Modelling Concurrent Systems: Protocol Verification in μCRL

Wan Fokkink & Jan Friso Groote
Department of Software Engineering, CWI, Amsterdam
Email: wan@cwi.nl, jfg@cwi.nl

Michel Reniers
Technical Applications, Computing Science Department
Eindhoven University of Technology, Eindhoven
Email: michelr@win.tue.nl

August 2000

Contents

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 [18, 57, 59], CSP [47, 48, 66] and ACP [9, 3, 29] 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 [2] 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[41] has been developed,[1] denoted μCRL (or mCRL, if Greek letters are

[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.

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 parameterised 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 [40], 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 [12, 43] 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 extreme precision in proofs. Therefore, an important research area is to use theorem provers such as PVS [60], HOL [33], Isabelle [63] and Nqthm [15] to help in finding and checking derivations in μCRL. A large number of distributed systems have been verified in μCRL [13, 14, 31, 35, 51, 52, 67, 72], often with the help of a proof checker or theorem prover [11, 38, 53]. See [37] 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 process graph, an edge $t \overset{a(d_1,\dots,d_n)}{\to} t'$ means that process term t can perform action a, parametrised with data elements $d_1, \dots, d_n$, to evolve into process term t'. If the process graph belonging to a μCRL specification is finite, then the μCRL tool set in combination with the CADP tool set [30] can generate and visualise this graph. While the process algebraic proofs that were discussed earlier can cope with unspecified data types, the generation of a process graph 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.

A severe complication in the generation of process graphs 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 process graph becomes problematic, to say the least. The following methods have been developed to help tackle large process graphs. On-the-fly analysis of process graphs allows one to generate only part of a process graph. Scenario-based verification [25] takes as starting point a certain scenario of inputs from the outside world, to restrict the behavioural possibilities of a distributed system. Often a μCRL specification can be manipulated in such a way that the resulting process graph becomes significantly smaller; see [73]. The ATerm library [16] allows one to store process graphs 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 [36] targets optimal representation of process graphs using standard compression techniques, typically reducing a process graph to about 2% of its original size. Model checking [21] provides a framework to efficiently prove interesting properties of large process graphs. Finally, algorithms have been developed to reduce process graphs after abstracting away from internal computation steps.

This text is set up as follows. Section 2 gives an introduction into the algebraic specification of abstract data types. Section 3 provides an overview of process algebra, and explains the basics of the specification language μCRL. In Section 4 it is explained how one can abstract away from the internal computation steps of a process. Section 5 contains a number of μCRL specifications of protocols from the literature, together with extensive explanations to guide the reader through these specifications. In Section 6 a number of standard process algebraic techniques are described that can be used in the verification of μCRL specifications. In Section 7 the techniques of the previous section are applied the verify the tree identify protocol. Section 8 describes some algorithms to eliminate internal computation steps from a process graph, and explains the basics of model checking. The reader is referred to the manual of μCRL [73] for more detailed information on μCRL and its tool set.

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 [8, 55] 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*:

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

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 1.

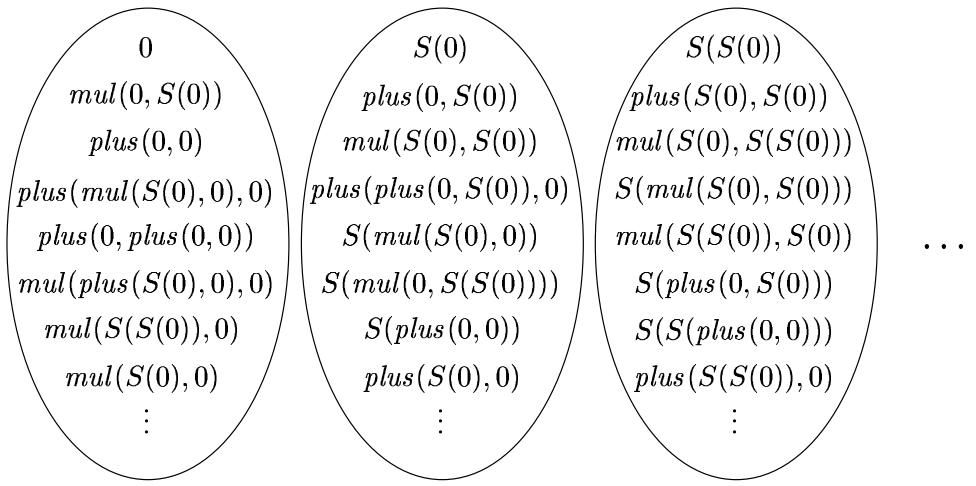


Figure 1: Initial model for the natural numbers

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.

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

and

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

In general, an *algebraic specification* consists of:

1. a signature consisting of function symbols, from which one can build terms;

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

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 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 $\quad Bool$

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 $\quad Bool$
func $\quad \mathsf{t}, \mathsf{f} :\to Bool$

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 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 $\mathsf{t} = \mathsf{f}$, we say that the specification is inconsistent and it looses any meaning.

Bool1	$\neg(\mathsf{t} = \mathsf{f})$
Bool2	$\neg(b = \mathsf{t}) \Rightarrow b = \mathsf{f}$

Table 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 $\mathsf{t}, \mathsf{f} :\rightarrow Bool$
$0 :\rightarrow Nat$
$S : Nat \rightarrow Nat$

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 D, it is required that this data type is not empty. For example, the following declaration is invalid.

sort D
func $f : D \rightarrow D$

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 [41]).

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. 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 $\wedge : Bool \times Bool \rightarrow Bool$
$plus : Nat \times Nat \rightarrow Nat$

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 $x \wedge \mathsf{t} = x$
$x \wedge \mathsf{f} = \mathsf{f}$
$plus(n, 0) = n$
$plus(n, S(m)) = S(plus(n, m))$

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 2.1 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 2.2 Give algebraic specifications of 'greater than or equal' $\geq$, 'smaller than' $<$ and 'greater than' $>$ on the natural numbers.

Exercise 2.3 Give algebraic specifications of $even : Nat \to Bool$ and $power : Nat \times Nat \to Nat$, such that $even(n) = \mathsf{t}$ if and only if n is an even number, and $power(m, n)$ equals m^n.

Exercise 2.4 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 into a list. Extend this with non-constructor functions $toe : List \to D$ to obtain the element at the end of a list, $untoe : List \to List$ to remove this last element from a list, $isempty : List \to Bool$ to check whether a list is empty and $++ : List \times List \to List$ to concatenate two lists.

Note that in Exercise 2.4 the function *toe* is not well-defined on the empty list, meaning that $toe([])$ is not equal to an element of D. Likewise, as the empty list does not contain a last element, preferably one leaves $untoe([])$ 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) last element of an empty list. However, if some process verification does use in an essential way that one is allowed to remove the last element of the empty list, one could consider adding the equation $untoe([]) = []$ 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

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. 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 [2]. We give an example.

Example 2.2 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.

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

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 this case, currently the tool set of μCRL selects the first rewrite rule in 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 this case μCRL uses *innermost* rewriting, meaning that it selects a subterm 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. That is, $eq(d,e) = \mathsf{t}$ if $d = e$ and $eq(d,e) = \mathsf{f}$ if $d \neq e$, for all $d, e{:}D$. For example, for the booleans one could define an equality function as follows.

rew $eq(\mathsf{t},\mathsf{t}) = \mathsf{t}$

$$\begin{aligned} &eq(\mathsf{f},\mathsf{f}) = \mathsf{t}\\ &eq(\mathsf{t},\mathsf{f}) = \mathsf{f}\\ &eq(\mathsf{f},\mathsf{t}) = \mathsf{f}\end{aligned}$$

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

var $n, m{:}Nat$
rew $eq(0,0) = \mathsf{t}$
$eq(S(n),0) = \mathsf{f}$
$eq(0,S(n)) = \mathsf{f}$
$eq(S(n),S(m)) = eq(n,m)$

Exercise 2.5 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 2.6 Jan Bergstra observed the following remarkable way to specify for any data type D an *eq* function to represent equality. We need an auxiliary if-then-else function called *if*.

map $eq : D \times D \to Bool$
$if : Bool \times D \times D \to D$
var $d, e{:}D$
rew $eq(d,d) = \mathsf{t}$
$if(\mathsf{t},d,e) = d$
$if(eq(d,e),d,e) = e$

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

$$eq(d,e) = \mathsf{t} \quad \Leftrightarrow \quad d = e.$$

We note that the specification of the *eq* function in Exercise 2.6 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 2.6 $eq(S(0),0)$ 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 $b = \mathsf{t}$ and $b = \mathsf{f}$. In other words, we must show that $\mathsf{t} \wedge \mathsf{t} = \mathsf{t}$ and $\mathsf{f} \wedge \mathsf{f} = \mathsf{f}$. These are trivial instances of the defining rewrite rules for $\wedge$ mentioned before.

Exercise 2.7 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 2.8 Prove by induction that:

(1) $\mathsf{f} \wedge x = \mathsf{f}$;

(2) $x \Rightarrow \mathsf{t} = \mathsf{t}$;

(3) $x \Rightarrow \mathsf{f} = \neg x$;

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

(5) $x \Leftrightarrow \mathsf{t} = x$;

(6) $x \Leftrightarrow x = \mathsf{t}$;

(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 2.9 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 2.10 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 2.4 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 $\mathsf{t}=\mathsf{f}$, contradicting axiom Bool1. We give an example showing that the natural numbers zero and one are not equal.

Example 2.3 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 $0\leq n=\mathsf{t}$
$S(n)\leq 0=\mathsf{f}$
$S(n)\leq S(m)=n\leq m$

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

Exercise 2.11 Prove, using Exercise 2.4, that the empty list and a nonempty list must always be different.

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 [41].

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.

All operators in the language are provided with a characterising set of axioms, following the process algebraic tradition. Basically, they assist in interpreting the language correctly. Usually the semantics of the language is given in terms of a data algebra and an operational semantics, but 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 are abstract representations of 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 one or more data terms within brackets. Intuitively, an action $a(d_1, \ldots, d_n)$ can execute itself, after which it terminates successfully:

$$\begin{tikzcd} a(d_1, \ldots, d_n) \arrow[d, "a(d_1, \ldots, d_n)"] \\ \surd \end{tikzcd}$$

The symbol $\surd$ 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.

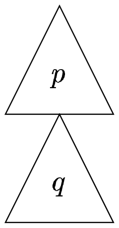
act *time-out*
$a{:}Bool$
$b{:}Nat \times D$

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.

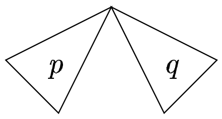
3.2 Alternative and sequential composition

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 process graph of $p{\cdot}q$ is obtained by replacing each successful termination $r \xrightarrow{a} \surd$ in the process graph 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 process graph of $p+q$ is obtained by joining p and q at their roots:



Exercise 3.1 Specify the process that first executes $a(d)$, and then $b(stop, \mathsf{f})$ or c. Also specify the process that executes either $a(d)$ followed by $b(stop, \mathsf{f})$, 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 2: Basic axioms for μCRL

In Table 2 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 $\mathcal{E}_{\mathrm{BPA}}$:

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

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

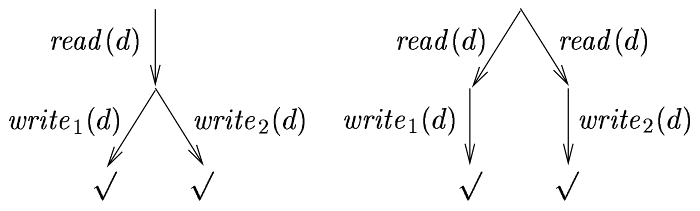
(3) $((a+b)\cdot c + a\cdot c)\cdot d = (b+a)\cdot(c\cdot d)$.

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 For all μCRL processes p and q we have: 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 3.1 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 3.4 Prove that the axioms A1-3 are equivalent to axiom A3 together with

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

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 $a\,|\,b = c$

This means that if actions $a(d_1, \ldots, d_n)$ and $b(d_1, \ldots, d_n)$ can happen in parallel, they may synchronise and this communication is denoted by $c(d_1, \ldots, d_n)$. 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:

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

If a communication between actions is declared as above, then communication is another possibility for parallel processes. For example, the process $a \parallel b$ now behaves as $a{\cdot}b + b{\cdot}a + c$.

Example 3.2 Let the communication of two actions from $\{a, b, c\}$ always result to c. The process graph of the process term $(a{\cdot}b) \parallel (b{\cdot}a)$ is depicted in Figure 2.

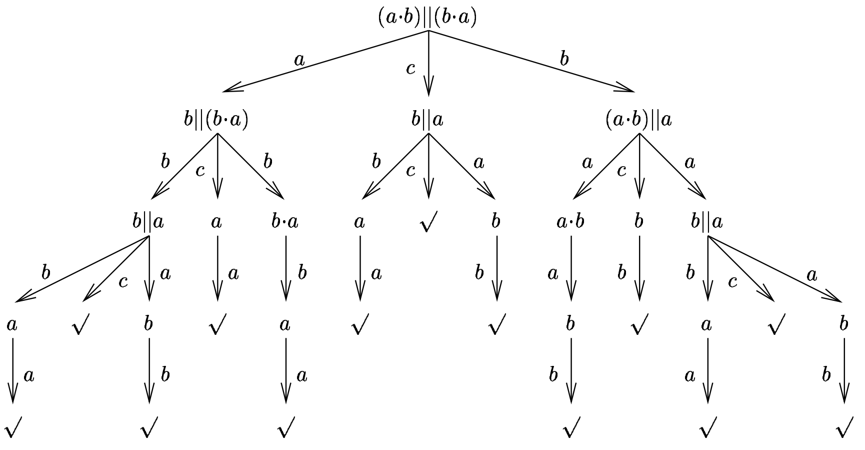


Figure 2: Process graph of $(a{\cdot}b) \parallel (b{\cdot}a)$

Example 3.2 shows that the parallel composition of two simple processes produces a relatively large process graph. 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 that describe the parallel operator are in Table 3. In order to formulate the axioms two auxiliary operators have been introduced. The *left merge* $\mathbin{\|\!\lfloor}$ 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 $\parallel$, $\mathbin{\|\!\lfloor}$ and $|$ bind stronger than the $+$, and weaker than $\cdot$. For example, $a{\cdot}b \mathbin{\|\!\lfloor} c + a \parallel b{\cdot}c$ represents $((a{\cdot}b) \mathbin{\|\!\lfloor} c) + (a \parallel (b{\cdot}c))$.

