhd \delta \end{array}$$

- *We apply* `parelm` *to the LPE. Since the parameter* d *occurs neither in conditions nor in arguments of actions, it is found to be inert. The resulting LPE is*

$$\begin{array}{rcl} X & = & \sum_{d':D} a{\cdot}X \lhd \mathtt{T} \rhd \delta \\ & + & c{\cdot}X \lhd \mathtt{T} \rhd \delta \end{array}$$

- *Finally, we apply* `sumelm` *to the LPE once again. The sum variable* d' *is found to be inert. The resulting LPE is*

$$\begin{array}{rcl} X & = & a{\cdot}X \lhd \mathtt{T} \rhd \delta \\ & + & c{\cdot}X \lhd \mathtt{T} \rhd \delta \end{array}$$

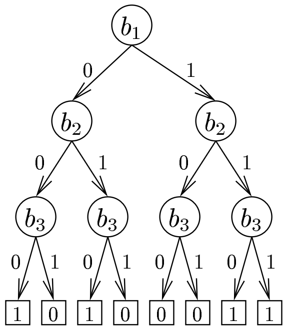
Elimination of data structures Sometimes the aforementioned simplifications cannot be applied to parameters and sum variables in full generality, but they can be applied to parts of the data structures that these variables range over. For this to take place, such data structures must first be partitioned into their constituents. The tool `structelm` performs such a partitioning. The algorithm underlying `structelm` is omitted here, as it is quite hairy and intricate. We refer to [51] for the ideas and algorithm underlying `structelm`.

9.4 Ordered binary decision diagrams

A state space can be represented as a mapping from a sequence of booleans to the booleans. In this context it is usance to write true as 1 and false as 0. Suppose the number of states does not exceed 2^n, for some $n > 0$. Then each state can be represented uniquely by a sequence over $\{0,1\}$ of length n. Moreover, the state space can be captured by mappings $\varphi_a : \{0,1\}^{2n} \to \{0,1\}$ for $a \in \mathsf{Act}$, 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. Thus, in order to capture state spaces in a compact fashion, we are interested in compressed representations of the mappings φ_a; this observation is due to McMillan [83].

For a start, we note that a mapping $\varphi(b_1,\ldots,b_m)$ from $\{0,1\}^m$ to $\{0,1\}$ 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 b_{i+1}, 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 $\{0,1\}$. Intuitively, starting at the root of the binary decision tree for ϕ, the values $\mathtt{b}_1,\ldots,\mathtt{b}_m$ of $b_1,\ldots,b_m$ are assessed consecutively, and the edge taken from a node associated with b_i depends on its value $\mathtt{b}_i$; finally, the leaf at the end of this path is labelled $\varphi(\mathtt{b}_1,\ldots,\mathtt{b}_n)$.

Example 9.11 *The boolean formula $(b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_3)$ takes the form*



(For the sake of clarity, non-leaves are depicted as circles, while leaves are depicted as boxes.)

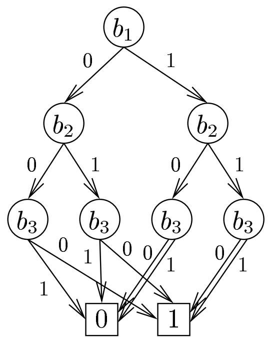
An *ordered binary decision diagram* (OBDD), introduced by Bryant [25], is obtained by maximal sharing of nodes in a binary decision tree. First of all, the leaves labelled 0 are collapsed, and likewise for the leaves labelled 1. Next, two minimisation steps are performed, until neither of them can be 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.

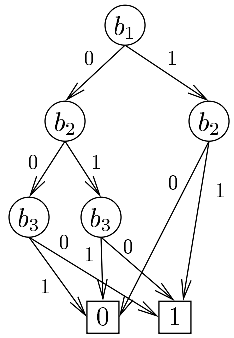
- If the 0-transition and the 1-transition of a non-leaf ν lead to the same node ν', then ν can be eliminated, where all its incoming edge are redirected to ν'.

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 $\{0,1\}^m$ to $\{0,1\}$ boils down to checking isomorphism of the corresponding OBDDs.

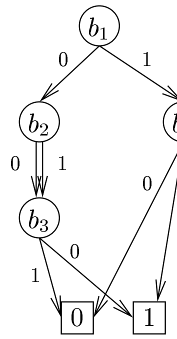
Example 9.12 *As an example we transform the binary decision tree from Example 9.11 into an OBDD. First, leaves with the same label are collapsed.*



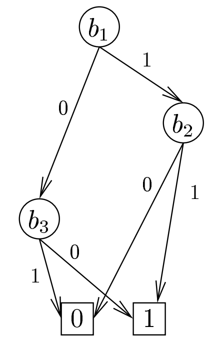
Next, the two rightmost nodes associated with b_3 can be eliminated, because for both nodes, their 0- and 1-transition lead to the same node.



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.

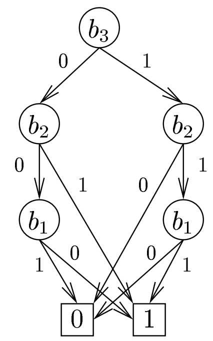


Finally, the leftmost node associated with b_2 *can be eliminated, because its 0- and 1-transition lead to the same node. The result is an OBDD.*



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 9.13 *Starting from the reverse ordering* b_3, b_2, b_1*, the OBDD for* $(b_1 \wedge b_2) \vee (\neg b_1 \wedge \neg b_3)$ *would become*



The chosen ordering can have a huge impact on the number of nodes in the resulting OBDD. Assume a family of mappings $\varphi_m(b_1, \ldots, b_m)$ from $\{0,1\}^m$

to $\{0,1\}$ for $m > 0$. While one ordering could produce OBDDs that grow in a polynomial fashion (with respect to m), another ordering could produce OBDDs that grow exponentially. Moreover, some families grow in an exponential fashion for any ordering on variables. Given a mapping $\varphi : \{0,1\}^m \to \{0,1\}$, finding an ordering on its variables to obtain an OBDD of minimal size is an NP-complete problem [26], which suggests that this problem cannot always be solved in polynomial time.

Given two OBDDs $\mathcal{O}$ and $\hat{\mathcal{O}}$ over the same (ordered) 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 [25]. The OBDD for $\neg\mathcal{O}$ is obtained by simply swapping 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 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 variable b, while either $\hat{\mathcal{O}}$ is a single leaf or the root of $\hat{\mathcal{O}}$ is associated with a 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 variable b, while either $\mathcal{O}$ is a single leaf or its root is associated with a 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 single leaf labelled 1. Otherwise, it is a single 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 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 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.

The result of computing $\mathcal{O} \wedge \hat{\mathcal{O}}$, $\mathcal{O} \vee \hat{\mathcal{O}}$, $\exists b\, \mathcal{O}$ or $\forall b\, \mathcal{O}$ is in general not yet an OBDD. Therefore, as a final step, the aforementioned minimisation procedure has to be applied to obtain an OBDD.

9.5 Symbolic model checking

In Section 7.3, the μ-calculus was presented, together with an algorithm for computing which states in a state space satisfy a given μ-calculus formula.

At the start of Section 9.4 it was explained how a state space with at most 2^n states can be captured by OBDDs $\mathcal{O}_a : \{0,1\}^{2n} \to \{0,1\}$ for $a \in \mathsf{Act}$, 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. 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 9.4). Let *FALSE* and *TRUE* denote the OBDD (over $b_1,\ldots,b_n$) consisting of a single leaf with the label 0 and 1, respectively.

$$
\begin{array}{lcl}
\mathcal{B}(\mathtt{T}) & = & \mathit{TRUE} \\
\mathcal{B}(\neg\phi) & = & \neg\mathcal{B}(\phi) \\
\mathcal{B}(\phi \wedge \phi') & = & \mathcal{B}(\phi) \wedge \mathcal{B}(\phi') \\
\mathcal{B}(\langle a\rangle\phi) & = & \exists b_1' \cdots \exists b_n'\, (\mathcal{O}_a(b_1,\ldots,b_n,b_1',\ldots,b_n') \wedge \mathcal{B}'(\phi)) \\
\mathcal{B}(\mu X.\phi) & = & \mathit{FIX}(\phi, X{=}\mathit{FALSE}) \\
\mathcal{B}(\nu X.\phi) & = & \mathit{FIX}(\phi, X{=}\mathit{TRUE})
\end{array}
$$

At the right-hand side of the fourth case, $\mathcal{B}'(\phi)$ is an OBDD over $b_1',\ldots,b_n'$, calculated against the OBDDs $\mathcal{O}_a(b_1',\ldots,b_n',b_1'',\ldots,b_n'')$ for $a \in \mathsf{Act}$. In the last two cases, the construct $\mathit{FIX}(\phi, X{=}\mathcal{O})$ operates as follows (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 7.3). Given an OBDD $\mathcal{O}$ over $b_1,\ldots,b_n$, $\mathit{FIX}(\phi, X{=}\mathcal{O})$ computes an OBDD $\mathcal{O}'$ over $b_1,\ldots,b_n$, and continues to compute $\mathit{FIX}(\phi, X{=}\mathcal{O}')$, until it reaches a fixpoint (i.e., until $\mathit{FIX}(\phi, X{=}\mathcal{O})$ equals $\mathcal{O}$).