The core law for the parallel operator is CM1 in Table 3. It says that $x \parallel y$ either x performs the first step, represented by the summand $x \mathbin{\|\!\lfloor} y$, or y can do the first step, represented by

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	$a(\vec{d}) \mid b(\vec{d}) = c(\vec{d})$
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: Axioms for parallellism in μCRL

$y \mathbin{\|\!\lfloor} x$, or the first step of $x \parallel y$ is a communication between x and y, represented by $x \mid y$. All other axioms in Table 3 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.

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 $\vec{d}$ does not correspond with $\vec{e}$.

Exercise 3.5 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 communication in such cases, we introduce a special constant δ called *deadlock*, which does not display any behaviour. 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.

Note that the deadlock does not carry any data parameters.

Sometimes we want to express that certain actions cannot happen, and must be blocked, i.e., renamed to δ. Generally, this is only done when we want to force this action into a communication. The unary *encapsulation* operator ∂_H ($H \subseteq \mathsf{Act}$) is specially designed for this task. The process $\partial_H(p)$ can execute all actions of p that are not in H. Typically, $\partial_{\{b\}}(a{\cdot}b(3){\cdot}c) = a{\cdot}\delta$.

Example 3.3 Suppose a datum 0 or 1 is sent into a channel, which is expressed by the process term $\mathit{send}(0) + \mathit{send}(1)$. Let this datum be received at the other side of the channel, which is expressed by the process term $\mathit{read}(0) + \mathit{read}(1)$. The communication of $\mathit{send}(d)$ and $\mathit{read}(d)$ results to $\mathit{comm}(d)$ 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

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

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

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	$\delta \mathbin{\|\!\lfloor} x = \delta$	
CD2	$\delta \mid x = \delta$	
CD3	$x \mid \delta = \delta$	

Table 4: Axioms for deadlock

Table 4 lists the axioms for deadlock and encapsulation.

Exercise 3.6 Suppose action a cannot communicate with itself. Derive the equation $(b{\cdot}a) \parallel a = (b \parallel a){\cdot}a$.

Exercise 3.7 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 3.2).

Exercise 3.8 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 3.3 and $comm(0) + comm(1)$.

Exercise 3.9 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 3.10 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 $t \xrightarrow{a} \delta$ with a transition of the form $t \xrightarrow{a} \surd$; 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 are transitions $p \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} p_n$ 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 [70] contains a deadlock; see [34, *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 3.11 Let the communication of b and c result to a, while a and c do not communicate. Say for each of the following processes 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 form

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, process $X(x_1, \ldots, x_n)$ 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 in some sense that occurrences of expressions $Y(d_1, \ldots, d_m)$ are to be proceeded by actions. For example, an equation $Z = Z + a$ is considered meaningless, as it does not determine the entire initial behaviour of the process name Z.

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 which represents the natural numbers.

act	tick
	$\mathit{display}{:}\mathit{Nat}$
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 3.12 Derive $\partial_{\{\mathit{tick}\}}(\mathit{Clock}(0)) = \mathit{display}(0)\cdot\partial_{\{\mathit{tick}\}}(\mathit{Clock}(0))$.

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 t (true) and behaves as q if b is equal to 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 5. There are no further axioms needed, because all essential properties of conditionals are provable using axioms Bool1 and Bool2.

C1	$x \lhd \mathsf{t} \rhd y = x$
C2	$x \lhd \mathsf{f} \rhd y = y$

Table 5: Axioms for conditionals

Exercise 3.13 Specify a process that adds the elements of a list of natural numbers and prints the final result.

Exercise 3.14 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 $(b = \mathsf{t} \Rightarrow x = y)$ 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 $Buffer = \sum_{n:Nat} r(n){\cdot}s(n){\cdot}Buffer$

Exercise 3.15 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.

In Table 6 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 [5]. As before the variable x in the axioms may be instantiated with processes, while the

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 6: Axioms for summation

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.

When substituting a process p for a variable x, then this p does not contain free variables, so in particular in 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 6. 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 single summand instances from a given 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:

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

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

$$\sum_{n:Nat} Xn = X0 + \sum_{m:Nat} XS(m).$$

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:

$$Xn \subseteq X0 + \sum_{m:Nat} XS(m). \tag{2}$$

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

$$\begin{array}{l} X0 + \sum_{m:Nat} XS(m) \overset{\text{SUM3}}{=} \\ X0 + \sum_{m:Nat} XS(m) + XS(n') \supseteq \\ XS(n'). \end{array}$$

As (2) has been proven for all natural numbers n, application of SUM11, SUM4 and SUM1 yields

$$\sum_{n:Nat} Xn \subseteq X0 + \sum_{m:Nat} XS(m)$$

as had to be shown.

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

$$\sum_{n:Nat} Xn \supseteq X0 + XS(m).$$

So SUM11, SUM4 and SUM1 yield:

$$\sum_{n:Nat} Xn \supseteq X0 + \sum_{m:Nat} XS(m).$$

⊠

After α-conversion, namely renaming the bound variable m into n, Lemma 3.4 takes the form

$$\sum_{n:Nat} Xn = X0 + \sum_{n:Nat} XS(n).$$

Equation (1) can now be derived as follows:

$$\begin{array}{ll} & \sum_{n:Nat} r(n) \triangleleft n \leq S(S(0)) \triangleright \delta \\ = & (r(0) \triangleleft 0 \leq S(S(0)) \triangleright \delta) + \sum_{n:Nat} r(S(n)) \triangleleft S(n) \leq S(S(0)) \triangleright \delta \\ = & r(0) + (r(S(0)) \triangleleft S(0) \leq S(S(0)) \triangleright \delta) + \sum_{n:Nat} r(S(S(n))) \triangleleft S(S(n)) \leq S(S(0)) \triangleright \delta \\ = & r(0) + r(S(0)) + (r(S(S(0))) \triangleleft S(S(0)) \leq S(S(0)) \triangleright \delta) \\ & + \sum_{n:Nat} r(S(S(S(n)))) \triangleleft S(S(S(n))) \leq S(S(0)) \triangleright \delta \\ = & r(0) + r(S(0)) + r(S(S(0))) + \delta \\ = & r(0) + r(S(0)) + r(S(S(0))). \end{array}$$

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 3.16 Derive $0 \leq S(S(0)) = \mathsf{t}$, $S(0) \leq S(S(0)) = \mathsf{t}$ and $S(S(0)) \leq S(S(0)) = \mathsf{t}$. Moreover, prove $S(S(S(n))) \leq S(S(0)) = \mathsf{f}$ for all natural numbers n.

Exercise 3.17 Derive

$$\sum_{b:Bool} x \triangleleft b \triangleright y = x + y.$$

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 3.17 that $\sum_{b:Bool} x \triangleleft b \triangleright y = x + y$. An easy corollary of this fact is that $\sum_{b:Bool} x \triangleleft b \triangleright y = \sum_{b:Bool} y \triangleleft b \triangleright x$. So, as $x \triangleleft \mathsf{t} \triangleright y = x$ and $y \triangleleft \mathsf{f} \triangleright x = x$, it is tempting to conclude on the basis of SUM11 that $\sum_{b:Bool} x \triangleleft b \triangleright y = x$. 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 [35]. Note that we assume that we have a function *eq* available, reflecting equality between terms. That is, $eq(d,e) = \mathsf{t}$ if $d = e$ and $eq(d,e) = \mathsf{f}$ if $d \neq e$, for all $d, e{:}D$ (see Section 2.3).

Lemma 3.5 (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 $eq(d,e) = \mathsf{t}$ 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 $eq(d,e) = \mathsf{t}$ or $\neg eq(d,e) = \mathsf{t}$, 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 3.14(3) and the assumption $eq(d,e) = \mathsf{t} \to d = e$ to derive:

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

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 $eq(e,e) = \mathsf{t}$ 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 3.18 Show that if there is some $e{:}D$ such that $b(e) = \mathsf{t}$, 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 [10]. 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 process graph in Figure 3 depicts the behaviour of the bag over $\{0,1\}$, with the root state placed in the leftmost uppermost corner.

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

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

Note that in the case that D is $\{0,1\}$, the process Bag represents the bag over $\{0,1\}$ as depicted in Figure 3. 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 3.19 Give a process declaration of the bag over $\{d_1, d_2\}$ that does not include the three parallel operators.

3.9 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

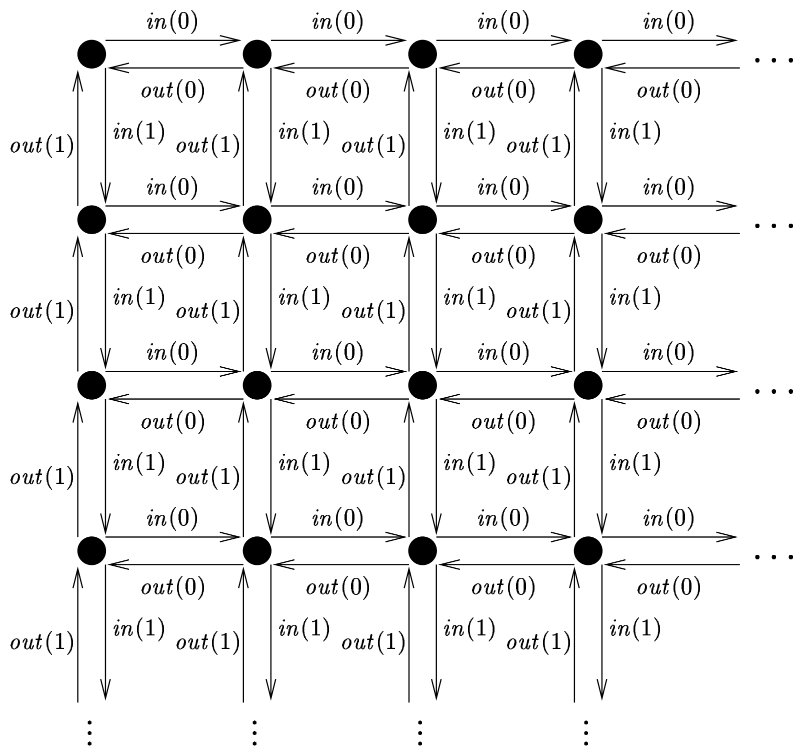
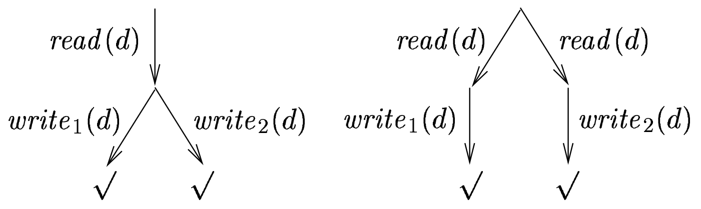


Figure 3: Process graph of the bag over $\{0,1\}$

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

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 3.1. The process graphs 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 [4, 58, 62] 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.

Definition 3.6 (Bisimulation) A *bisimulation relation* $\mathcal{B}$ is a binary relation on processes such that:

1. if $p\,\mathcal{B}\,q$ and $p \overset{a(\vec{d})}{\to} p'$, then $q \overset{a(\vec{d})}{\to} q'$ with $p'\,\mathcal{B}\,q'$;
2. if $p\,\mathcal{B}\,q$ and $q \overset{a(\vec{d})}{\to} q'$, then $p \overset{a(\vec{d})}{\to} p'$ with $p'\,\mathcal{B}\,q'$;
3. if $p\,\mathcal{B}\,q$ and $p \overset{a(\vec{d})}{\to} \surd$, then $q \overset{a(\vec{d})}{\to} \surd$;
4. if $p\,\mathcal{B}\,q$ and $q \overset{a(\vec{d})}{\to} \surd$, then $p \overset{a(\vec{d})}{\to} \surd$.

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 graphs are rooted branching 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 3.7 $(a + a)\cdot b \underline{\leftrightarrow} a\cdot b + a\cdot(b + b)$.

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:

$$\begin{tikzcd}
(a+a)\cdot b \arrow[d, bend right, "a"'] \arrow[d, bend left, "a"] \arrow[rr, dash, dashed] & & a\cdot b+a\cdot(b+b) \arrow[dl, "a"'] \arrow[d, "a"] \\
b \arrow[r, dash, dashed] \arrow[rr, dash, dashed, bend right] \arrow[d, "b"'] & b \arrow[dr, "b"'] & b+b \arrow[d, bend right, "b"'] \arrow[d, bend left, "b"] \\
\surd & & \surd
\end{tikzcd}$$

Exercise 3.20 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 3.21 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 3.22 Let a^1 denote a, and let a^{k+1} denote $a\cdot a^k$ for $k > 0$. Prove that $a^k \not\leftrightarrow a^{k+1}$ for $k > 0$.

4 Abstraction from internal behaviour

In this section it is explained how one can abstract away from internal steps of a process, so that only its external, visible steps remain. As an aside we also explain about renaming of action names, to support the reuse of generic components within a process.

4.1 Internal actions and hiding

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 or internal 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 internal action is meant for analysis purposes, and hardly ever used in specifications, as it is very uncommon to specify that something unobservable must happen.

Typical identities characterising τ are $a\cdot\tau\cdot p = a\cdot p$, with a an action and p a process term. It says that it is by observation impossible to tell whether or not internal actions happen after the a. Sometimes, the presence of internal 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 7 are the characterising laws for the τ. They express that a τ is invisible if it does not lose any possible behaviours (cf. Appendix 4.5). The internal action τ does not communicate with any actions. In order to make actions hidden, the hiding operator τ_I ($I \subseteq \mathsf{Act}$) 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 4.1 Derive the following equations from the axioms.

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	$\tau_I(x{\cdot}y) = \tau_I(x){\cdot}\tau_I(y)$	
SUM9	$\tau_I(\sum_{d:D} X(d)) = \sum_{d:D} \tau_I(X(d))$	

Table 7: Axioms for internal actions and hiding

(1) $a(\tau b + b) = ab$;

(2) $a(\tau(b+c)+b) = a(\tau(b+c)+c)$;

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

(4) Prove that if $y \subseteq x$ then $\tau(\tau x + y) = \tau x$.

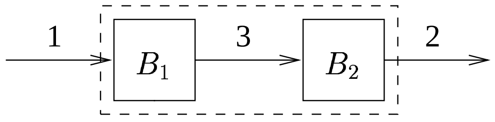
4.2 Overview

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 ($\|$, $⌊\!⌊$, $|$) to express parallelism, deadlock and encapsulation to force actions into communication, τ and hiding to make internal computations invisible, and process declarations to express recursion.

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.

4.3 An example: two 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:



Action $r_i(d)$ represents reading datum d from channel i, while action $s_i(d)$ represents sending datum d into channel i. B_1 and B_2 are defined by process declaration

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

Action $c_3(d)$ denotes communication of datum d through channel 3. Similar as in Example 3.3, $s_3(d) \mid r_3(d) = c_3(d)$, while all other communications between actions result to δ. The system initially consists of buffers B_1 and B_2 in sequence, which is described by the process declaration

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

The encapsulation operator enforces send and read actions over channel 3 into communication, while the hiding operator makes internal 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 D is described by the process declaration

proc $X = \sum_{d:D} r_1(d){\cdot}Y(d)$
$Y(d{:}D) = \sum_{d' \in D} r_1(d'){\cdot}Z(d,d') + s_2(d){\cdot}X$
$Z(d{:}D, d'{:}D) = s_2(d){\cdot}Y(d')$

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 4.2 Give a μCRL specification of the two buffers of capacity one in sequence together with their data types. Use the μCRL tool set 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 D 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 of ACP together with the process declarations yield:

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

So the axioms for deadlock and encapsulation yield:

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

Summarising, we have derived

$$\partial_{\{s_3,r_3\}}(B_2 \parallel B_1) = r_1{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2). \tag{3}$$

We proceed to expand $\partial_{\{s_3,r_3\}}((s_3 B_1) \parallel B_2)$. As above, it can be derived from the axioms of ACP 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

$$\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2) = c_3{\cdot}\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2)). \tag{4}$$

We proceed to expand $\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2 B_2))$. By the axioms of ACP 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,

$$\begin{aligned}
& \partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2)) \\
= \quad & r_1{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2)) + s_2{\cdot}\partial_{\{s_3,r_3\}}(B_2 \parallel B_1).
\end{aligned} \tag{5}$$

We proceed to expand $\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2))$. By the axioms of ACP 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

$$\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2)) = s_2{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2). \tag{6}$$

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 (3) and (4) together with the axioms for τ and hiding yield:

$$\begin{aligned}
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)) &\overset{(3)}{=} \tau_{\{c_3\}}(r_1{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2)) \\
&\overset{\text{TI1,4}}{=} r_1{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2)) \\
&\overset{(4)}{=} r_1{\cdot}\tau_{\{c_3\}}(c_3{\cdot}\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
&\overset{\text{TI2,4}}{=} r_1{\cdot}\tau{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
&\overset{\text{B1}}{=} r_1{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))).
\end{aligned}$$

Moreover, equation (5) together with the axioms for hiding yield:

$$\begin{aligned}
&\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
\overset{(5)}{=}\ &\tau_{\{c_3\}}(r_1{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2)) + s_2{\cdot}\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)) \\
\overset{\text{TI1,3,4}}{=}\ &r_1{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2))) + s_2{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_2 \parallel B_1)).
\end{aligned}$$

Finally, equations (4) and (6) together with the axioms for τ and hiding yield:

$$\begin{aligned}
\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel (s_2{\cdot}B_2))) &\overset{(6)}{=} \tau_{\{c_3\}}(s_2{\cdot}\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2)) \\
&\overset{\text{TI1,4}}{=} s_2{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}((s_3{\cdot}B_1) \parallel B_2)) \\
&\overset{(4)}{=} s_2{\cdot}\tau_{\{c_3\}}(c_3{\cdot}\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
&\overset{\text{TI2,4}}{=} s_2{\cdot}\tau{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))) \\
&\overset{\text{B1}}{=} s_2{\cdot}\tau_{\{c_3\}}(\partial_{\{s_3,r_3\}}(B_1 \parallel (s_2{\cdot}B_2))).
\end{aligned}$$

The last three derivations together show that

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

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

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

4.4 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 8.

R1	$\rho_f(\delta) = \delta$
R2	$\rho_f(\tau) = \tau$
R3	$\rho_f(a(\vec{d})) = f(a)(\vec{d})$
R4	$\rho_f(x + y) = \rho_f(x) + \rho_f(y)$
R5	$\rho_f(x{\cdot}y) = \rho_f(x){\cdot}\rho_f(y)$
SUM10	$\rho_f(\sum_{d:D} X(d)) = \sum_{d:D} \rho_f(X(d))$

Table 8: Axioms for renaming

Exercise 4.3 Use the renaming operator to extract the buffer B_2 Exercise 4.2 from the process declaration of B_1.

4.5 Branching bisimulation equivalence

While on the level of process terms with internal actions we have defined an equality relation, we have not yet introduced a way to relate process graphs. In this section we define the notion of *rooted branching bisimulation equivalence* on process graphs. This equivalence agrees with the equality relation on process terms, in the sense that if two terms can be equated then their graphs 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.

As the equality relation on terms is closed under contexts (i.e., $p = q$ implies $C[p] = C[q]$ for all contexts $C[]$), we want the equivalence relation on graphs to satisfy this same property.

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

which τ's are silent?

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

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

Example 4.1 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.

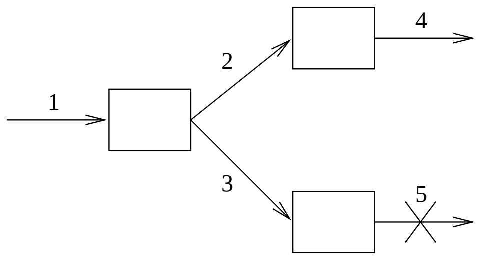


Figure 4: Protocol with a malfunctioning channel

The system, which is depicted in Figure 4, is described by the process term

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

where $s_i(d)$, $r_i(d)$, and $c_i(d)$ represent a send, read, and communication action of the datum d via channel i, respectively. 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 second τ in this term cannot be deleted, because then the process would no longer be able to get into a deadlock. Hence, this τ is not silent.

As a further example of a τ-transition that is not silent, 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 internal steps. A correct answer to the question

which τ-transitions are silent?

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 silent: 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 silent, since execution of this τ means losing the option to execute a.

The intuition above is formalised in the notion of *branching bisimulation equivalence* [32]. 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 silent, meaning that p' and q are branching bisimilar. Moreover, a non-silent transition $p\xrightarrow{a(\vec{d})}p'$ need not be simulated by q immediately, but only after a number of silent 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 silent) 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 4.2 (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 binary relation on the collection of processes such that:

1. if $p\,\mathcal{B}\,q$ and $p\xrightarrow{a(\vec{d})}p'$, then
   - either $a=\tau$ and $p'\,\mathcal{B}\,q$;
   - or there is a sequence of (zero or more) τ-transitions $q\xrightarrow{\tau}\cdots\xrightarrow{\tau}q_0$ such that $p\,\mathcal{B}\,q_0$ and $q_0\xrightarrow{a(\vec{d})}q'$ with $p'\,\mathcal{B}\,q'$;
2. if $p\,\mathcal{B}\,q$ and $q\xrightarrow{a(\vec{d})}q'$, then
   - either $a=\tau$ and $p\,\mathcal{B}\,q'$;
   - or there is a sequence of (zero or more) τ-transitions $p\xrightarrow{\tau}\cdots\xrightarrow{\tau}p_0$ such that $p_0\,\mathcal{B}\,q$ and $p_0\xrightarrow{a(\vec{d})}p'$ with $p'\,\mathcal{B}\,q'$.

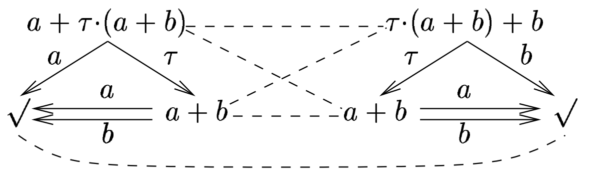
3. if $p\,\mathcal{B}\,q$ and $p\downarrow$, then there is a sequence of (zero or more) τ-transitions $q \xrightarrow{\tau} \cdots \xrightarrow{\tau} q_0$ such that $q_0\downarrow$;

4. if $p\,\mathcal{B}\,q$ and $q\downarrow$, then there is a sequence of (zero or more) τ-transitions $p \xrightarrow{\tau} \cdots \xrightarrow{\tau} p_0$ such that $p_0\downarrow$.

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 4.3 $a+\tau\cdot(a+b) \,\underline{\leftrightarrow}_b\, \tau\cdot(a+b)+b$.

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:



$$\begin{tikzcd}
& a+\tau\cdot(a+b) \arrow[dl, "a"'] \arrow[dr, "\tau"] \arrow[rrr, dash, dashed] \arrow[drr, dash, dashed] & & & \tau\cdot(a+b)+b \arrow[dl, "\tau"'] \arrow[dr, "b"] \arrow[dll, dash, dashed] & \\
\surd & & a+b \arrow[ll, shift right, "a"'] \arrow[ll, shift left, "b"'] \arrow[r, dash, dashed] & a+b \arrow[rr, shift left, "a"] \arrow[rr, shift right, "b"'] & & \surd \arrow[lllll, dash, dashed, bend left]
\end{tikzcd}$$

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

Exercise 4.4 Give branching bisimulation relations to prove that the process terms a, $a\cdot\tau$, and $\tau\cdot a$ are branching bisimilar.

Exercise 4.5 Give a branching bisimulation relation to prove that the process terms $\tau\cdot(\tau\cdot(a+b)+b)+a$ and $a+b$ are branching bisimilar.

Exercise 4.6 Assume a process graph, and let the states s and s' in this process graph be on a τ-loop; that is, there exist sequences of τ-transitions $s \xrightarrow{\tau} \cdots \xrightarrow{\tau} s'$ and $s' \xrightarrow{\tau} \cdots \xrightarrow{\tau} s$. Prove that s and s' are branching bisimilar.

Groote and Vaandrager [44] 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 8.1.

Branching bisimilarity is an equivalence relation; see [7]. Branching bisimulation equivalence, however, is not a congruence with respect to BPA. For example, $\tau\cdot a$ and a are branching bisimilar (see Exercise 4.4), 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 silent.

Milner [59] showed that this problem can be overcome by adding a rootedness condition: initial τ-transitions are never silent. 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 4.4 (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 binary relation on processes such that:

1. if $p\,\mathcal{B}\,q$ and $p \xrightarrow{a} p'$, then $q \xrightarrow{a} q'$ with $p' \leftrightarrow_b q'$;
2. if $p\,\mathcal{B}\,q$ and $q \xrightarrow{a} q'$, then $p \xrightarrow{a} p'$ with $p' \leftrightarrow_b q'$;
3. if $p\,\mathcal{B}\,q$ and $p\downarrow$, then $q\downarrow$;
4. if $p\,\mathcal{B}\,q$ and $q\downarrow$, then $p\downarrow$.

Two processes p and q are *rooted branching bisimilar*, denoted by $p \leftrightarrow_{rb} q$, 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 4.7 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 $(b+a)\cdot(d+c) + a\cdot(c+d)$;

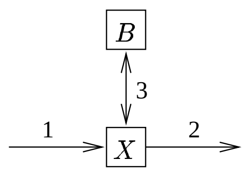
(3) $\tau\cdot(b+a) + \tau\cdot(a+b)$ and $a+b$;

(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 4.8 Data elements (of a collection D) 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 $r_1, s_2, s_3, r_3, c_3{:}D$
comm $s_3\,|\,r_3 = c_3$
proc $X = \sum_{d:D} r_1(d)\cdot(s_2(d)\cdot X + s_3(d)\cdot Y(d))$
$Y(d{:}D) = r_3(d)\cdot(s_2(d)\cdot X + s_3(d)\cdot Y(d)) + \sum_{d':D} r_1(d')\cdot s_2(d')\cdot Y(d)$
$B = \sum_{d:D} r_3(d)\cdot s_3(d)\cdot B$

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

- Draw the process graph of t.
- Are data elements read via channel 1 and sent via channel 2 in the same order?
- Does $\partial_{\{s_2(d_1)\}}(t)$ contain a deadlock? If yes, then give an execution trace to a deadlock state.
- Draw the process graph of $\tau_{\{c_3(d_1),c_3(d_2)\}}(t)$. Distinguish the silent τ-transitions from the non-silent ones.

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

Exercise 4.9 Data elements (of a collection D) 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.

$$\begin{tikzcd} {} \arrow[r, "1"] & \boxed{X} \arrow[r, leftrightarrow, "2"] & \boxed{Y} \arrow[r, "3"] & {} \end{tikzcd}$$

X and Y are defined by the following μCRL specification:

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

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

- Draw the process graph of t.
- Are data elements read via channel 1 and sent via channel 3 in the same order?
- Does $\partial_{\{s_3(d_1)\}}(t)$ contain a deadlock? If yes, then give an execution trace to a deadlock state.
- Draw the process graph of $\tau_{\{c_2(d_1),c_2(d_2)\}}(t)$ after minimisation modulo branching bisimulation equivalence.

5 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 7 we will verify the correctness of some of these protocols.

5.1 Alternating bit protocol

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

The alternating bit protocol (ABP) [6] 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 5.