Chapter 10

Protocol Verifications

In this chapter, verifications are presented of the SWP and Implementation A of the TIP, using the techniques developed in the previous chapter.

The process behaviours of the ABP and of the BRP are quite independent of their data parts. Thus, the verifications of the ABP and of the BRP do not really require the machinery that has been developed in Chapter 8. Therefore, these verifications are omitted here; the reader is referred to [43] for expositions on these verifications. A verification of Implementation B of the TIP can be found in [97].

10.1 Tree identify protocol

In this section we give a formal proof that for all connected networks that are free of cycles, Implementation A of the TIP (see Section 6.4) produces a unique root. This proof uses the verification techniques explained in Chapter 8.

We recall that Implementation A, without parametrisation of *leader* actions, consists of node processes

$$\begin{array}{rcl} X(i,p,0) & = & \sum_{j:Nodelist} r(j,i)\cdot X(i,p\backslash\{j\},0) \lhd j\in p \rhd \delta \\ & + & \sum_{j:Nodelist} s(i,j)\cdot X(i,p,1) \lhd p=\{j\} \rhd \delta \\ X(i,[],0) & = & \mathit{leader}\cdot X(i,[],1) \end{array}$$

Here, i is the identifier of the node; p is the list of possible parents of node i; and each node is in 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

$$\tau_I(\partial_H(X(n_0,p_0[n_0],0)\|\cdots\|X(n_k,p_0[n_k],0)))$$

where $p_0[i]$ consists of the neighbours of i in the network. The aim of this section is to show that the external behaviour of this process term, for any connected network without cycles, is $\tau\cdot\mathit{leader}\cdot\delta$. Actually, if the network consists of a single node, then no parent requests are exchanged, so in that special case the external behaviour is $leader \cdot \delta$.

By CL-RSP, one can prove that for mappings p from nodes to lists of nodes and mappings s from nodes to $\{0,1\}$,

$$\tau_I(\partial_H(X(n_0,p[n_0],s[n_0])\|\cdots\|X(n_k,p[n_k],s[n_k])))$$

is equal to $Y(p,s)$, defined by the LPE

$$\begin{array}{l} Y(p{:}Nodelistlist, s{:}Statelist) \ = \\ \sum_{i,j:Node} \tau \cdot Y(p[i] := p[i] \backslash \{j\}, s[j] := 1) \lhd\ j \in p[i] \wedge p[j] = \{i\} \wedge s[i] = s[j] = 0 \rhd \delta \\ +\ \sum_{i:Node} leader \cdot Y(p, s[i] := 1) \lhd\ empty(p[i]) \wedge s[i] = 0 \rhd \delta \end{array}$$

See Example 8.6 and Exercise 8.9 for examples of how to derive such an equality, using CL-RSP.

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 execution sequence of hidden actions.

Exercise 10.1 *Prove* $\tau_I(\partial_H(X(n_0,p[n_0],s[n_0])\|\cdots\|X(n_k,p[n_k],s[n_k]))) = Y(p,s)$ *for pairs* (p,s).

We list four invariants for the LPE Y, which are universally quantified over i and j:

$$\mathcal{I}_1(p): \quad j \in p[i] \vee i \in p[j]$$

$$\mathcal{I}_2(p,s): \ (j \notin p[i] \wedge i \in p[j]) \Rightarrow s[j] = 1$$

$$\mathcal{I}_3(p,s): \ s[j] = 1 \Rightarrow (empty(p[j]) \vee singleton(p[j]))$$

$$\mathcal{I}_4(p,s): \ (j \in p[i] \wedge s[i] = 0) \Rightarrow (i \in p[j] \wedge s[j] = 0)$$

Note that $\mathcal{I}_n(p_0, s_0) = \mathtt{T}$ for $n = 1,2,3,4$. We show that the first three formulas are invariants for Y.

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 $s[j] = 1$, then after performing an action still $s[j] = 1$.

   Suppose $s[j] = 0$ and $j \in p[i]$, 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 $s[j] = 0$.

Exercise 10.2 *Prove that $\mathcal{I}_4$ is an invariant for Y.*

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 10.3 (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

$$p[j] = \{i\} \;\wedge\; j \in p[i] \;\wedge\; s[i] = s[j] = 0$$

The LPE for the external behaviour is

$$Z(b{:}\mathit{Bool}) \;=\; \mathit{leader}\cdot Z(\mathsf{F}) \triangleleft b \triangleright \delta$$

Clearly, $Z(\mathtt{T}) = leader{\cdot}\delta$ and $Z(\mathtt{F}) = \delta$.

We define the *state mapping* ϕ from pairs (p, s) to *Bool* by

$$\phi(p,s) = \begin{cases} \mathsf{T} & \text{if } s[i] = 0 \text{ for some node } i \\ \mathsf{F} & \text{if } s[i] = 1 \text{ for all nodes } i \end{cases}$$

Then the matching criteria from Definition 9.2, applied to the LPEs Y and Z, produce the following four formulas:

$$\forall i,j{:}Node\,((j \in p[i] \wedge p[j] = \{i\} \wedge s[i] = s[j] = 0) \Rightarrow \\ \phi(p,s) = \phi(p[i] := p[i]\backslash\{j\}, s[j] := 1));$$

$$\forall i{:}Node\,((empty(p[i]) \wedge s[i] = 0) \Rightarrow \phi(p,s));$$

$$\forall i{:}Node\,((empty(p[i]) \wedge s[i] = 0) \Rightarrow \phi(p, s[i] := 1) = \mathsf{F});$$

$$\forall i,j{:}Node\,((p[j] \neq \{i\} \vee j \notin p[i] \vee s[i] = 1 \vee s[j] = 1 \vee i = j) \wedge \phi(p,s)) \Rightarrow \\ \exists i'{:}Node\,(empty(p[i']) \wedge s[i'] = 0).$$

We show that these four formulas are true.

- $\phi(p,s) = \mathsf{T} = \phi(p[i]\backslash\{j\}, s[j] := 1)$, because $s[i]$ remains 0.
- $\phi(p,s) = \mathsf{T}$, because $s[i] = 0$.
- By uniqueness of the root, $empty(p[i])$ implies $s[j] = 1$ for $j \neq i$. Hence, $\phi(p, s[i] := 1) = \mathsf{F}$.
- Since $\phi(p,s) = \mathsf{T}$ there is a node i' with $s[i'] = 0$. Suppose $p[i']$ is not 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 $p[j] \neq \{i\}$, so there is a $k \neq i$ with $k \in p[j]$. Then similarly $s[k] = 0$ and there is an $\ell \neq j$ with $\ell \in p[k]$, etc. This contradicts the fact that there is no cycle.

By the General Equality Theorem, if p establishes a connected network without cycles, then

$$Y(p,s) \;\underline{\leftrightarrow}_b\; Z(\phi(p,s))$$

This implies

$$\begin{array}{rl} & \tau_I(\partial_H(X(n_0, p_0[n_0], 0) \| \cdots \| X(n_k, p_0[n_k], 0))) \\ \underline{\leftrightarrow}_{rb} & Y(p_0, s_0) \\ \underline{\leftrightarrow}_b & Z(\mathsf{T}) \\ \underline{\leftrightarrow}_{rb} & leader{\cdot}\delta \end{array}$$

10.2 Sliding window protocol

Fokkink, Groote and Pang

Chapter 11

Timed Systems

11.1 Syntax and semantics of timed μCRL

Groote, Reniers, van Wamel and van der Zwaag

11.2 From timed to untimed μCRL

Reniers and Usenko

11.3 Analysis of three hybrid systems

Groote and van Wamel

Chapter 12

The μCRL Toolset in a Nutshell

The μCRL toolset [18, 105] (see `http://www.cwi.nl/~mcrl`) supports the analysis and manipulation of μCRL specifications. A μCRL specification can be automatically transformed into an LPE that is stored in a binary format or as a plain text file. The resulting LPE and its data structures are stored as ATerms. The ATerm library [22] stores terms in a very compact way by minimal memory requirements, employing maximal sharing, and using a tailor-made garbage collector. Moreover, the ATerm library uses a file format that is even more compact than the memory format. All other tools in the μCRL toolset use LPEs as their starting point. The simulator allows the interactive simulation of an LPE. There are a number of tools that allow optimisations on the level of LPEs. The instantiator generates a state space from an LPE (under the condition that it is finite-state), and the resulting state space can be visualised, analysed and minimised.