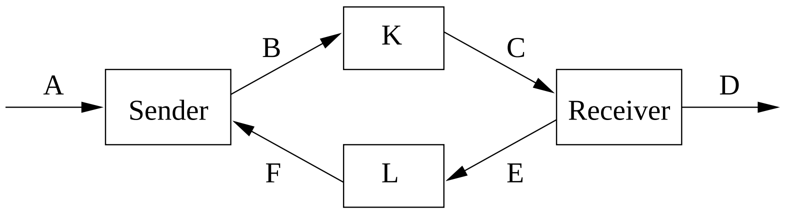


Figure 5: Alternating bit protocol

Data elements $d_1, d_2, d_3, \ldots$ from a finite 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, 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 $b \in \{0,1\}$:

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

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 $\bot$, then the system sends the pair (d,b) into channel B once more.

Next, we specify the Receiver in the state that it is expecting to receive a datum with the bit b attached to it, represented by the process name $R(b)$ for $b \in \{0,1\}$:

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

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 $\bot$ from channel C, then this does not constitute new information. So then the Receiver sends acknowledgement $1-b$ to the Sender via channel E and remains in state $R(b)$.

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

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

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:

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

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 process graph of $\partial_H(S(0)\parallel R(0)\parallel K\parallel L)$ is depicted in Figure 6.

Exercise 5.1 Give a μCRL specification of the ABP together with its data types. Use the μCRL tool set to analyse this specification, for $\Delta=\{d_1,d_2\}$.

Exercise 5.2 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 tool set that in that case data elements could get lost.

5.2 Bounded retransmission protocol

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

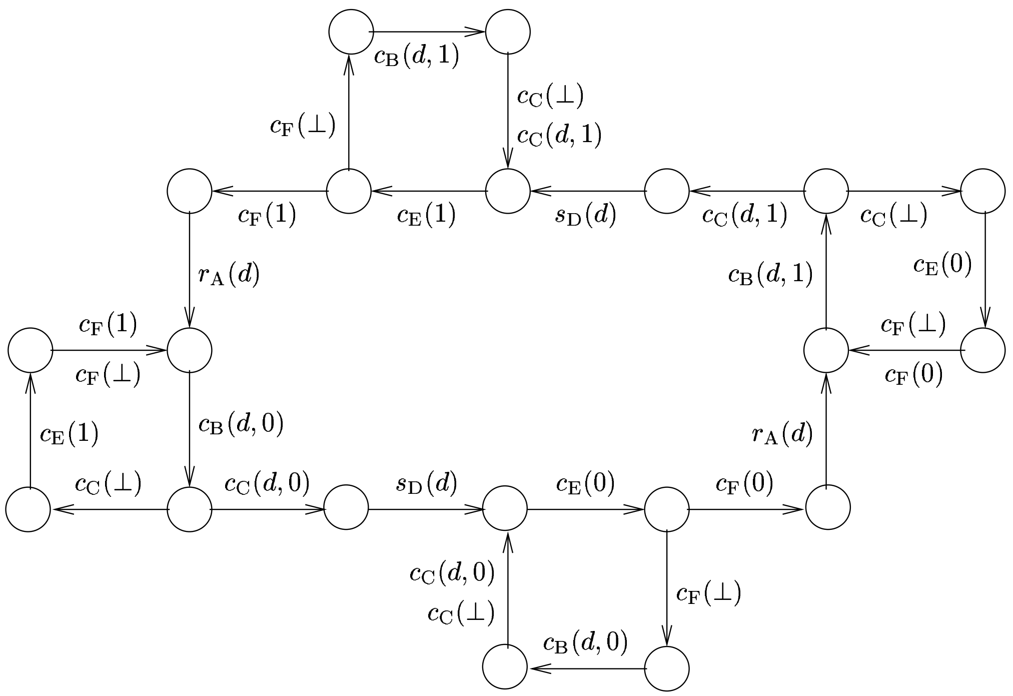


Figure 6: Transition graph of $\partial_H(S(0) \parallel R(0) \parallel K \parallel L)$.

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.

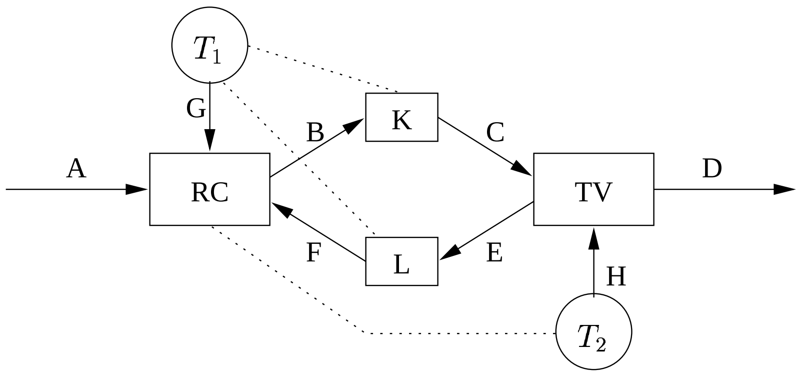


Figure 7: Bounded retransmission protocol

The BRP is depicted in Figure 7. 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 [38] 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, 23, 38, 46].

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 [38], 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 8. 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 8, intertwined with non-invisible τ-transitions.

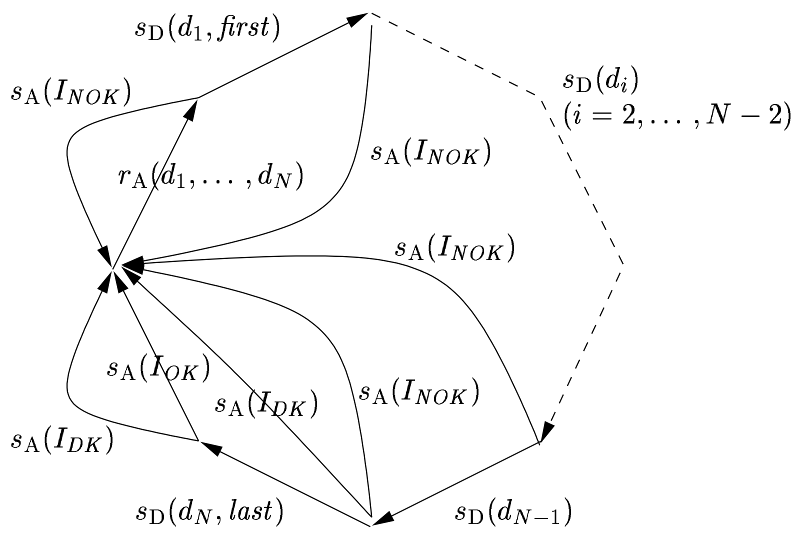


Figure 8: External behaviour of the BRP

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 finite 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 $tail(d_1, \ldots, d_N) = (d_2, \ldots, d_N)$.

- b ranges over $\{0, 1\}$, while n ranges over $\{0, \ldots, max\}$, 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:

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

The intuition behind these process declarations is as follows. Let l range over lists of data of length ≥ 1.

- 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 $\mathit{head}(\ell)$, 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 $\mathit{tail}(l)$ 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 $n < \mathit{max}$, then it sends the pair $(\mathit{head}(l),b)$ to the TV again, with the counter increased by one. If $n = \mathit{max}$, 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:

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

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:

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

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 8), intertwined with non-silent τ-transitions.

Exercise 5.3 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.

5.3 Sliding window protocol

In the ABP and the BRP, the Sender sends out a datum and then waits for an acknowledgement or a time-out before it sends the next datum. In situations where transmission of data is relatively time consuming, this procedure tends to be unacceptably slow. In sliding window protocols (see [70]), a Sender can send out data elements without waiting for acknowledgements. See [13, 17] for specifications and verifications of sliding window protocols in process algebra.

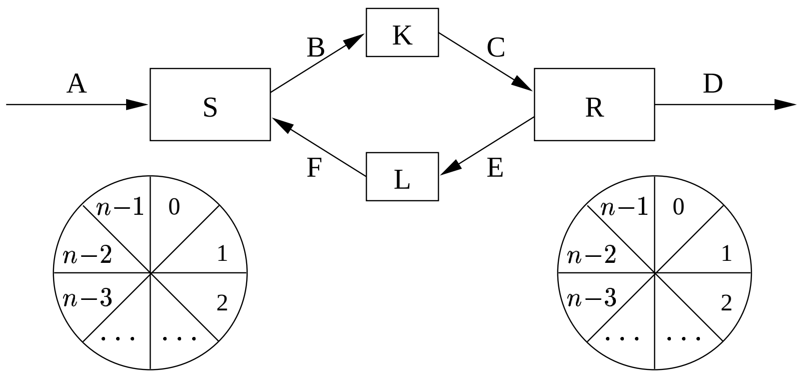


Figure 9: Sliding window protocol

Again we assume that the elements from a 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; see Figure 9. 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 $k \in \mathit{Nat}$, representing that position k of the buffer is occupied by datum d. The data type *Buffer* is specified as follows, where [] denotes the empty buffer:

$$\begin{array}{l} [] :\to \mathit{Buffer} \\ \mathit{in} : \Delta \times \mathit{Nat} \times \mathit{Buffer} \to \mathit{Buffer} \end{array}$$

The nature of positions in buffers is firmly linked with so-called *modulo arithmetic* (see, e.g., [24]). Two natural numbers are considered equal *modulo* n if their difference is divisible by n. It is not hard to see that this defines an equivalence relation on natural numbers, with equivalence classes $0, \ldots, n-1$.

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

For $k \in \{0, \ldots, n-1\}$, let $\mathit{succmod}(k)$ denote the equivalence class of the successor of k modulo n; in particular, $\mathit{succmod}(n-1) = 0$. The operator $\mathit{succmod} : \mathit{Nat} \to \mathit{Nat}$ is defined by the following equation, where $\mathit{if} : \mathit{Bool} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat}$ is the if-then-else function from Exercise 2.6 and $\mathit{eq} : \mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$ is the equality function on natural numbers from Section 2.3:

$$\mathit{succmod}(m) \;=\; \mathit{if}(\mathit{eq}(S(m), n), 0, S(m))$$

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 n. 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 n), 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 a predefined size n, and that the sliding windows in these buffers are restricted to *max-fill* positions, where *max-fill* is not allowed to exceed $n/2$. Again, in the μCRL specification of the SWP, one can declare $max-fill :\to Nat$, and this function symbol of arity zero can be instantiated with its desired value in the algebraic specification of the natural numbers.

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:

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

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 $in(d,k,in(d',k,[]))$, because $remove(k,in(d,k,in(d',k,[])))$ would produce the erroneous result $in(d',k,[])$. In order to support the assumption that there is no overloading of positions in buffers, it is essential that at the right-hand side of the equation for $add(d,k,t)$, position k in t is emptied.

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

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

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

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

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 $\{0,\dots,n-1\}$, then with the *range* from k to ℓ (modulo n) we mean either $\ell-k+1$ (i.e., the number of elements in $\{k,\dots,\ell\}$) if $k \leq \ell$, or $n-k+\ell+1$ (i.e., the number of elements in $\{k,\dots,n-1\} \cup \{0,\dots,\ell\}$) 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 n, *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*.

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

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 n:

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

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 n, while *first-in* represents the first position in the window of *buffer*.

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

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

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

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.

Ideally, 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. It can be shown, using the μCRL tool set, that this property holds for specific data sets Δ, and for small buffer sizes n. 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 5.4 Define a function *plusmod* : $Nat \times Nat \to Nat$, where for $k, \ell \in \{0, \ldots, n-1\}$ $plusmod(k, \ell)$ produces the equivalence class of $k + \ell$ modulo n in $\{0, \ldots, n-1\}$.

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 n; that is, if $k \leq m < n$ then $\ell \in \{k, \ldots, m-1\}$, and if $m < k < n$ then $\ell \in \{k, \ldots, n-1\} \cup \{0, \ldots, m-1\}$.

Use the functions *plusmod* and *ordered* to give an algebraic formulation of "the range from k to ℓ does not exceed m".

Exercise 5.5 Give a μCRL specification of the SWP and its data types. Use the μCRL tool set to analyse this specification, where $\Delta = \{d_1, d_2\}$, the buffer size n is four, and *max-fill* is two.

Exercise 5.6 Suppose the buffer size n is three and *max-fill* is two (violating the restriction that *max-fill* must be no more than $n/2$). 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, 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 10. 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 5.7 Give a μCRL specification of the two-way SWP. Use renaming to extract the Receiver and L from the process declarations of the Sender and K, respectively.

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

Piggybacking as described in Exercise 5.8 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),

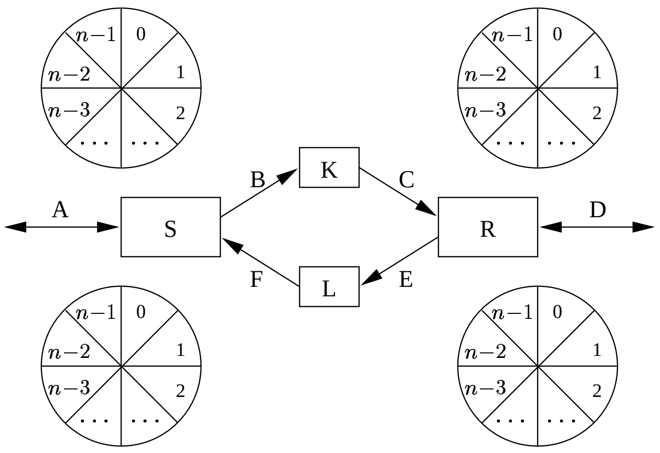


Figure 10: Two-way sliding window protocol

which sends a time-out message if an acknowledgement must be sent out without further delay; see [70] for more details.

Exercise 5.9 Give a μCRL specification of the two-way SWP with piggybacking supplied with timers.

Bezem and Groote [13] 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.

5.4 Tree identify protocol

IEEE 1394 connects together 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 [56] and using E-LOTOS in [68]. The former is essentially a description only, with five correctness properties stated informally, but not formalised or proved. The exercise of [68] is based on the μCRL description, adding another layer of the protocol and carrying out the verification suggested, using the tool CADP [30].

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 [67], 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 [49].

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 7.

For a formal specification of the TIP using I/O automata, and for an analysis of timing aspects and probabilities, see [27, 65, 69].

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.

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

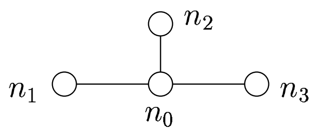
The initial state of Implementation A consists of the parallel composition of the node processes for the nodes $n_1, \ldots, n_k$ in state 0, with as possible parents the lists of all neighbours $p_1, \ldots, p_k$:

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

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

Example 5.1 The network



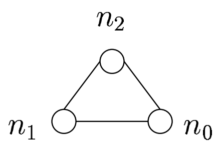
is captured by

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

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:

Exercise 5.10 Explain why the τ-transitions in the external behaviour depicted in Figure 11 are not silent.

Exercise 5.11 Consider the network



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

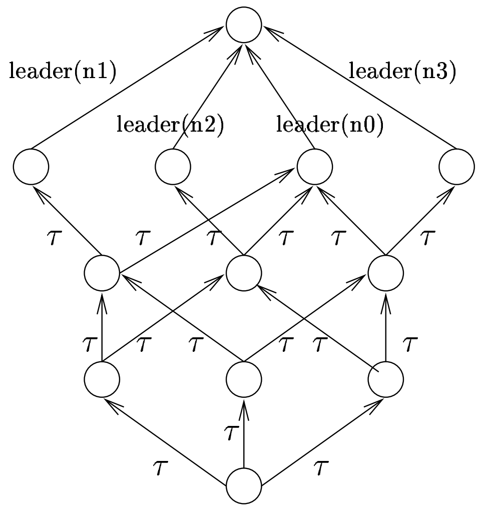


Figure 11: External behaviour of the TIP

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.

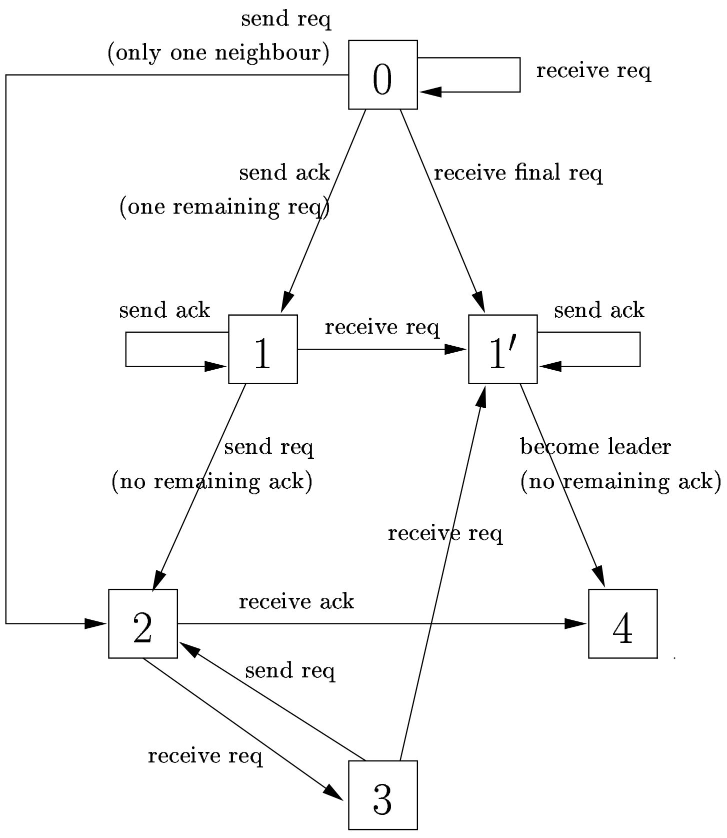


Figure 12: Relations between the five states

The relations between the five states of a node are depicted in Figure 12. 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.

In the μCRL specification of Implementation B, the node i in state s is represented by the process name $\mathit{Node}(i,p,q,s)$, 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:

- $\mathit{rem}(j,p)$ removes node j from list p;
- $\mathit{single}(p)$ tests whether list p contains exactly one element, while $\mathit{single}(p,j)$ tests whether list p consists of a single node j;
- $\mathit{occur}(j,p)$ tests whether node j occurs in list p;
- $\mathit{empty}(p)$ tests whether list p is empty.

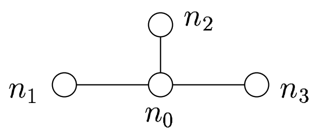
$$\begin{array}{rcl}
\mathit{Node}(i,p,q,0) & = & \sum_{j:N} r(j,i,\mathit{req})\cdot \mathit{Node}(i,\mathit{rem}(j,p),\mathit{in}(j,q),\mathit{if}(\mathit{single}(p),1,0)) \lhd \mathit{occur}(j,p) \rhd \delta \\
& + & \sum_{j:N} s'(i,j,\mathit{ack})\cdot \mathit{Node}(i,p,\mathit{rem}(j,q),1) \lhd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

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:

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

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 5.2 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 11.

6 Verification techniques

In the previous chapter we presented a number of μCRL specifications. The μCRL tool set can be applied to show that these specifications exhibit the desired external behaviour for *specific* sets of data. We now set out to prove that these specifications exhibit the desired external behaviour for *general* sets of data.

In this chapter we present mathematical tools that will help to analyse the external behaviour of μCRL specifications. In the next chapter these mathematical tools will be used for the formal verification of Implementation A of the TIP.

6.1 Linear process equations

A *linear process equation* (LPE) [12] 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.

Definition 6.1 (Linear process equation) A *linear process equation* is a process declaration of the form

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

where the I is some index set, and for each $i{:}I$ we have $a_i \in \mathsf{Act} \cup \{\tau\}$, $f_i : D \times E_i \to D_i$, $g_i : D \times E_i \to D$ and $h_i : D \times E_i \to Bool$.

Intuitively, in the LPE in Definition 6.1, 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_i for $i{:}I$, carrying a data parameter $f_i(d,x_i)$, under the condition that $h_i(d,x_i)$ is true; in this case the resulting state is $g_i(d,x_i)$. The data types E_i help to give LPEs a more general form that is widely applicable, as not only the data parameter $d{:}D$, but also the data parameter $x_i{:}E_i$ can influence the parameter of action a_i, the condition h_i and the resulting state g_i.

Given an LPE X over a data domain $d{:}D$, terms

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

for $d_0, \ldots, d_n{:}D$ can be equated to $Y(d_0, \ldots, d_n)$, where Y is an LPE. Groote and van Wamel [45] presented a formal proof of this fact, using the derivation rule CL-RSP that will be explained in the next section; moreover, they spelled out the precise syntactic form of Y. We give an example; note in particular that a communication of actions in the parallel composition leads to a conjunction of conditions in the resulting LPE.

Example 6.2 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 $k{:}Nat$, the parallel composition $X(n_1) \parallel \cdots \parallel X(n_k)$ is equal to $Y(n_1, \ldots, n_k)$, where

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

Here, $Y(n_i := n_i + 1)$ 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.

Exercise 6.1 Describe the following process declaration by means of an LPE:

$$\begin{array}{lcl}
X & = & Y \\
Y & = & a{\cdot}c{\cdot}X
\end{array}$$

Exercise 6.2 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 $k{:}Nat$, the parallel composition $X(n_1) \parallel \cdots \parallel X(n_k)$ is equal to $Y(n_1, \ldots, n_k)$

Groote, Ponse and Usenko [42] 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 [42] underlies the `mcrl` command of the μCRL tool set, which transforms a parallel pCRL specification into an LPE; see [73]. Currently, applying `mcrl` to a μCRL specification that lies outside parallel pCRL produces an error message; a lineariser for full μCRL is under construction.

6.2 CL-RSP

Definition 6.3 (Convergent LPE) The LPE in Definition 6.1 is *convergent* if there exists a well-founded ordering $<$ on D such that for all $i{:}I$ with $a_i = \tau$ we have that, for all $e_i{:}E_i$ and $d{:}D$, $h_i(d, e_i)$ implies $g_i(d, e_i) < d$.

If an LPE is convergent, then it cannot exhibit an infinite sequence of τ-transitions. Namely, Definition 6.3 guarantees that for each τ-transition, the original state d is strictly greater than the resulting state $g_i(d, e_i)$, 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 elements in D.

CL-RSP (*Convergent Linear Recursive Specification Principle*) [12] says that every convergent LPE has at most one solution.

Theorem 6.4 (CL-RSP) *Let the LPE*

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

be convergent. If $t(d)$ ranges over processes such that, for all $d{:}D$,

$$t(d) \;=\; \textstyle\sum_{i:I} \sum_{x_i:E_i} a_i(f_i(d, x_i))\cdot t(g_i(d, x_i)) \lhd h_i(d, x_i) \rhd \delta$$

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

Convergence is essential for the soundness of CL-RSP. For example, consider the LPE $X = \tau\cdot X \lhd \mathsf{t} \rhd \delta$; note that this LPE not convergent. By axioms B1 and C1 (see Tables 7 and 5, respectively) we have, for all $a \in \mathsf{Act}$, $\tau\cdot a = \tau\cdot\tau\cdot a = \tau\cdot\tau\cdot a \lhd \mathsf{t} \rhd \delta$. 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, in the sense that $\tau\cdot a$ and $\tau\cdot b$ are not branching bisimilar if $a \neq b$.

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

Example 6.5 Consider the following two LPEs:

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

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

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 $m{:}Nat$.

Exercise 6.3 Give a formal proof, using CL-RSP, that the parallel composition $X(n_1) \parallel \cdots \parallel X(n_k)$ in Example 6.2 is equal to $Y(n_1, \ldots, n_k)$, for $k = 2$.

6.3 Invariants

Definition 6.6 (Invariant) A mapping $\mathcal{I} : D \to \mathit{Bool}$ is an *invariant* for the LPE in Definition 6.1 if, for all $i{:}I$, $d{:}D$ and $e_i{:}E_i$,

$$\mathcal{I}(d) \wedge h_i(d, e_i) \;\Rightarrow\; \mathcal{I}(g_i(d, e_i)) \tag{7}$$

Intuitively, an invariant characterises the set of reachable states of an LPE. That is, if $\mathcal{I}(d) = \mathsf{t}$ and if one can evolve from state d to state d' in zero or more transitions, then $\mathcal{I}(d') = \mathsf{t}$. Namely, if $\mathcal{I}$ holds in state d and it is possible to execute $a_i(f_i(d, e_i))$ in this state (meaning that $h_i(d, e_i) = \mathsf{t}$), then it is ensured by (7) that $\mathcal{I}$ holds in the resulting state $g_i(d, e_i)$. Invariants tend to play a crucial role in algebraic verifications of system behaviour that involve data.

Example 6.7 Consider the LPE $X(n{:}\mathit{Nat}) \;=\; a(n){\cdot}X(n + 2)$. Invariants for this LPE are

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

Exercise 6.4 Show that the mappings $\mathcal{I}_1$ and $\mathcal{I}_2$ in Example 6.7 are invariants.

Exercise 6.5 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 6.6 Consider the LPE

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

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 $n{:}\mathit{Nat}$ as possible.)

In Theorem 6.4, 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 only necessary 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 $\mathcal{I}(d) = \mathsf{t}$.

Theorem 6.8 (CL-RSP with invariants) *Let the LPE*

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

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

$$t(d) \;=\; \textstyle\sum_{i:I} \sum_{x_i:E_i} a_i(f_i(d, x_i)){\cdot}t(g_i(d, x_i)) \lhd h_i(d, x_i) \rhd \delta$$

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

Example 6.9 Let *even* : $Nat \to Bool$ map even numbers to t and odd numbers to f. Consider the following two LPEs:

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

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 6.7), and this LPE is convergent, by Theorem 6.8

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

for even n.

6.4 Cones and foci

The *cones and foci* technique of [43] aims to eliminate internal actions from an LPE. The main idea of this technique is that usually internal events progress silently towards a state in which no internal 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 internal actions. Imagine the transition system forming a cone or funnel pointing towards the focus. Figure 13 visualises the core idea underlying this method. Note that the visible 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 internal actions in the cone would not be silent.

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 without internal actions (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 $a \neq \tau$:

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

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 $\mathcal{I}(d) = \mathsf{t}$, where $\mathcal{I}$ is some invariant for X.

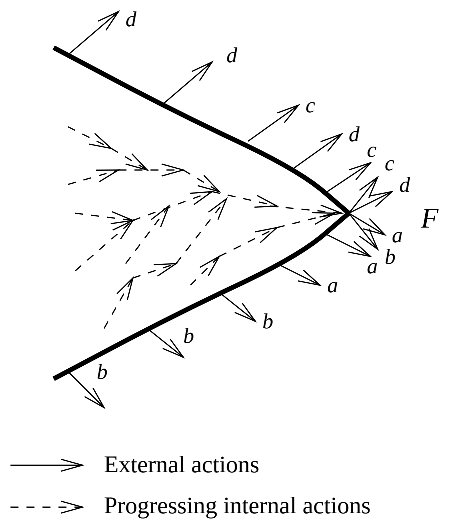
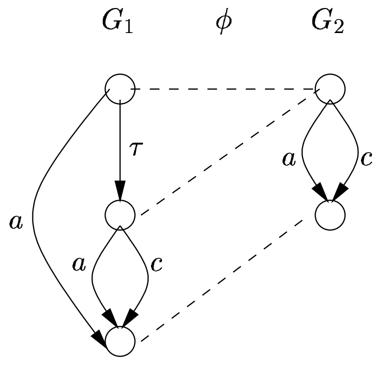


Figure 13: A focus point and its cone

Example 6.10 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. One is allowed to appoint the focus points in the LPE X by means of a *focus condition* $FC_X : D \to Bool$; ideally, this focus condition takes the form of a logical formula. Furthermore, in the cones and foci method as explained below it is assumed that an abstraction operator τ_I is applied to the LPE X, and the actions from I in X assume the role of internal actions (i.e., if $d \xrightarrow{c} (\vec{e})d'$ with $c \in I$, then $\phi(d) = \phi(d')$). The LPE Y must not contain any actions from I.