An overview of the μCRL toolset is presented in Figure 12.1. This picture is divided into three layers: μCRL specifications, LPEs and state spaces. 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 three outgoing dashed arrows actually belong to the CADP toolset [44].

The μCRL toolset was successfully used to analyse a wide range of protocols and distributed algorithms. Recently it was used to support the optimised redesign of the Transactions Capabilities Procedures in the SS No. 7 protocol stack for telephone exchanges [2, 3], to detect a number of mistakes in a real-life proto-

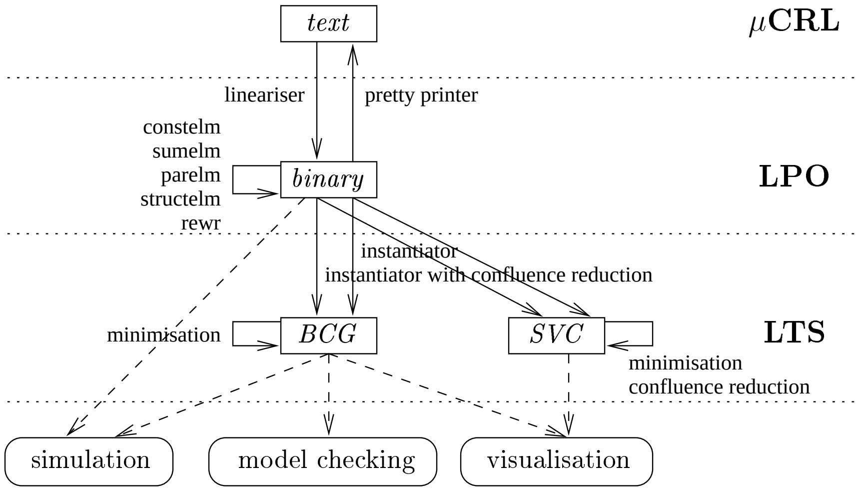


Figure 12.1: The main components of the μCRL toolset

col over the CAN bus for lifting trucks [54], to analyse a leader election protocol from the Home Audio/Video interoperability (HAVi) architecture [103], and to perform scenario-based verifications of the coordination language SPLICE [38].

Solutions to Exercises

2.2	**map**	$\wedge, \Rightarrow, \Leftrightarrow : Bool \times Bool \to Bool$
		$\neg : Bool \to Bool$
		$\dot{-} : Nat \times Nat \to Nat$
	var	$x{:}Bool$
		$n, m{:}Nat$
	rew	$x \vee \mathtt{T} = \mathtt{T}$
		$x \vee \mathtt{F} = x$
		$\neg\, \mathtt{T} = \mathtt{F}$
		$\neg\, \mathtt{F} = \mathtt{T}$
		$\mathtt{T} \Rightarrow x = x$
		$\mathtt{F} \Rightarrow x = \mathtt{T}$
		$\mathtt{T} \Leftrightarrow x = x$
		$\mathtt{F} \Leftrightarrow x = \neg x$
		$0 \dot{-} n = 0$
		$n \dot{-} 0 = n$
		$S(n) \dot{-} S(m) = n \dot{-} m$
2.3	**map**	$\geq : Nat \times Nat \to Bool$
		$< : Nat \times Nat \to Bool$
		$> : Nat \times Nat \to Bool$
	var	$n, m{:}Nat$
	rew	$n \geq 0 = \mathtt{T}$
		$0 \geq S(n) = \mathtt{F}$
		$S(n) \geq S(m) = n \geq m$
		$0 < S(n) = \mathtt{T}$
		$n < 0 = \mathtt{F}$
		$S(n) < S(m) = n < m$
		$S(n) > 0 = \mathtt{T}$
		$0 > n = \mathtt{F}$
		$S(n) > S(m) = n > m$
2.4	**map**	$even : Nat \to Bool$
	var	$m, n{:}Nat$
	rew	$even(0) = \mathtt{T}$
		$even(S(n)) = \neg even(n)$
		$power(m, 0) = S(0)$
		$power(m, S(n)) = mul(power(m, n), m)$
2.5	**sort**	$Bool,\ Nat,\ List,\ D$
	func	$[] :\to List$
		$in : D \times List \to List$
	map	$append : D \times List \to List$
		$top, toe : List \to D$
		$tail, untoe : List \to List$
		$nonempty : List \to Bool$

$++ : List \times List \to List$

var $d, e{:}D,\ q, q'{:}List$

rew $append(d, []) = in(d, [])$

$append(d, in(e, q)) = in(e, append(d, q))$

$top(in(d, q)) = d$

$toe(in(d, [])) = d$

$toe(in(d, in(e, q))) = toe(in(e, q))$

$tail(in(d, q)) = q$

$untoe(in(d, [])) = []$

$untoe(in(d, in(e, q))) = in(d, untoe(in(e, q)))$

$nonempty([]) = \mathtt{F}$

$nonempty(in(d, q)) = \mathtt{T}$

$length([]) = 0$

$length(in(d, q)) = S(length(q))$

$++([], q) = q$

$++(in(d, q), q') = in(d, ++(q, q'))$

2.7 **var** $d, e{:}D$

$q, q'{:}List$

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

$eq(in(d, q), []) = \mathtt{F}$

$eq([], in(d, q)) = \mathtt{F}$

$eq(in(d, q), in(e, q')) = eq(d, e) \wedge eq(q, q')$

2.8 If $eq(d, e) = \mathtt{T}$, then $d = if(\mathtt{T}, d, e) = if(eq(d, e), d, e) = e$.
If $d = e$, then $eq(d, e) = eq(d, d) = \mathtt{T}$.

2.9 $\mathtt{T} \vee \mathtt{T} = \mathtt{T}$ and $\mathtt{F} \vee \mathtt{T} = \mathtt{T}$.
$\neg\neg\mathtt{T} = \neg\mathtt{F} = \mathtt{T}$ and $\neg\neg\mathtt{F} = \neg\mathtt{T} = \mathtt{F}$.

2.10

(1) $\mathsf{F} \wedge \mathsf{T} = \mathsf{F}$.
$\mathsf{F} \wedge \mathsf{F} = \mathsf{F}$.

(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) $\mathtt{T} \Rightarrow y = y = \neg\neg y = \neg y \Rightarrow \mathtt{F} = \neg y \Rightarrow \neg\mathtt{T}$.
$\mathtt{F} \Rightarrow y = \mathtt{T} = \neg y \Rightarrow \mathtt{T} = \neg y \Rightarrow \neg\mathtt{F}$.

(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) $\mathtt{T} \Leftrightarrow \neg y = \neg y = \neg(\mathtt{T} \Leftrightarrow y)$.
$\mathtt{F} \Leftrightarrow \neg y = \neg\neg y = \neg(\mathtt{F} \Leftrightarrow y)$.

(8) $(x \vee \mathtt{T}) \Leftrightarrow x = \mathtt{T} \Leftrightarrow x = x = x \vee \mathtt{F} = x \vee \neg\mathtt{T}$.
$(x \vee \mathtt{F}) \Leftrightarrow x = x \Leftrightarrow x = \mathtt{T} = x \vee \mathtt{T} = x \vee \neg\mathtt{F}$.

(9) $even(plus(m, 0)) = even(m) = even(m) \Leftrightarrow \mathtt{T} = even(m) \Leftrightarrow even(0)$.
$even(plus(m, S(n))) = even(S(plus(m, n))) = \neg even(plus(m, n)) = \neg(even(m) \Leftrightarrow even(n)) = even(m) \Leftrightarrow \neg even(n) = even(m) \Leftrightarrow even(S(n))$.

(10) $even(mul(m, 0)) = even(0) = \mathtt{T} = even(m) \vee \mathtt{T} = even(m) \vee even(0)$.
$even(mul(m, S(n))) = even(plus(mul(m, n), m)) = even(mul(m, n)) \Leftrightarrow even(m) = (even(m) \vee even(n)) \Leftrightarrow even(m) = even(m) \vee \neg even(n) = even(m) \vee even(S(n))$.

2.11

(1) $mul(0,0) = 0$.
$mul(0, S(n)) = plus(mul(0,n), 0) = mul(0,n) = 0$.

(2) $plus(plus(k,\ell), 0) = plus(k,\ell) = plus(k, plus(\ell, 0))$.
$plus(plus(k,\ell), S(m)) == S(plus(plus(k,\ell), m)) = S(plus(k, plus(\ell, m))) = plus(k, S(plus(\ell, m))) = plus(k, plus(\ell, S(m)))$.

(3) $mul(k, plus(\ell, 0)) = mul(k,\ell) = plus(mul(k,\ell), 0) = plus(mul(k,\ell), mul(k,0))$.
$mul(k, plus(\ell, S(m))) = mul(k, S(plus(\ell, m))) = plus(mul(k, plus(\ell, m)), k) = plus(plus(mul(k,\ell), mul(k,m)), k) = plus(mul(k,\ell), plus(mul(k,m), k)) = plus(mul(k,\ell), mul(k, S(m)))$.