Definition 6.11 (Focus point) Assume an LPE X over D. A *focus condition* is a mapping

$FC_X : D \to Bool$. If $FC(d) = \mathsf{t}$, then d is called a *focus point* of X.

Definition 6.12 (Matching criteria) Let $I \subseteq \mathsf{Act}$. Assume an LPE

$$X(d{:}D) = \sum_{a:\mathsf{Act}} \sum_{x_a:E_a} a(f_a(d,x_a)){\cdot}X(g_a(d,x_a)) \lhd h_a(d,x_a) \rhd \delta$$

with a focus condition FC_X. Furthermore, assume an LPE without actions from I

$$Y(d'{:}D') = \sum_{b:\mathsf{Act}\backslash I} \sum_{x_b:E_b} b(f'_b(d',x_b)){\cdot}Y(g'_b(d',x_b)) \lhd h'_b(d',x_b) \rhd \delta$$

A state mapping $\phi : D \to D'$ satisfies the *matching criteria* with respect to I if for all $c \in I$ and $b \in \mathsf{Act}\backslash I$:

- $h_c(d,e_c) \Rightarrow \phi(d) = \phi(g_c(d,e_c))$;
- $h_b(d,e_b) \Rightarrow h'_b(\phi(d),e_b)$;
- $(FC_X(d) \wedge h'_b(\phi(d),e_b)) \Rightarrow h_b(d,e_b)$;
- $h_b(d,e_b) \Rightarrow f_b(d,e_b) = f'_b(\phi(d),e_b)$;
- $h_b(d,e_b) \Rightarrow \phi(g_b(d,e_b)) = g'_b(\phi(d),e_b)$.

The first matching criterion in Definition 6.12 requires that all the states in a cone of $\tau_I(X)$ are mapped to the same state of Y. The second criterion expresses that each visible action in $\tau_I(X)$ can be simulated. Moreover, the fourth criterion ensures that these actions carry the same data parameters, while by the fifth criterion the resulting states in $\tau_I(X)$ and Y are still related. Finally, the third criterion expresses that if a state in Y can perform an external action, then the corresponding *focus points* in $\tau_I(X)$ are also able to perform this action.

Theorem 6.13 (General Equality Theorem) *Let $I \subseteq \mathsf{Act}$. Assume an LPE X over D with a focus condition FC_X. Let $\mathcal{I}$ be an invariant for X. Suppose each $d{:}D$ with $I(d) = \mathsf{t}$ can reach a focus point of X by executing a sequence of actions from I. Furthermore, assume an LPE Y without actions from I. If $\phi : D \to D'$ satisfies the matching criteria from Definition 6.12, with respect to I, for all $d{:}D$ with $\mathcal{I}(d) = \mathsf{t}$, then for all $d{:}D$ with $\mathcal{I}(d) = \mathsf{t}$:*

$$\tau{\cdot}\tau_I(X(d)) = \tau{\cdot}Y(\phi(d)).$$

As always, the invariant $\mathcal{I}$ in Theorem 6.13 restricts the collection of reachable states of X.

Exercise 6.7 Let

$$\begin{array}{rcl} X(n{:}Nat) & = & \sum_{m:Nat}(a{\cdot}X(n+1) \lhd n = 3m \rhd \delta + d{\cdot}X(n+1) \lhd n = 3m+1 \rhd \delta \\ & & + c{\cdot}X(n+1) \lhd n = 3m+2 \rhd \delta) \\ Y(b{:}Bool) & = & a{\cdot}Y(\mathsf{f}) \lhd b \rhd \delta + d{\cdot}Y(\mathsf{t}) \lhd \neg b \rhd \delta \end{array}$$

Give a focus condition FC_X together with a state mapping $\phi : Nat \to Bool$ that satisfies the matching criteria with respect to $\{c\}$.

7 Verification of the tree identify protocol

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 Section 6. Therefore, these verifications are omitted here; the reader is referred to [29] for expositions on these verifications. As far as we know, the external behaviour of the SWP has not yet been verified within a process algebraic framework. See [13] for a verification of a restricted version of the SWP, where the size of the window is limited to one.

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 5.4) produces a unique root. This proof uses the verification techniques explained in Section 6.

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

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

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 $\tau\cdot leader\cdot\delta$. 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 $\{0,1\}$,

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

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

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

See Example 6.2 and Exercise 6.3 for examples of how to derive such an equality, using CL-RSP.

Exercise 7.1 Prove $\partial_H(X(n_0,p[n_0],s[n_0])\|\cdots\|X(n_k,p[n_k],s[n_k])) = Y(p,s)$ for pairs (p,s).

We list four invariants for the LPE Y.

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

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

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

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

Note that $\mathcal{I}_n(p_0, s_0) = \mathsf{t}$ 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 $c(j,i)$, 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 $c(i,j)$, 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 $c(j,i)$, 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 7.2 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 7.1 (Uniqueness of the root)

$$\mathcal{I}(p,s): \quad (\mathit{empty}(p[i]) \wedge j \neq i) \Rightarrow s[j] = 1$$

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 $s[i_m] = 1$. ⊠

For each pair (p, s), let $\mathit{busy}(p, s)$ return a node i with $s[i] = 0$ (if there is such a node). By uniqueness of the root, at any time $\mathit{busy}(p, s)$ is the only node that can perform a *leader* action. Therefore, using CL-RSP and Lemma 7.1, the summation sign in front of the *leader* action in the LPE Y can be eliminated, by instantiating $\mathit{busy}(p, s)$ for the parameter i of this summation. Thus we obtain the following LPE.

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

Intuitively, we say that (p, s) is a *focus point* of Y if $Y(p, s)$ cannot execute any $c(j, i)$-actions. To be more precise, the *focus condition* $FC_Y(p, s)$ is that there do not exist nodes i and j with

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

If the focus condition does not hold for some pair (p, s), then clearly $Y(p, s)$ can execute a $c(j, i)$-action. Since each $c(j, i)$-action reduces the number of nodes j with $s[j] = 0$, there does not exist an infinite execution sequence of such actions. Hence, for each pair (p, s), $Y(p, s)$ can execute a sequence of $c(j, i)$-actions to reach a focus point for Y.

The LPE for the external behaviour is

$$Z(b{:}\mathit{Bool}) \ = \ \mathit{leader}\cdot Z(\mathsf{f}) \lhd b \rhd \delta$$

Clearly, $Z(\mathsf{t}) = \mathit{leader}\cdot\delta$ and $Z(\mathsf{f}) = \delta$.

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 6.12, applied to the LPEs Y and Z, produce the following four formulas:

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

$$(\mathit{empty}(p[\mathit{busy}(p,s)]) \wedge s[\mathit{busy}(p,s)] = 0) \;\Rightarrow\; \phi(p,s);$$

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

$$(\mathit{empty}(p[\mathit{busy}(p,s)]) \wedge s[\mathit{busy}(p,s)] = 0) \;\Rightarrow\; \phi(p, s[\mathit{busy}(p,s)] := 1) = \mathsf{f}.$$

We show that the first three of these matching criteria hold.

1. $\phi(p,s) = \mathsf{t} = \phi(p[i]\backslash\{j\}, s[j] := 1)$, because $s[i]$ remains 0.

2. $\phi(p,s) = \mathsf{t}$, because $s[\mathit{busy}(p,s)] = 0$.

3. Since $\phi(p,s) = \mathsf{t}$ there is a node i with $s[i] = 0$, so also $s[\mathit{busy}(p,s)] = 0$. Suppose $p[\mathit{busy}(p,s)]$ is not empty; we derive a contradiction.

   Let $j \in p[i]$ and $p[i] = 0$ for some 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.

Exercise 7.3 Prove that the fourth matching criterion above holds.

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

$$\tau{\cdot}\tau_I(Y(p,s)) \;=\; \tau{\cdot}Z(\phi(p,s))$$

This implies

$$\begin{array}{ll} & \tau{\cdot}\tau_I(\partial_H(X(n_0, p_0[n_0], 0) \| \cdots \| X(n_k, p_0[n_k], 0))) \\ = & \tau{\cdot}\tau_I(Y(p_0, s_0)) \\ = & \tau{\cdot}Z(\mathsf{t}) \\ = & \tau{\cdot}\mathit{leader}{\cdot}\delta \end{array}$$

8 Graph algorithms

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

8.1 Minimisation modulo branching bisimulation

We sketch an algorithm by Groote and Vaandrager [44] to decide whether two finite-state processes are branching bisimilar (see Definition 4.5). 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 process graph containing finitely many states. As a preprocessing step, first we collapse all states in this process graph that are on a τ-loop 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 4.6. The τ-loops in a process graph can be detected using a well-known algorithm by Tarjan [71] for finding the *strongly connected components* in a process graph; see, e.g., [22]. Two states s and s' are in the same strongly connected component of a process graph if there exist execution sequences from s to s' and vice versa.

In order to explain the minimisation algorithm, we need to introduce some preliminary terminology. If P and P' are two set of states, then $s_0 \in \mathit{split}_\tau(P,P')$ if there exists a sequence of τ-transitions $s_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n$ for some $n \geq 0$ such that $s_i \in P$ for $i = 0, \ldots, n-1$ and $s_n \in P'$. Likewise, for $a \in \mathsf{Act}$, $s_0 \in \mathit{split}_a(P,P')$ 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. 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.

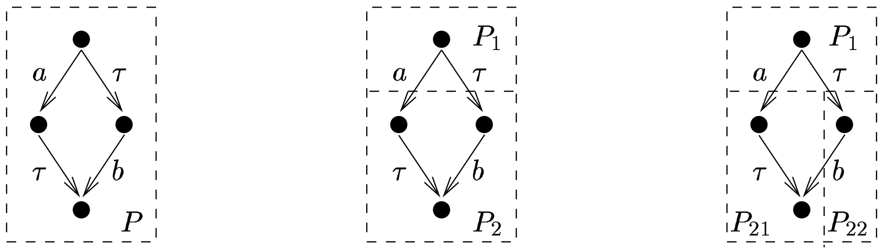
The minimisation algorithm works as follows. Suppose that at some point we have constructed a partition $P_1, \ldots, P_k$ of disjoint non-empty sets of states, where $P_1 \cup \cdots \cup P_k$ is the set of states in the input graph. (Remember that initially this partition consists of a single set.) If for some $i, j \in \{1, \ldots, k\}$ and $b \in \mathsf{Act} \cup \{\tau\}$ we have that $\emptyset \subset \mathit{split}_b(P_i, P_j) \subset P_i$ (where $\subset$ denotes strict set inclusion), then P_i is 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.

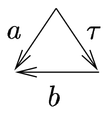
Example 8.1 We show how the minimisation algorithm minimises the process term $(a{\cdot}\tau + \tau{\cdot}b){\cdot}\delta$. Note that the process graph belonging to this term does not contain τ-loops. Initially, the set P contains all four states in this process graph.

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

Next, $split_b(P_2, P_2)$ 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 process graph:



Exercise 8.1 Apply the minimisation algorithm to the process graphs belonging to the following process terms and declarations:

(1) $(a + \tau{\cdot}b){\cdot}\delta$;

(2) $(a + \tau{\cdot}(a + b)){\cdot}\delta$;

(3) $(a{\cdot}\tau + \tau{\cdot}a){\cdot}\delta$;

(4) $(a{\cdot}b{\cdot}c + a{\cdot}b{\cdot}d){\cdot}\delta$;

(5) $a{\cdot}a{\cdot}a{\cdot}a{\cdot}\delta$;

(6) $X = \tau{\cdot}X + a{\cdot}Y$
$Y = a{\cdot}X$;

(7) $X = \tau{\cdot}X + a{\cdot}Y$
$Y = (a + b){\cdot}X$;

(8) $X = (\tau + a){\cdot}Y$
$Y = (a + b){\cdot}X$.

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. Their argumentation is briefly as follows. The algorithm to detect τ-loops (or better, to find strongly connected components) has worst-case time complexity $O(m)$. 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 $\emptyset \subset split_b(P_i, P_j) \subset P_i$; this method, which has been 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 $O(mn)$.

The algorithm of Groote and Vaandrager is an adaptation of a minimisation algorithm modulo bisimulation equivalence (see Definition 3.6) by Kanellakis and Smolka [50]. Paige and Tarjan [61] presented a more efficient algorithm to minimise a finite-state process graph modulo bisimilation equivalence, 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.

Exercise 8.2 Explain how the minimisation algorithm can be adapted to take into account the successful termination predicate $\xrightarrow{\tau} \surd$.

Weak bisimulation [57] 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 [50] 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 bisimilation 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.

8.2 Confluence

Milner [57, 59] was the first to recognise the usefulness of the notion of confluence (cf. Section 2.2) when analysing processes that involve the internal action τ. Here we use the following notion of confluence, which differs from the notion that was originally studied by Milner. Assume a process graph, together with a set T of τ-transitions in this process graph; 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 distinct transitions $s \xrightarrow{a} s'$ (with $a \in \mathsf{Act} \cup \{\tau\}$) and $s \Rightarrow s''$ in the process graph, the picture

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

can be completed in one of the following three fashions:

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

In the third case, $a = \tau$.

Definition 8.2 (Confluence) Assume a process graph L. A collection T of τ-transitions in L is *confluent* if for all transitions $s \overset{a}{\to} s' \in L$ and $s \overset{\tau}{\to} s'' \in T$:

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

(2) or $s'' \overset{a}{\to} s' \in L$;

(3) or $a = \tau$ and $s' \overset{\tau}{\to} s'' \in T$;

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

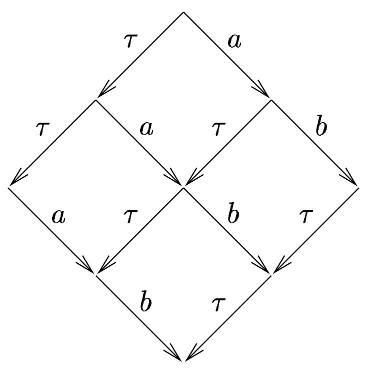
Groote and van de Pol [39] presented an efficient algorithm to compute, for a given finite-state process graph, the maximal set T of τ-transitions that is confluent with respect to this process graph. Initially it is assumed that all τ-transitions of the process graph are in T, and subsequently transitions 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 $s \overset{a}{\to} s'$ is taken from the stack, and for each $s \overset{\tau}{\to} s'' \in T$ it is verified whether at least one of the properties (1)-(4) in Definition 8.2 holds. If this is not the case, then $s \overset{\tau}{\to} s''$ is eliminated from T, and all transitions $s''' \overset{b}{\to} s$ that were previously eliminated from the stack are placed back on the stack. Namely, $s''' \overset{b}{\to} s$ and some transition $s''' \overset{\tau}{\to} s'''' \in T$ may previously have been found not to obstruct confluence due to the fact that $s \overset{\tau}{\to} s''$ 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 process graph L. Namely, if $s \overset{\tau}{\to} s' \in T$, 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 all transitions of the form $s \overset{a}{\to} s''$, except the transition $s \overset{\tau}{\to} s' \in T$ mentioned above, can be eliminated from L without influencing the branching bisimulation class of s. The next example shows that the absence of τ-loops in L is essential here.