(4) $mul(mul(k,\ell), 0) = 0 = mul(k, 0) = mul(k, mul(\ell, 0))$.
$mul(mul(k,\ell), S(m)) = plus(mul(mul(k,\ell), m), mul(k,\ell)) = plus(mul(k, mul(\ell, m)), mul(k,\ell)) = mul(k, plus(mul(\ell, m), \ell)) = mul(k, mul(\ell, S(m)))$.

(5) $mul(power(m,k), power(m,0)) = mul(power(m,k), S(0)) = plus(mul(power(m,k), 0), power(m,k)) = plus(0, power(m,k)) = power(m,k) = power(m, plus(k, 0))$.
$mul(power(m,k), power(m, S(\ell))) = mul(power(m,k), mul(power(m,\ell), m)) = mul(mul(power(m,k), power(m,\ell)), m) = mul(power(m, plus(k,\ell)), m) = power(m, S(plus(k,\ell))) = power(m, plus(k, S(\ell)))$.

2.12 $+\!\!+(untoe(q), in(toe(q), q'))$.
Base case: $+\!\!+(untoe(in(d,[])), in(toe(in(d,[])), q')) = +\!\!+([], in(d,q')) = in(d,q') = in(d, +\!\!+([], q')) = +\!\!+(in(d,[]), q')$.
Inductive case:

$$\begin{array}{ll}
 & +\!\!+(untoe(in(d, in(e,q))), in(toe(in(d, in(e,q))), q')) \\
= & +\!\!+(in(d, untoe(in(e,q))), in(toe(in(e,q)), q')) \\
= & in(d, +\!\!+(untoe(in(e,q)), in(toe(in(e,q)), q'))) \\
= & in(d, +\!\!+(in(e,q), q')) \qquad\qquad \text{(by induction)} \\
= & +\!\!+(in(d, in(e,q)), q').
\end{array}$$

2.14 Suppose $[] = in(d,q)$. Then $\mathtt{F} = nonempty([]) = nonempty(in(d,q)) = \mathtt{T}$, contradicting axiom Bool1.

3.1 $a(d)\cdot(b(stop, \mathtt{F}) + c)$
$a(d)\cdot b(stop, \mathtt{F}) + a(d)\cdot c$.

3.2

(1) $((a+a)\cdot(b+b))\cdot(c+c) \overset{\text{A3}}{=} (a\cdot(b+b))\cdot(c+c) \overset{\text{A3}}{=} (a\cdot b)\cdot(c+c) \overset{\text{A3}}{=} (a\cdot b)\cdot c \overset{\text{A5}}{=} a\cdot(b\cdot c)$;

(2) $(a+a)\cdot(b\cdot c) + (a\cdot b)\cdot(c+c) \overset{\text{A3}}{=} a\cdot(b\cdot c) + (a\cdot b)\cdot(c+c) \overset{\text{A5}}{=} (a\cdot b)\cdot c + (a\cdot b)\cdot(c+c) \overset{\text{A3}}{=} (a\cdot b)\cdot(c+c) + (a\cdot b)\cdot(c+c) \overset{\text{A3}}{=} (a\cdot b)\cdot(c+c) \overset{\text{A3}}{=} (a\cdot(b+b))\cdot(c+c)$;

(3) $((a+b)\cdot c + a\cdot c)\cdot d \overset{\text{A4}}{=} ((a\cdot c + b\cdot c) + a\cdot c)\cdot d \overset{\text{A1}}{=} ((b\cdot c + a\cdot c) + a\cdot c)\cdot d \overset{\text{A2}}{=} (b\cdot c + (a\cdot c + a\cdot c))\cdot d \overset{\text{A3}}{=} (b\cdot c + a\cdot c)\cdot d \overset{\text{A4}}{=} ((b+a)\cdot c)\cdot d \overset{\text{A5}}{=} (b+a)\cdot(c\cdot d)$.

3.3 Suppose $p \subseteq q$ and $q \subseteq p$. By definition we have (1) $p+q=q$ and (2) $q+p=p$. Thus we obtain: $p \overset{(2)}{=} q+p \overset{\text{A1}}{=} p+q \overset{(1)}{=} q$.

3.5 The crux of this exercise is to show that A2′ and A3 together prove A1.
$x+y \overset{\text{A3}}{=} (x+y)+(x+y) \overset{\text{A2}'}{=} y+((x+y)+x) \overset{\text{A2}'}{=} y+(y+(x+x)) \overset{\text{A2}'}{=} ((x+x)+y)+y \overset{\text{A2}'}{=} (x+(y+x))+y \overset{\text{A2}'}{=} (y+x)+(y+x) \overset{\text{A3}}{=} y+x$.

3.7 $a \parallel (b+c) \overset{\text{CM1}}{=} (a \mathbin{⌊\!⌊} (b+c) + (b+c) \mathbin{⌊\!⌊} a) + a \mid (b+c) \overset{\text{CM9}}{=} (a \mathbin{⌊\!⌊} (b+c) + (b+c) \mathbin{⌊\!⌊} a) + (a \mid b + a \mid c) \overset{\text{CF}}{=} (a \mathbin{⌊\!⌊} (b+c) + (b+c) \mathbin{⌊\!⌊} a) + (b' + c') \overset{\text{CF}}{=} (a \mathbin{⌊\!⌊} (b+c) + (b+c) \mathbin{⌊\!⌊} a) + (b \mid a + c \mid a) \overset{\text{CM8}}{=} (a \mathbin{⌊\!⌊} (b+c) + (b+c) \mathbin{⌊\!⌊} a) + (b+c) \mid a \overset{\text{A1}}{=} ((b+c) \mathbin{⌊\!⌊} a + a \mathbin{⌊\!⌊} (b+c)) + (b+c) \mid a \overset{\text{CM1}}{=} (b+c) \parallel a$.

3.9 $(b{\cdot}a) \parallel a \stackrel{\mathrm{CM1}}{=} ((b{\cdot}a) \mathbin{\lfloor\!\lfloor} a + a \mathbin{\lfloor\!\lfloor} (b{\cdot}a)) + (b{\cdot}a) \mid a \stackrel{\mathrm{CM2,3,5}}{=} (b{\cdot}(a \parallel a) + a{\cdot}(b{\cdot}a)) + (b \mid a){\cdot}a \stackrel{\mathrm{CM1}}{=} (b{\cdot}((a \mathbin{\lfloor\!\lfloor} a + a \mathbin{\lfloor\!\lfloor} a) + a \mid a) + a{\cdot}(b{\cdot}a)) + (b \mid a){\cdot}a \stackrel{\mathrm{CM2,CF'}}{=} (b{\cdot}((a{\cdot}a + a{\cdot}a) + \delta) + a{\cdot}(b{\cdot}a)) + (b \mid a){\cdot}a \stackrel{\mathrm{A3,6}}{=} (b{\cdot}(a{\cdot}a) + a{\cdot}(b{\cdot}a)) + (b \mid a){\cdot}a \stackrel{\mathrm{A4,5}}{=} (b{\cdot}a + a{\cdot}b){\cdot}a + (b \mid a){\cdot}a \stackrel{\mathrm{CM2}}{=} (b \mathbin{\lfloor\!\lfloor} a + a \mathbin{\lfloor\!\lfloor} b){\cdot}a + (b \mid a){\cdot}a \stackrel{\mathrm{A4}}{=} ((b \mathbin{\lfloor\!\lfloor} a + a \mathbin{\lfloor\!\lfloor} b) + b \mid a){\cdot}a \stackrel{\mathrm{CM1}}{=} (b \parallel a){\cdot}a$.