Example 8.3 Consider the process graph defined by the process declaration $X = (\tau + a)\cdot X$; note that it contains a τ-loop. The τ-transition in this process graph is confluent. If the a-transition is eliminated from this process graph, then the resulting process graph 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 compression of process graphs.

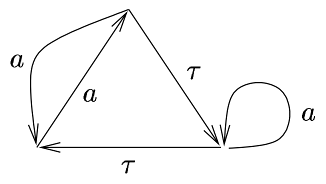
Example 8.4 Consider the process graph



The maximal confluent set of τ-transitions contains all six τ-transitions in this process graph. Compression with respect to this set produces the process graph belonging to $a{\cdot}b{\cdot}\delta$.

The next example shows that in general the maximal confluent set of τ-transitions is a proper subset of the collection of silent τ-transitions of a process graph.

Example 8.5 Consider the process graph



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

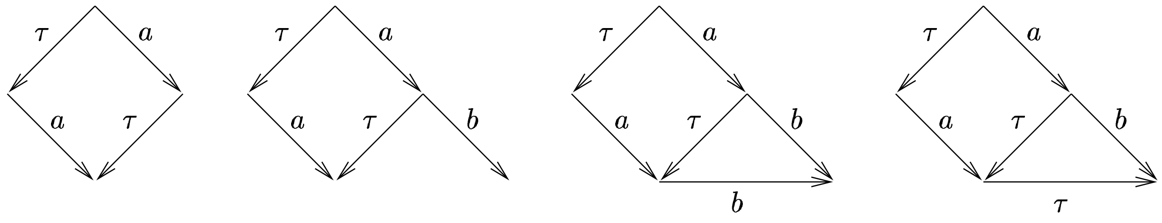
This process graph is minimal modulo bisimulation equivalence. So minimisation modulo branching bisimulation equivalence appears to be the only available method to reduce this graph.

Assume a finite-state process graph with m transitions. We recall from Section 8.1 that τ-loops can be eliminated from this process graph with worst-case time complexity $O(m)$. Groote and van de Pol [39] 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 finite number N such that each state has no more than N outgoing τ-transitions). Thus their algorithm performs better than the minimisation algorithm modulo branching bisimulation equivalence; see Section 8.1. 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 tool set.

Finally, we note that after compression of a process graph on the basis of its maximal confluent set of τ-transitions, the resulting process graph may again contain confluent τ-transitions. An example of this phenomenon is given below. 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 process graph has become empty.

Example 8.6 Consider the process graph that belongs to $(\tau{\cdot}a{\cdot}\tau + a){\cdot}\delta$. The first compression with respect to the maximal confluent set of τ-transitions produces the process graph belonging to $(\tau{\cdot}a + a){\cdot}\delta$. The second compression produces the process graph belonging to $(a + a){\cdot}\delta$.

Exercise 8.3 Give the maximal confluent sets of τ-transitions of the following four process graphs:



8.3 Model checking

Computation tree logic (CTL) [19] is a temporal logic to express properties of process graphs that do not carry actions on their transitions. ACTL [26] is an extension of CTL to process graphs with actions. ACTL consists of formulas on states, defined by the following BNF grammar:

$$\phi \ ::= \ \mathtt{T} \ | \ \neg\phi \ | \ \phi \wedge \phi' \ | \ \mathtt{X}_a\,\phi \ | \ \phi\,\mathtt{U}\,\phi' \ | \ \mathtt{G}\,\phi$$

where a ranges over $\mathsf{Act} \cup \{\tau\}$. Here, $\mathtt{T}$ is the universal predicate that holds for all states.[2] As usual, $\neg$ denotes negation and $\wedge$ denotes conjunction. The intuition behind the remaining constructs is as follows.

- $\mathtt{X}_a\,\phi$ holds in a state s if $s \xrightarrow{a} s'$ where formula ϕ holds in state s';
- $\phi\,\mathtt{U}\,\phi'$ holds in 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;
- $\mathtt{G}\,\phi$ 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 process graph. 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 executsion sequence

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

$s_0 \overset{a_0}{\to} \cdots \overset{a_{\ell-1}}{\to} s_\ell$ where there is no transition $s_\ell \overset{b}{\to} s$. The states s_0 that satisfy an ACTL formula ϕ, denoted by $s_0 \models \phi$, are defined inductively as follows:

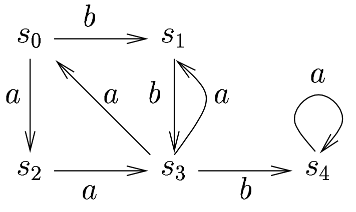
$$\begin{array}{ll}
s_0 \models \mathtt{T} & \\
s_0 \models \neg\phi & \text{if } s_0 \not\models \phi \\
s_0 \models \phi \wedge \phi' & \text{if } s_0 \models \phi \text{ and } s_0 \models \phi' \\
s_0 \models \mathtt{X}_a\,\phi & \text{if there is a state } s_1 \text{ with } s_0 \overset{a}{\to} s_1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

The verification whether a given formula holds for a certain state is known as *model checking*. Clarke, Emerson, and Sistla [20] presented an efficient model checking algorithm for CTL formulas, which extends to ACTL without any complications. Assume a finite-state process graph. 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.
- $\mathtt{X}_a\,\phi$ holds in each state that can perform an a-transition resulting in a state where ϕ holds.
- To compute the states in which $\phi\,\mathtt{U}\,\phi'$ 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 a path in which each state satisfies ϕ.
- To compute the states in which $\mathtt{G}\,\phi$ holds, first we eliminate all states from the input graph where ϕ does not hold. A state satisfies $\mathtt{G}\,\phi$ if and only if there is a path in the resulting process graph to a deadlock state, or to a strongly connected component that contains an infinite path.

Clarke, Emerson, and Sistla showed that their 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 algoritm [71] (cf. Section 8.1).

Exercise 8.4 Consider the process graph



$$\begin{tikzcd}
s_0 \arrow[r, "b"] \arrow[d, "a"'] & s_1 \arrow[d, "b"'] & \\
s_2 \arrow[r, "a"'] & s_3 \arrow[ul, "a"'] \arrow[u, bend right, "a"'] \arrow[r, "b"'] & s_4 \arrow[loop above, "a"]
\end{tikzcd}$$

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

(1) $\mathtt{X}_a\,\mathtt{X}_b\,\mathtt{T}$;

(2) $\mathtt{X}_b\,\mathtt{X}_a\,\mathtt{T}$;

(3) $\mathtt{G}\,\mathtt{X}_a\,\mathtt{T}$;

(4) $\mathtt{G}\,\mathtt{X}_b\,\mathtt{T}$;

(5) $\mathtt{G}\,(\mathtt{X}_a\,\mathtt{X}_b\,\mathtt{T} \vee \mathtt{X}_b\,\mathtt{X}_a\,\mathtt{T})$;

(6) $(\neg\mathtt{G}\,\mathtt{X}_a\,\mathtt{T} \vee \neg\mathtt{G}\,\mathtt{X}_b\,\mathtt{T})\,\mathtt{U}\,(\mathtt{X}_b\,\neg\mathtt{X}_b\,\mathtt{T})$.

Finally, for each state in the process graph, give an ACTL formula that is only satisfied by this state.

Linear temporal logic (LTL) [64] is an alternative temporal logic to express properties of process graphs that do not carry actions on their transitions. The model-checking algorithm for LTL [54] 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 problematic, because in general the size of a formula is small with respect to the size of the process graph against which it is checked. CTL is usually referred to as a *branching-time* temporal logic, because the construct $\mathtt{G}\,\phi$ constitutes an explicit quantification over full paths. LTL is referred to as a *linear-time* temporal logic, because formulas are interpreted over linear sequences of states. See [28] for a comparison of branching-time and linear-time temporal logics.

Solutions to the exercises

2.1 **map** $\wedge, \Rightarrow, \Leftrightarrow\colon Bool \times Bool \to Bool$
$\neg : Bool \to Bool$
$\dot{-}\colon Nat \times Nat \to Nat$
var $x{:}Bool$
$n, m{:}Nat$
rew $x \vee \mathsf{t} = \mathsf{t}$
$x \vee \mathsf{f} = x$
$\neg\, \mathsf{t} = \mathsf{f}$
$\neg\, \mathsf{f} = \mathsf{t}$
$\mathsf{t} \Rightarrow x = x$
$\mathsf{f} \Rightarrow x = \mathsf{t}$
$\mathsf{t} \Leftrightarrow x = x$
$\mathsf{f} \Leftrightarrow x = \neg x$
$0 \dot{-} n = 0$
$n \dot{-} 0 = n$
$S(n) \dot{-} S(m) = n \dot{-} m$

2.2 **map** $\geq\colon Nat \times Nat \to Bool$
$<\colon Nat \times Nat \to Bool$
$>\colon Nat \times Nat \to Bool$
var $n, m{:}Nat$
rew $n \geq 0 = \mathsf{t}$
$0 \geq S(n) = \mathsf{f}$
$S(n) \geq S(m) = n \geq m$
$0 < S(n) = \mathsf{t}$
$n < 0 = \mathsf{f}$
$S(n) < S(m) = n < m$
$S(n) > 0 = \mathsf{t}$
$0 > n = \mathsf{f}$
$S(n) > S(m) = n > m$

2.3 **map** $even : Nat \to Bool$
var $m, n{:}Nat$
rew $even(0) = \mathsf{t}$
$even(S(n)) = \neg even(n)$
$power(m, 0) = S(0)$
$power(m, S(n)) = mul(power(m, n), m)$

2.4 **sort** $Bool,\ List,\ D$
func $[] :\to List$
$in : D \times List \to List$
map $toe : List \to D$
$untoe : List \to List$
$isempty : List \to Bool$
$++ : List \times List \to List$
var $d, e{:}D,\ q, q'{:}List$
rew $toe(in(d, [])) = d$
$toe(in(d, in(e, q))) = toe(in(e, q))$
$untoe(in(d, [])) = []$
$untoe(in(d, in(e, q))) = in(d, untoe(in(e, q)))$
$isempty([]) = \mathsf{t}$
$isempty(in(d, q)) = \mathsf{f}$
$++([], q) = q$
$++(in(d, q), q') = in(d, ++(q, q'))$

2.5 **var** $d, e{:}D$
$q, q'{:}\mathit{List}$
rew $\mathit{eq}([], []) = \mathsf{t}$
$\mathit{eq}(\mathit{in}(d, q), []) = \mathsf{f}$
$\mathit{eq}([], \mathit{in}(d, q)) = \mathsf{f}$
$\mathit{eq}(\mathit{in}(d, q), \mathit{in}(e, q')) = \mathit{eq}(d, e) \wedge \mathit{eq}(q, q')$

2.6 If $\mathit{eq}(d,e) = \mathsf{t}$, then $d = \mathit{if}(\mathsf{t}, d, e) = \mathit{if}(\mathit{eq}(d,e), d, e) = e$.
If $d = e$, then $\mathit{eq}(d,e) = \mathit{eq}(d,d) = \mathsf{t}$.

2.7 $\mathsf{t} \vee \mathsf{t} = \mathsf{t}$ and $\mathsf{f} \vee \mathsf{t} = \mathsf{t}$.
$\neg\neg\mathsf{t} = \neg\mathsf{f} = \mathsf{t}$ and $\neg\neg\mathsf{f} = \neg\mathsf{t} = \mathsf{f}$.

2.8

(1) $\mathsf{f} \wedge \mathsf{t} = \mathsf{f}$.
$\mathsf{f} \wedge \mathsf{f} = \mathsf{f}$.

(2) $\mathsf{t} \Rightarrow \mathsf{t} = \mathsf{t}$.
$\mathsf{f} \Rightarrow \mathsf{t} = \mathsf{t}$.

(3) $\mathsf{t} \Rightarrow \mathsf{f} = \mathsf{f} = \neg\mathsf{t}$.
$\mathsf{f} \Rightarrow \mathsf{f} = \mathsf{t} = \neg\mathsf{f}$.

(4) $\mathsf{t} \Rightarrow y = y = \neg\neg y = \neg y \Rightarrow \mathsf{f} = \neg y \Rightarrow \neg\mathsf{t}$.
$\mathsf{f} \Rightarrow y = \mathsf{t} = \neg y \Rightarrow \mathsf{t} = \neg y \Rightarrow \neg\mathsf{f}$.

(5) $\mathsf{t} \Leftrightarrow \mathsf{t} = \mathsf{t}$.
$\mathsf{f} \Leftrightarrow \mathsf{t} = \neg\mathsf{t} = \mathsf{f}$.

(6) $\mathsf{t} \Leftrightarrow \mathsf{t} = \mathsf{t}$.
$\mathsf{f} \Leftrightarrow \mathsf{f} = \neg\mathsf{f} = \mathsf{t}$.

(7) $\mathsf{t} \Leftrightarrow \neg y = \neg y = \neg(\mathsf{t} \Leftrightarrow y)$.
$\mathsf{f} \Leftrightarrow \neg y = \neg\neg y = \neg(\mathsf{f} \Leftrightarrow y)$.