3.10

$$\begin{array}{rcl}
\partial_{\{a,b\}}((a{\cdot}b) \parallel (b{\cdot}a)) & \stackrel{\mathrm{CM1}}{=} & \partial_{\{a,b\}}(((a{\cdot}b) \mathbin{\lfloor\!\lfloor} (b{\cdot}a) + (b{\cdot}a) \mathbin{\lfloor\!\lfloor} (a{\cdot}b)) + (a{\cdot}b) \mid (b{\cdot}a)) \\
& \stackrel{\mathrm{CM3,CM7}}{=} & \partial_{\{a,b\}}(a{\cdot}(b \parallel (b{\cdot}a)) + b{\cdot}(a \parallel (a{\cdot}b)) + c{\cdot}(b \parallel a)) \\
& \stackrel{\mathrm{D1\text{-}4}}{=} & \delta{\cdot}\partial_{\{a,b\}}(b \parallel (b{\cdot}a)) + \delta{\cdot}\partial_{\{a,b\}}(a \parallel (a{\cdot}b)) + c{\cdot}\partial_{\{a,b\}}(b \parallel a) \\
& \stackrel{\mathrm{A6,7}}{=} & c{\cdot}\partial_{\{a,b\}}(b \parallel a) \\
& \stackrel{\mathrm{CM1}}{=} & c{\cdot}\partial_{\{a,b\}}(b \mathbin{\lfloor\!\lfloor} a + a \mathbin{\lfloor\!\lfloor} b + b \mid a) \\
& \stackrel{\mathrm{CM2,CF}}{=} & c{\cdot}\partial_{\{a,b\}}(b{\cdot}a + a{\cdot}b + c) \\
& \stackrel{\mathrm{D1\text{-}4}}{=} & c{\cdot}(\delta{\cdot}\partial_{\{a,b\}}(a) + \delta{\cdot}\partial_{\{a,b\}}(b) + c) \\
& \stackrel{\mathrm{A6,7}}{=} & c{\cdot}c.
\end{array}$$

3.11 *send*(d), *read*(d) and *comm*(d) are abbreviated to $s(d)$, $r(d)$ and $c(d)$, respectively, for $d \in \{0,1\}$, and H denotes $\{s(0), s(1), r(0), r(1)\}$.

$$\begin{array}{cl}
& (s(0)+s(1)) \parallel (r(0)+r(1)) \\
\stackrel{\mathrm{CM1}}{=} & (s(0)+s(1)) \mathbin{\lfloor\!\lfloor} (r(0)+r(1)) + (r(0)+r(1)) \mathbin{\lfloor\!\lfloor} (s(0)+s(1)) \\
& +\ (s(0)+s(1)) \mid (r(0)+r(1)) \\
\stackrel{\mathrm{CM4,\,CM8,9}}{=} & s(0) \mathbin{\lfloor\!\lfloor} (r(0)+r(1)) + s(1) \mathbin{\lfloor\!\lfloor} (r(0)+r(1)) + r(0) \mathbin{\lfloor\!\lfloor} (s(0)+s(1)) \\
& +\ r(1) \mathbin{\lfloor\!\lfloor} (s(0)+s(1)) + s(0) \mid r(0) + s(0) \mid r(1) + s(1) \mid r(0) \\
& +\ s(1) \mid r(1) \\
\stackrel{\mathrm{CM2,CF,CF'}}{=} & s(0)(r(0)+r(1)) + s(1)(r(0)+r(1)) + r(0)(s(0)+s(1)) \\
& +\ r(1)(s(0)+s(1)) + c(0) + \delta + \delta + c(1) \\
\stackrel{\mathrm{A6}}{=} & s(0)(r(0)+r(1)) + s(1)(r(0)+r(1)) + r(0)(s(0)+s(1)) \\
& +\ r(1)(s(0)+s(1)) + c(0) + c(1).
\end{array}$$

Hence,

$$\begin{array}{cl}
& \partial_H((s(0)+s(1)) \parallel (r(0)+r(1))) \\
= & \partial_H(s(0)(r(0)+r(1)) + s(1)(r(0)+r(1)) + r(0)(s(0)+s(1)) \\
& +\ r(1)(s(0)+s(1)) + c(0) + c(1)) \\
\stackrel{\mathrm{D1\text{-}4}}{=} & \delta\partial_H(r(0)+r(1)) + \delta\partial_H(r(0)+r(1)) + \delta\partial_H(s(0)+s(1)) \\
& +\ \delta\partial_H(s(0)+s(1)) + c(0) + c(1) \\
\stackrel{\mathrm{A6,7}}{=} & c(0) + c(1).
\end{array}$$

3.12 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.

3.13 $\delta = p + q \stackrel{\mathrm{A3}}{=} (p+q) + (p+q) \stackrel{\mathrm{A1,2}}{=} p + (p + (q+q)) \stackrel{\mathrm{A3}}{=} p + (p+q) = p + \delta \stackrel{\mathrm{A6}}{=} p$.

3.14 yes; no; yes; yes; no.

3.15

$$\begin{array}{rcl}
\partial_{\{tick\}}(Clock(0)) & = & \partial_{\{tick\}}(tick{\cdot}Clock(S(0)) + display(0){\cdot}Clock(0)) \\
& = & \partial_{\{tick\}}(tick{\cdot}Clock(S(0))) + \partial_{\{tick\}}(display(0){\cdot}Clock(0)) \\
& = & \delta{\cdot}\partial_{\{tick\}}(Clock(S(0))) + display(0){\cdot}\partial_{\{tick\}}(Clock(0)) \\
& = & display(0){\cdot}\partial_{\{tick\}}(Clock(0)).
\end{array}$$

3.16 **var** n,m:Nat
q:$List$
proc $\mathit{Add\text{-}list}(q{:}List, n{:}Nat) = \mathit{print}(n) \lhd \mathit{eq}(q, []) \rhd \mathit{add}(\mathit{toe}(q)) \cdot \mathit{Add\text{-}list}(\mathit{untoe}(q), \mathit{plus}(n, \mathit{toe}(q)))$

3.17 (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 $(b = \mathtt{T} \Rightarrow x = y)$ we have $x = y$ and so $x \lhd b \rhd z = y \lhd b \rhd z$;
if $b = \mathtt{F}$, then $x \lhd b \rhd z = z = y \lhd b \rhd z$.

3.18 **var** $d{:}D$, $q{:}List$
rew $\mathit{top}(\mathit{in}(d, q)) = d$
$\mathit{tail}(\mathit{in}(d, q)) = q$

proc $\mathit{Stack}(q{:}List) = \sum_{d:D} r(d) \cdot \mathit{Stack}(\mathit{in}(d, q)) + (\delta \lhd \mathit{eq}(q, []) \rhd s(\mathit{top}(q)) \cdot \mathit{Stack}(\mathit{tail}(q)))$
init $\mathit{Stack}([])$

proc $\mathit{Queue}(q{:}List) = \sum_{d:D} r(d) \cdot \mathit{Queue}(\mathit{in}(d, q)) + (\delta \lhd \mathit{eq}(q, []) \rhd s(\mathit{toe}(q)) \cdot \mathit{Queue}(\mathit{untoe}(q)))$
init $\mathit{Queue}([])$

3.20 $0 \le S(S(0)) = \mathtt{T}$;
$S(0) \le S(S(0)) = 0 \le S(0) = \mathtt{T}$;
$S(S(0)) \le S(S(0)) = S(0) \le S(0) = 0 \le 0 = \mathtt{T}$;
$S(S(S(n))) \le S(S(0)) = S(S(n)) \le S(0) = S(n) \le 0 = \mathtt{F}$.

3.21 ($\supseteq$) By SUM3

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

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 booleans $x \lhd b \rhd y = x \subseteq x + y$. Then by SUM11, SUM4 and SUM1 $\sum_{b:Bool} x \lhd b \rhd y \subseteq x + y$.

3.23 ($\subseteq$) By SUM3

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

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 $d{:}D$. Then by SUM11, SUM4 and SUM1 $\sum_{d:D} x \lhd b(d) \rhd \delta \subseteq x$.

3.24 Let D denote $\{d_1, d_2\}$.

act $\mathit{in}, \mathit{out}{:}D$
proc $X(n{:}Nat, m{:}Nat) = \mathit{in}(d_1) \cdot X(S(n), m) + \mathit{in}(d_2) \cdot X(n, S(m))$
$+ (\mathit{out}(d_1) \cdot X(n \mathbin{\dot{-}} 1, m) \lhd n > 0 \rhd \delta) + (\mathit{out}(d_2) \cdot X(n, m \mathbin{\dot{-}} 1) \lhd m > 0 \rhd \delta)$
init $X(0, 0)$

4.3

(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 $c+c\,\mathcal{B}\,c+c$.

4.5 Base case: $a \xrightarrow{a} \surd$, while aa cannot terminate successfully by the execution of an a-transition. Hence, $a \not\underline{\leftrightarrow}\, aa$.

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, $a^{k+1} \not\underline{\leftrightarrow}\, a^{k+2}$.

5.1

(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 $\tau(\tau x+y) = \tau(\tau(x+y)+y) \overset{\text{B2}}{=} \tau(x+y) = \tau x$.

5.2
```
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
     eq(T,T)=T
     eq(F,F)=T
     eq(T,F)=F
     eq(F,T)=F

sort D
func d1,d2: -> D
map  eq: D # D -> Bool
rew  eq(d1,d1)=T
     eq(d2,d2)=T
     eq(d1,d2)=F
     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))
```

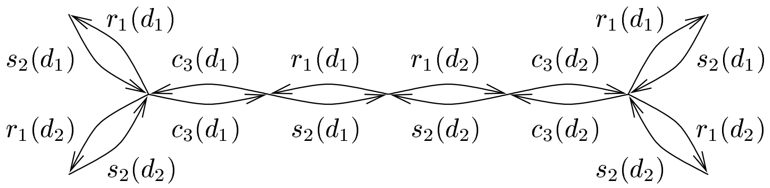
5.3
```
     Buf2 = rename({r1 -> r3, s3 -> s2},Buf1)
```

5.7 $a\,\mathcal{B}_1\,a\cdot\tau$, $\surd\,\mathcal{B}_1\,\tau$, and $\surd\,\mathcal{B}_1\,\surd$ proves $a \underline{\leftrightarrow}_b\, a\cdot\tau$;
$a\,\mathcal{B}_2\,\tau\cdot a$, $a\,\mathcal{B}_2\,a$, and $\surd\,\mathcal{B}_2\,\surd$ proves $a \underline{\leftrightarrow}_b\, \tau\cdot a$.
$a\cdot\tau\,\mathcal{B}_3\,\tau\cdot a$, $a\cdot\tau\,\mathcal{B}_3\,a$, $\tau\,\mathcal{B}_3\,\surd$, and $\surd\,\mathcal{B}_3\,\surd$ proves $a\cdot\tau \underline{\leftrightarrow}_b\, \tau\cdot a$.

5.8 $\tau\cdot(\tau\cdot(a+b)+b)+a\,\mathcal{B}\,a+b$, $\tau\cdot(a+b)+b\,\mathcal{B}\,a+b$, $a+b\,\mathcal{B}\,a+b$, and $\surd\,\mathcal{B}\,\surd$.

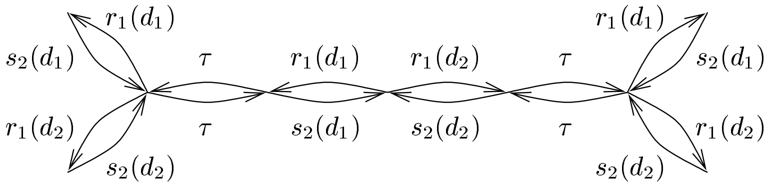
5.11 not branching bisimilar; bisimilar; branching bisimilar but not rooted branching bisimilar; rooted branching bisimilar but not bisimilar; not branching bisimilar.

5.12 The node in the middle is the root node.

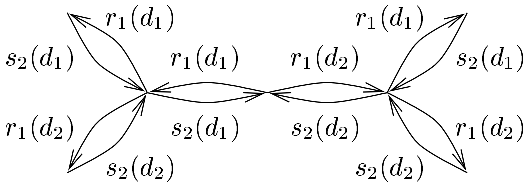


In the following execution trace, d_1 is read before d_2 via channel 1, while d_2 is sent before d_1 via channel 2: $r_1(d_1)\,c_3(d_1)\,r_1(d_2)\,s_2(d_2)\,c_3(d_1)\,s_2(d_1)$.

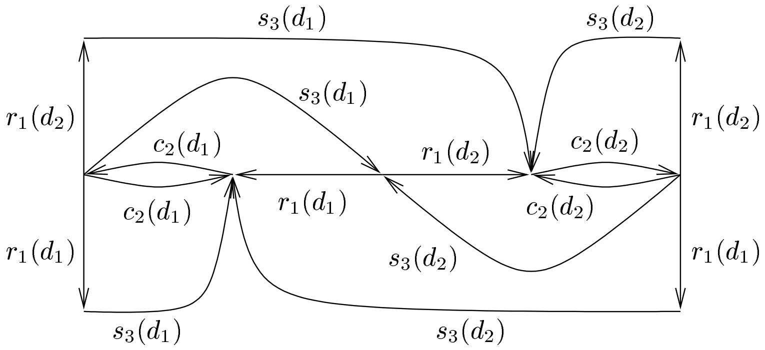
The two shortest execution traces of $\partial_{\{s_2(d_1)\}}(t)$ to a deadlock state are $r_1(d_1)\,c_3(d_1)\,r_1(d_1)$ and $r_1(d_2)\,c_3(d_2)\,r_1(d_1)$.



The four τ-transitions in the graph above are all inert, as they do not lose possible behaviours; namely, each τ-transition can be 'undone' by its reverse τ-transition.

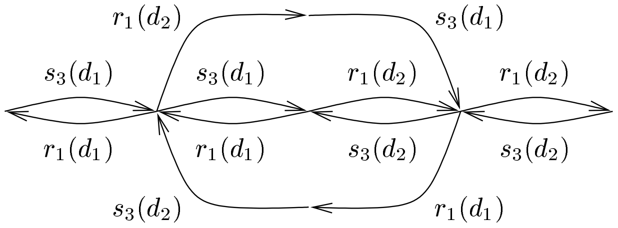


5.13 The node in the middle is the root node.

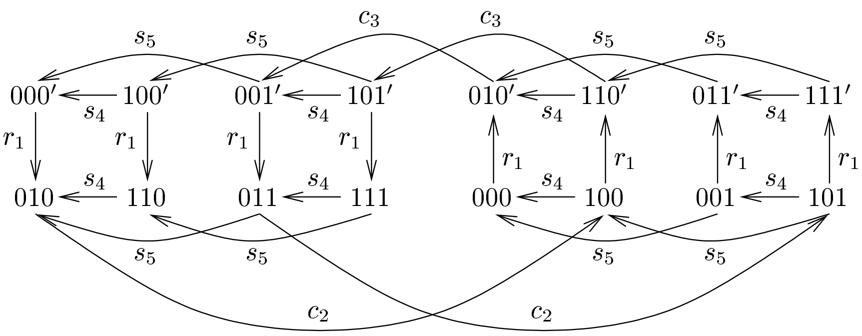


Data elements are read via channel 1 and sent via channel 3 in the same order.

The two shortest execution traces of $\partial_{\{s_3(d_1)\}}(t)$ to a deadlock state are $r_1(d_1)\,c_2(d_1)\,r_1(d_1)$ and $r_1(d_1)\,c_2(d_1)\,r_1(d_2)$.



5.14 **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 $\partial_{\{s_2(d), r_2(d), s_3(d), r_3(d) | d\in\Delta\}}(X \| Y \| Z)$



The three bits in the states above denote whether there is a datum in the buffer of X, Y or Z, respectively. In a state with or without prime the next incoming datum is sent on via channel 2 or 3, respectively. The initial state is $000'$.

6.4

$$
\begin{array}{lcl}
plusmod(k, 0) & = & k \\
plusmod(k, S(\ell)) & = & succmod(plusmod(k, \ell)) \\
ordered(k, \ell, m) & = & (k \leq \ell \wedge l < m) \vee (m < k \wedge k \leq \ell) \vee (\ell < m \wedge m < k)
\end{array}
$$

The algebraic formulation reads $ordered(k, \ell, plusmod(k, m))$.

6.6 Let $d, d', d'' \in \Delta$. An error trace is:

(0) $r_{\mathrm{A}}(d)\, c_{\mathrm{B}}(d, 0)\, j\, c_{\mathrm{C}}(d, 0)\, s_{\mathrm{D}}(d)$ **(1)** $r_{\mathrm{A}}(d')\, c_{\mathrm{B}}(d', S(0))\, j\, c_{\mathrm{C}}(d', S(0))\, s_{\mathrm{D}}(d')$ **(2)** $r_{\mathrm{A}}(d)\, c_{\mathrm{B}}(d, 0)\, j$ $c_{\mathrm{C}}(d, 0)$ **(3)** $c_{\mathrm{E}}(S(S(0)))\, j\, c_{\mathrm{F}}(S(S(0)))$ **(4)** $r_{\mathrm{A}}(d'')\, c_{\mathrm{B}}(d'', S(S(0)))\, j\, c_{\mathrm{C}}(d'', S(S(0)))\, s_{\mathrm{D}}(d'')$ **(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 $[0, S(0)]$;

6.7
```
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),add(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)
```

6.8
```
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),add(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
```

6.11 Each τ-transition loses the possibility to execute one of the $leader(n)$-actions at the end.

6.12 The network is captured by

$$\tau_I(\partial_H(X(n_0,\{n_1,n_2\},0) \parallel X(n_1,\{n_0,n_2\},0) \parallel X(n_2,\{n_0,n_1\},0)))$$

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.

7.2 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) $Z=(a+b)\cdot Z$.

7.3 Assume a state space. For P a set of states, define $s_0 \in split_{\surd}(P)$ if there exists a sequence $s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{\tau} \surd$ for some $n \geq 0$ such that $s_i \in P$ for $i=0,\ldots,n$.
P can be split with respect to $\surd$ if $\emptyset \subset split_{\surd}(P) \subset P$.

7.9 all τ-transitions;
$\emptyset$;
all τ-transitions;
$\emptyset$.

7.10 $\{s_0 \xrightarrow{\tau} s_1\}$;
$\emptyset$;
$\{s_0 \xrightarrow{\tau} s_1\}$.

7.11 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 be composed as a conjunction of an appropriate subset of the six aforementioned ACTL formulas and their negations.

7.12 s_6 and s_8; s_6, s_8 and s_9; s_8; s_4; s_3, s_4, s_5, s_7, s_8, s_{10} and s_{11}.

7.13

(1) $\mathtt{E\,T\,U}\,(\neg\,(\langle a\rangle\mathtt{T} \vee \langle b\rangle\mathtt{T}))$;

(2) $\mathtt{EG}\,(\langle a\rangle\mathtt{T} \vee \langle b\rangle\mathtt{T})$.

7.19 $\emptyset$ is the solution for both X and Y.

8.7 The process declaration is captured by $Z(\mathtt{T})$, where

$$Z(b{:}Bool) = a{\cdot}Z(\neg b) \lhd b \rhd \delta + c{\cdot}Z(\neg b) \lhd \neg b \rhd \delta.$$

8.8

$$\begin{aligned}
&Y(n_1{:}Nat,\ldots,n_k{:}Nat) \;= \\
&\textstyle\sum_{i=1}^{k} a(f(n_i)){\cdot}Y(n_i := g(n_i)) \lhd h(n_i) \rhd \delta \\
&+\; \textstyle\sum_{j=1}^{k} b(f'(n_j)){\cdot}Y(n_j := g'(n_j)) \lhd h'(n_j) \rhd \delta \\
&+\; \textstyle\sum_{i,j=1}^{k} c(f(n_i)){\cdot}Y(n_i := f(n_i), n_j := f'(n_j)) \lhd h(n_i) \wedge h'(n_j) \wedge f(n_i) = f'(n_j) \wedge i \neq j \rhd \delta
\end{aligned}$$

8.9

$$\begin{array}{ll}
& X(n_1) \parallel X(n_2) \\
= & X(n_1) \mathbin{\|\!_} X(n_2) + X(n_2) \mathbin{\|\!_} X(n_1) + X(n_1) \,|\, X(n_2) \\
= & (a(n_1){\cdot}X(n_1+1) \lhd n_1 < 10 \rhd \delta + b(n_1){\cdot}X(n_1+2) \lhd n_1 > 5 \rhd \delta) \mathbin{\|\!_} X(n_2) \\
+ & (a(n_2){\cdot}X(n_2+1) \lhd n_2 < 10 \rhd \delta + b(n_2){\cdot}X(n_2+2) \lhd n_2 > 5 \rhd \delta) \mathbin{\|\!_} X(n_1) \\
+ & (a(n_1){\cdot}X(n_1+1) \lhd n_1 < 10 \rhd \delta + b(n_1){\cdot}X(n_1+2) \lhd n_1 > 5 \rhd \delta) \,| \\
& (a(n_2){\cdot}X(n_2+1) \lhd n_2 < 10 \rhd \delta + b(n_2){\cdot}X(n_2+2) \lhd n_2 > 5 \rhd \delta) \\
= & a(n_1){\cdot}(X(n_1+1) \parallel X(n_2)) \lhd n_1 < 10 \rhd \delta \;+\; b(n_1){\cdot}(X(n_1+2) \parallel X(n_2)) \lhd n_1 > 5 \rhd \delta \\
+ & a(n_2){\cdot}(X(n_1) \parallel X(n_2+1)) \lhd n_2 < 10 \rhd \delta \;+\; b(n_2){\cdot}(X(n_1) \parallel X(n_2+2)) \lhd n_2 > 5 \rhd \delta \\
+ & c(n_1){\cdot}(X(n_1) \parallel X(n_2)) \lhd n_1 < 10 \wedge n_2 > 5 \wedge n_1 = n_2 \rhd \delta \\
+ & c(n_1){\cdot}(X(n_1) \parallel X(n_2)) \lhd n_1 > 5 \wedge n_1 > 10 \wedge n_1 = n_2 \rhd \delta
\end{array}$$

8.12 We need to prove $\mathcal{I}_i(n) \Rightarrow \mathcal{I}_i(n+2)$ for $i = 1, 2$.
If n is even, then $\mathcal{I}_1(n) = \mathcal{I}_1(n+2) = \mathtt{T}$ and $\mathcal{I}_2(n) = \mathcal{I}_2(n+2) = \mathtt{F}$.
If n is odd, then $\mathcal{I}_1(n) = \mathcal{I}_1(n+2) = \mathtt{F}$ and $\mathcal{I}_2(n) = \mathcal{I}_2(n+2) = \mathtt{T}$.

8.13 We prove that $\mathcal{I}_i$ is an invariant for the LPE X in Definition 8.1; 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.

8.14 $\mathcal{I}(n) = \mathtt{T}$ 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.

9.4 The focus condition for $n{:}Nat$ is: $\exists m{:}Nat(n = 3m \vee n = 3m+1)$.

$$\phi(n) = \begin{cases} \mathtt{T} & \text{if } \exists m{:}Nat(n = 3m \vee n = 3m+2) \\ \mathtt{F} & \text{if } \exists m{:}Nat(n = 3m+1) \end{cases}$$

(Actually, it does not matter whether one defines $\phi(3m+2)$ to be T or F.)

The matching criteria are fulfilled:

- $n = 3m+2 \Rightarrow \phi(n) = \phi(n+1) = \mathtt{T}$;
- $n = 3m \Rightarrow h'_a(\phi(n)) = \phi(n) = \mathtt{T}$;
  $n = 3m+1 \Rightarrow h'_d(\phi(n)) = \neg\phi(n) = \neg\mathtt{F} = \mathtt{T}$;
- $((n = 3m \vee n = 3m+1) \wedge \phi(n)) \Rightarrow n = 3m$;
  $((n = 3m \vee n = 3m+1) \wedge \neg\phi(n)) \Rightarrow n = 3m+1$;
- actions do not carry data parameters;
- $n = 3m \Rightarrow \phi(n+1) = \mathtt{F}$;
  $n = 3m+1 \Rightarrow \phi(n+1) = \mathtt{T}$.

9.5 The focus condition for $n{:}Nat$ is: $\exists m{:}Nat(n = 3m \vee n = 3m+1)$.

$$\mathcal{I}(n) = \begin{cases} \mathtt{T} & \text{if } \exists m{:}Nat(n = 3m \vee n = 3m+2) \\ \mathtt{F} & \text{if } \exists m{:}Nat(n = 3m+1) \end{cases}$$

The matching criteria are fulfilled for all n with $\mathcal{I}(n) = \mathtt{T}$:

- $n = 3m+2 \Rightarrow \phi(n) = \phi(n+1) = nil$;
- $n = 3m \Rightarrow h'_a(\phi(n)) = \mathtt{T}$;
- $n = 3m \Rightarrow n = 3m$;
- actions do not carry data parameters;
- $n = 3m \Rightarrow \phi(n+2) = nil$.

9.8 Parameters y and z are found to be constant.

$$X(w{:}Nat, x{:}Nat) \;=\; a(x)\cdot X(x,w) \lhd eq(0,0) \rhd \delta \;+\; b(0)\cdot X(S(0),x,)$$

9.9 Parameter x is found to be inert.

$$X(y{:}D, z{:}D) \;=\; a\cdot X(z,y) \;+\; \sum_{w:D} b(z)\cdot X(y,z)$$

10.2 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 $j \notin p[i]$.

Bibliography

[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., *Proceedings 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] Th. Arts and I.A. van Langevelde. How μCRL supported a smart redesign of a real-world protocol. In S. Gnesi and D. Latella, eds, *Proceedings 4th ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'99)*. Trento, Italy, pp. 31–53, 1999.

[3] Th. Arts and I.A. van Langevelde. Correct performance of transaction capabilities. In *Proceedings 2nd Conference on Application of Concurrency to System Design (ICACSD'01)*, Newcastle upon Tyne, UK, pp. 35-42. IEEE Computer Society, 2001.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[5] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[6] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, 1982.

[7] H.P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. North-Holland, 1984. Second edition.

[8] 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.

[9] T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.

[10] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. ACM/Addison Wesley, 1989.

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

[12] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, ed., *Proceedings 11th Colloquium on Automata, Languages and Programming (ICALP'84)*, Antwerp, Belgium, LNCS 172, pp. 82–95. Springer-Verlag, 1984.

[13] 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.

[14] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, eds, *Proceedings 5th Conference on Concurrency Theory (CONCUR'94)*, Uppsala, Sweden, LNCS 836, pp. 401–416. Springer-Verlag, 1994.

[15] M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[16] M.A. Bezem and A. Ponse. Two finite specifications of a queue. *Theoretical Computer Science*, 177(2):487–507, 1997.

[17] S.C.C. Blom. Partial τ-confluence for efficient state space generation. Technical Report SEN-R0123, CWI, 2001.

[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, *Proceedings 13th Conference on Computer Aided Verification (CAV'01)*, Paris, France, LNCS 2102, pp. 250–254. Springer-Verlag, 2001.

[19] S.C.C. Blom and J.C. van de Pol. State space reduction by proving confluence. In *Proceedings 14th Conference on Computer Aided Verification (CAV'02)*, Copenhagen, Denmark, LNCS. Springer-Verlag, 2002. To appear.

[20] G. von Bochmann. Logical verification and implementation of protocols. In *Proceedings 4th ACM-IEEE Data Communications Symposium*, Quebec, Canada, pp. 7-15–7-20. IEEE, 1975.

[21] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[22] 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.

[23] J.J. Brunekreef. Sliding window protocols. In S. Mauw and G.J. Veltink, eds, *Algebraic Specification of Communication Protocols*. Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.

[24] G. Bruns. *Distributed Systems Analysis with CCS*. Prentice Hall, 1997.

[25] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[26] R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[27] 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.

[28] V.G. Cerf and R.E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, COM-22:637–648, 1974.

[29] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, ed., *Proceedings 3rd Workshop on Logics of Programs*, Yorktown Heights, New York, LNCS 131, pp. 52–71. Springer, 1982.

[30] 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.

[31] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In C. Courcoubetis, ed., *Proceedings 5th Conference on Computer Aided Verification (CAV'93)*, Elounda, Greece, LNCS 697, pp. 450–462. Springer-Verlag, 1993.

[32] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.

[33] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[34] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Conference Record 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, Los Angeles, California, pp. 238–252. ACM, 1977.

[35] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[36] 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., *Proceedings 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.

[37] H. Davenport. *The Higher Arithmetic*. Cambridge University Press, 1952.

[38] P.F.G. Dechering and I.A. van Langevelde. The verification of coordination. In A. Porto and C. Roman, eds, *Proceedings 4th Conference on Coordination Models and Languages (COORDINATION'00)*, Limassol, Cyprus, LNCS 1906, pp. 335–340. Springer-Verlag, 2000.

[39] R. De Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, ed., *Proceedings Spring School on Semantics of Systems of Concurrent Processes*, La Roche Posay, France, LNCS 469, pp. 407–419. Springer, 1990.

[40] 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.

[41] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings 1st IEEE Symposium on Logic in Computer Science (LICS'86)*, Cambridge, Massachusetts, pp. 267–278. IEEE Computer Society Press, 1986.

[42] 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.

[43] W.J. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2000.

[44] 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, *Proceedings 8th Conference on Computer-Aided Verification (CAV'96)*, New Brunswick, New Jersey, LNCS 1102, pp. 437–440. Springer-Verlag, 1996.

[45] L.-å. Fredlund, J.F. Groote, and H.P. Korver. Formal verification of a leader election protocol in process algebra. *Theoretical Computer Science*, 177(2):459–486, 1997.

[46] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In B. Steffen, ed., *Proceedings 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.

[47] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[48] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[49] J.F. Groote. *Process Algebra and Structured Operational Semantics.* PhD thesis, University of Amsterdam, 1991.

[50] J.F. Groote and H.P. Korver. Correctness proof of the bakery protocol in 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. 63–86. Springer-Verlag, 1995.

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

[52] J.F. Groote, F. Monin, and J.C. van de Pol. Checking verifications of protocols and distributed systems by computer. In D. Sangiorgi and R. de Simone, eds, *Proceedings 9th Conference on Concurrency Theory (CONCUR'98)*, Sophia Antipolis, France, LNCS 1466, pp. 629–655. Springer, 1998.

[53] J.F. Groote, F. Monin, and J. Springintveld. A computer checked algebraic verification of a distributed summing protocol. Technical Report 97/14, Department of Mathematics and Computer Science, Eindhoven University of Technology, 1997.

[54] J.F. Groote, J. Pang, and A.G. Wouters. A balancing act: Analyzing a distributed lift system. *Proceeding 6th Workshop on Formal Methods for Industrial Critical Systems (FMICS'01)*, Paris, France, pp. 1–12, 2001.

[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, *Proceedings 5th Conference on Algebraic Methodology and Software Technology (AMAST'96)*, Munich, Germany, LNCS 1101, pp. 536–550. Springer, 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, *Proceedings 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, *Proceedings 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.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1/2):47–81, 1996.

[62] 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.

[63] J.F. Groote and F.W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In M.S. Paterson, ed., *Proceedings 17th Colloquium on Automata, Languages and Programming (ICALP'90)*, Warwick, UK, LNCS 443, pp. 626–638. Springer-Verlag, 1990.

[64] J.F. Groote and J.J. van Wamel. The parallel composition of uniform processes with data. *Theoretical Computer Science*, 266:631–652, 2001.

[65] 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.

[66] M.C.B. Hennessy and H. Lin. Unique fixpoint induction for message-passing process calculi. In *Proceedings 3rd Australasian Theory Symposium on Computing (CATS'97)*, Sydney, Australia, pp. 122–131. Australia Computer Science Communications, 1997.

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

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

[69] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[70] G.J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305. Kluwer, 1998

[71] IEEE Computer Society. *IEEE Standard for a High Performance Serial Bus*. Std. 1394-1995, August 1996.

[72] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, 1990.

[73] H.P. Korver. *Protocol Verification in μCRL*. PhD thesis, University of Amsterdam, 1994.

[74] H.P. Korver and M.P.A. Sellink. Example verifications using alphabet axioms. *Formal Aspects of Computing*, 10(1):43–58, 1998.

[75] H.P. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. In M. Hagiya and J.C. Mitchell, eds, *Proceedings 2nd Symposium on Theoretical Aspects of Computer Software (TACS'94)*, Sendai, Japan, LNCS 789, pp. 161–178. Springer-Verlag, 1994.

[76] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[77] I.A. van Langevelde. A compact file format for labeled transition systems. Technical Report SEN-R0102, CWI, 2001.

[78] R.S. Lazic. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University Computing Laboratory, 1999.

[79] 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.

[80] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley/Teubner, 1996.

[81] S.P. Luttik. Description and formal specification of the link layer of P1394. In I. Lovrek, ed., *Proceedings 2nd Workshop on Applied Formal Methods in System Design*, Zagreb, Croatia, 1997.

[82] W.C. Lynch. Reliable full duplex file transmission over half-duplex telephone lines. *Communications of the ACM*, 11(6):407–410, 1968.

[83] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer, 1993.

[84] R. Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer-Verlag, 1980.

[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] S. Owre, J.M. Rushby, and N. Shankar. PVS: a Prototype Verification System. In D. Kapur, ed., *Proceedings 11th Conference on Automated Deduction (CADE'92)*, Saratoga Springs, New York, LNCS 607, pp. 748–752. Springer-Verlag, 1992.

[88] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[89] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, ed., *Proceedings 5th GI (Gesellschaft für Informatik) Conference*, Karlsruhe, Germany, LNCS 104, pp. 167–183. Springer-Verlag, 1981.

[90] L.C. Paulson. Isabelle: the next seven hundred theorem provers. In E. Lusk and R. Overbeek, eds, *Proceedings 9th Conference on Automated Deduction (CADE'88)*, Argonne, LNCS 310, pp. 772–773. Springer-Verlag, 1988.

[91] A. Pnueli. The temporal logic of programs. In *Proceedings 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, Providence, Rhode Island, pp. 46–57. IEEE Computer Society Press, 1977.

[92] J.C. van de Pol. A prover for the μCRL toolset with applications – version 1. Technical Report SEN-R0106, CWI, 2001.

[93] J. Rathke. Unique fixpoint induction for value-passing processes. In *Proceedings 12th IEEE Symposium on Logic in Computer Science (LICS'97)*, Warsaw, Poland, pp. 140–148. IEEE Computer Society Press, 1997.

[94] J.M.T. Romijn. A timed verification of the IEEE 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, 2001.

[95] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[96] A.A. Schoone. *Protocols by Invariants*. Cambridge International Series on Parallel Computing 7. Cambridge University Press, 1996.

[97] 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.

[98] 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.

[99] M.I.A. Stoelinga and F.W. Vaandrager. Root contention in IEEE 1394. In J.-P. Katoen, ed., *Proceedings 5th AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, Bamberg, Germany, LNCS 1601, pp. 53–74. Springer-Verlag, 1999.

[100] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.

[101] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[102] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[103] Y.S. Usenko. State space generation for the HAVi leader election protocol. *Science of Computer Programming*, 43(1):1–33, 2002.

[104] J.J. van Wamel. *Verification Techniques for Elementary Data Types and Retransmission Protocols*. PhD thesis, University of Amsterdam, 1995.

[105] A.G. Wouters. Manual for the μCRL toolset (version 2.8.2). Technical Report SEN-R0130, CWI, 2001.

[106] M. Ying. Weak confluence and τ-inertness. *Theoretical Computer Science*, 238(1/2):465–475, 2000.