(8) $(x \vee \mathsf{t}) \Leftrightarrow x = \mathsf{t} \Leftrightarrow x = x = x \vee \mathsf{f} = x \vee \neg\mathsf{t}$.
$(x \vee \mathsf{f}) \Leftrightarrow x = x \Leftrightarrow x = \mathsf{t} = x \vee \mathsf{t} = x \vee \neg\mathsf{f}$.

(9) $\mathit{even}(\mathit{plus}(m, 0)) = \mathit{even}(m) = \mathit{even}(m) \Leftrightarrow \mathsf{t} = \mathit{even}(m) \Leftrightarrow \mathit{even}(0)$.
$\mathit{even}(\mathit{plus}(m, S(n))) = \mathit{even}(S(\mathit{plus}(m,n))) = \neg\mathit{even}(\mathit{plus}(m,n)) = \neg(\mathit{even}(m) \Leftrightarrow \mathit{even}(n)) = \mathit{even}(m) \Leftrightarrow \neg\mathit{even}(n) = \mathit{even}(m) \Leftrightarrow \mathit{even}(S(n))$.

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

2.9

(1) $\mathit{mul}(0, 0) = 0$.
$\mathit{mul}(0, S(n)) = \mathit{plus}(\mathit{mul}(0,n), 0) = \mathit{mul}(0,n) = 0$.

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

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

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

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

2.10 $+\!\!+(\mathit{untoe}(q),\mathit{in}(\mathit{toe}(q),q'))$.
Base case: $+\!\!+(\mathit{untoe}(\mathit{in}(d,[])),\mathit{in}(\mathit{toe}(\mathit{in}(d,[])),q')) = +\!\!+([],\mathit{in}(d,q')) = \mathit{in}(d,q') = \mathit{in}(d,+\!\!+([],q')) = +\!\!+(\mathit{in}(d,[]),q')$.
Inductive case:

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

2.11 Suppose $[] = \mathit{in}(d,q)$. Then $\mathsf{t} = \mathit{isempty}([]) = \mathit{isempty}(\mathit{in}(d,q)) = \mathsf{f}$, contradicting axiom Bool1.

3.1 $a(d)\cdot(b(\mathit{stop},\mathsf{f}) + c)$
$a(d)\cdot b(\mathit{stop},\mathsf{f}) + a(d)\cdot c$.

3.2
(1) $((a+a)\cdot(b+b))\cdot(c+c) \overset{\mathrm{A3}}{=} (a\cdot(b+b))\cdot(c+c) \overset{\mathrm{A3}}{=} (a\cdot b)\cdot(c+c) \overset{\mathrm{A3}}{=} (a\cdot b)\cdot c \overset{\mathrm{A5}}{=} a\cdot(b\cdot c)$;

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

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

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: $p \overset{(2)}{=} q+p \overset{\mathrm{A1}}{=} p+q \overset{(1)}{=} q$.

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

3.5 $a \parallel (b+c) \overset{\mathrm{CM1}}{=} (a \mathbin{\lfloor\!\lfloor} (b+c) + (b+c) \mathbin{\lfloor\!\lfloor} a) + a \mid (b+c) \overset{\mathrm{CM9}}{=} (a \mathbin{\lfloor\!\lfloor} (b+c) + (b+c) \mathbin{\lfloor\!\lfloor} a) + (a \mid b + a \mid c) \overset{\mathrm{CF}}{=} (a \mathbin{\lfloor\!\lfloor} (b+c)+(b+c) \mathbin{\lfloor\!\lfloor} a)+(b'+c') \overset{\mathrm{CF}}{=} (a \mathbin{\lfloor\!\lfloor} (b+c)+(b+c) \mathbin{\lfloor\!\lfloor} a)+(b\mid a+c\mid a) \overset{\mathrm{CM8}}{=} (a \mathbin{\lfloor\!\lfloor} (b+c)+(b+c) \mathbin{\lfloor\!\lfloor} a)+(b+c)\mid a \overset{\mathrm{A1}}{=} ((b+c) \mathbin{\lfloor\!\lfloor} a + a \mathbin{\lfloor\!\lfloor} (b+c)) + (b+c)\mid a \overset{\mathrm{CM1}}{=} (b+c) \parallel a$.

3.6 $(b\cdot a) \parallel a \overset{\mathrm{CM1}}{=} ((b\cdot a) \mathbin{\lfloor\!\lfloor} a+a \mathbin{\lfloor\!\lfloor} (b\cdot a))+(b\cdot a)\mid a \overset{\mathrm{CM2,3,5}}{=} (b\cdot(a \parallel a)+a\cdot(b\cdot a))+(b\mid a)\cdot a \overset{\mathrm{CM1}}{=} (b\cdot((a \mathbin{\lfloor\!\lfloor} a+a \mathbin{\lfloor\!\lfloor} a)+a\mid a)+a\cdot(b\cdot a))+(b\mid a)\cdot a \overset{\mathrm{CM2,CF'}}{=} (b\cdot((a\cdot a+a\cdot a)+\delta)+a\cdot(b\cdot a))+(b\mid a)\cdot a \overset{\mathrm{A3,6}}{=} (b\cdot(a\cdot a)+a\cdot(b\cdot a))+(b\mid a)\cdot a \overset{\mathrm{A4,5}}{=} (b\cdot a+a\cdot b)\cdot a+(b\mid a)\cdot a \overset{\mathrm{CM2}}{=} (b \mathbin{\lfloor\!\lfloor} a + a \mathbin{\lfloor\!\lfloor} b)\cdot a + (b\mid a)\cdot a \overset{\mathrm{A4}}{=} ((b \mathbin{\lfloor\!\lfloor} a + a \mathbin{\lfloor\!\lfloor} b) + b\mid a)\cdot a \overset{\mathrm{CM1}}{=} (b \parallel a)\cdot a$.

3.7

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

3.8 *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 $\{s(0), s(1), r(0), r(1)\}$.

$$\begin{array}{cl}
& (s(0)+s(1)) \parallel (r(0)+r(1)) \\
\overset{\text{CM1}}{=} & (s(0)+s(1)) \mathbin{⫫} (r(0)+r(1)) + (r(0)+r(1)) \mathbin{⫫} (s(0)+s(1)) \\
& + \ (s(0)+s(1)) \,|\, (r(0)+r(1)) \\
\overset{\text{CM4,CM8,9}}{=} & s(0) \mathbin{⫫} (r(0)+r(1)) + s(1) \mathbin{⫫} (r(0)+r(1)) + r(0) \mathbin{⫫} (s(0)+s(1)) \\
& + \ r(1) \mathbin{⫫} (s(0)+s(1)) + s(0)\,|\,r(0) + s(0)\,|\,r(1) + s(1)\,|\,r(0) \\
& + \ s(1)\,|\,r(1) \\
\overset{\text{CM2,CF,CF}'}{=} & s(0)(r(0)+r(1)) + s(1)(r(0)+r(1)) + r(0)(s(0)+s(1)) \\
& + \ r(1)(s(0)+s(1)) + c(0) + \delta + \delta + c(1) \\
\overset{\text{A6}}{=} & s(0)(r(0)+r(1)) + s(1)(r(0)+r(1)) + r(0)(s(0)+s(1)) \\
& + \ r(1)(s(0)+s(1)) + c(0) + c(1).
\end{array}$$

Hence,

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

3.9 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.

3.10 $\delta = p + q \overset{\text{A3}}{=} (p+q)+(p+q) \overset{\text{A1,2}}{=} p + (p + (q+q)) \overset{\text{A3}}{=} p + (p+q) = p + \delta \overset{\text{A6}}{=} p.$

3.11 yes; no; yes; yes; no.

3.12

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

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

3.14 (1) $x \lhd \mathsf{t} \rhd y = x = x + \delta = x \lhd \mathsf{t} \rhd \delta + y \lhd \mathsf{f} \rhd \delta = x \lhd \mathsf{t} \rhd \delta + y \lhd \neg\mathsf{t} \rhd \delta$;
$x \lhd \mathsf{f} \rhd y = y = \delta + y = x \lhd \mathsf{f} \rhd \delta + y \lhd \mathsf{t} \rhd \delta = x \lhd \mathsf{f} \rhd \delta + y \lhd \neg\mathsf{f} \rhd \delta$;

(2) $x \lhd \mathsf{t} \vee \mathsf{t} \rhd \delta = x \lhd \mathsf{t} \rhd \delta = x \lhd \mathsf{t} \rhd \delta + x \lhd \mathsf{t} \rhd \delta$;
$x \lhd \mathsf{t} \vee \mathsf{f} \rhd \delta = x \lhd \mathsf{t} \rhd \delta = x \lhd \mathsf{t} \rhd \delta + \delta = x \lhd \mathsf{t} \rhd \delta + x \lhd \mathsf{f} \rhd \delta$;
$x \lhd \mathsf{f} \vee \mathsf{t} \rhd \delta = x \lhd \mathsf{t} \rhd \delta = \delta + x \lhd \mathsf{t} \rhd \delta = x \lhd \mathsf{f} \rhd \delta + x \lhd \mathsf{t} \rhd \delta$;
$x \lhd \mathsf{f} \vee \mathsf{f} \rhd \delta = x \lhd \mathsf{f} \rhd \delta = x \lhd \mathsf{f} \rhd \delta + x \lhd \mathsf{f} \rhd \delta$;

(3) if $b = \mathsf{t}$, then by the assumption $(b = \mathsf{t} \Rightarrow x = y)$ we have $x = y$ and so $x \lhd b \rhd z = y \lhd b \rhd z$;
if $b = \mathsf{f}$, then $x \lhd b \rhd z = z = y \lhd b \rhd z$.

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

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

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

3.16 $0 \leq S(S(0)) = \mathsf{t}$;
$S(0) \leq S(S(0)) = 0 \leq S(0) = \mathsf{t}$;
$S(S(0)) \leq S(S(0)) = S(0) \leq S(0) = 0 \leq 0 = \mathsf{t}$;
$S(S(S(n))) \leq S(S(0)) = S(S(n)) \leq S(0) = S(n) \leq 0 = \mathsf{f}$.

3.17 ($\supseteq$) By SUM3

$$\sum_{b:Bool} x \triangleleft b \triangleright y = \sum_{b:Bool} x \triangleleft b \triangleright y + x \triangleleft \mathsf{t} \triangleright y + x \triangleleft \mathsf{f} \triangleright y = \sum_{b:Bool} x \triangleleft b \triangleright y + x + y.$$

So $x + y \subseteq \sum_{b:Bool} x \triangleleft b \triangleright y$.
($\subseteq$) $x \triangleleft \mathsf{t} \triangleright y = x \subseteq x + y$ and $x \triangleleft \mathsf{f} \triangleright y = y \subseteq x + y$, so by induction on booleans $x \triangleleft b \triangleright y = x \subseteq x + y$. Then by SUM11, SUM4 and SUM1 $\sum_{b:Bool} x \triangleleft b \triangleright y \subseteq x + y$.

3.18 ($\subseteq$) By SUM3

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

So $x \subseteq \sum_{d:D} x \triangleleft b(d) \triangleright \delta$.
($\supseteq$) If $b(d) = \mathsf{t}$ then $x \triangleleft b(d) \triangleright \delta = x$, and if $b(d) = \mathsf{f}$ then $x \triangleleft b(d) \triangleright \delta = \delta \subseteq x$, so $x \triangleleft b(d) \triangleright \delta \subseteq x$ for all $d{:}D$. Then by SUM11, SUM4 and SUM1 $\sum_{d:D} x \triangleleft b(d) \triangleright \delta \subseteq x$.

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

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

3.20

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

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

4.1

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

4.2
```
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))
```

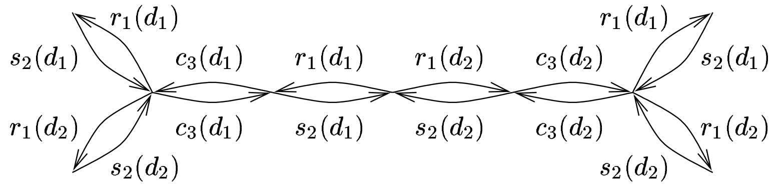
4.3
```
     Buf2 = rename({r1 -> r3, s3 -> s2},Buf1)
```

4.4 $a\,\mathcal{B}_1\,a{\cdot}\tau$, $\surd\,\mathcal{B}_1\,\tau$, 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 $a{\cdot}\tau\ \underline{\leftrightarrow}_b\ \tau{\cdot}a$.

4.5 $\tau{\cdot}(\tau{\cdot}(a+b)+b)+a\,\mathcal{B}\,a+b$, $\tau{\cdot}(a+b)+b\,\mathcal{B}\,a+b$, $a+b\,\mathcal{B}\,a+b$, and $\surd\,\mathcal{B}\,\surd$.

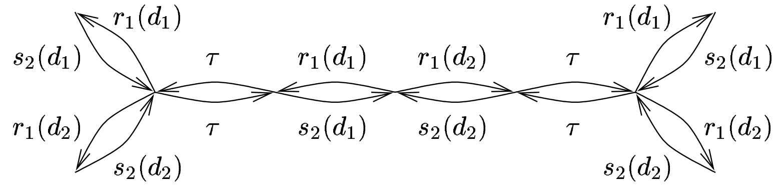
4.7 not branching bisimilar; bisimilar; branching bisimilar but not rooted branching bisimilar; rooted branching bisimilar but not bisimilar; not branching bisimilar.

4.8 The node in the middle is the root node.

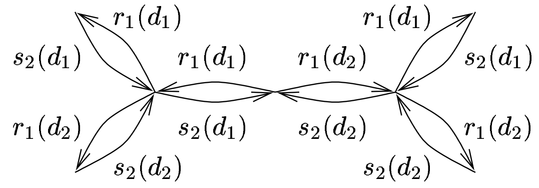


In the following execution trace, d_1 is read before d_2 via channel 1, while d_2 is sent before d_1 via channel 2: $r_1(d_1)\,c_3(d_1)\,r_1(d_2)\,s_2(d_2)\,c_3(d_1)\,s_2(d_1)$.

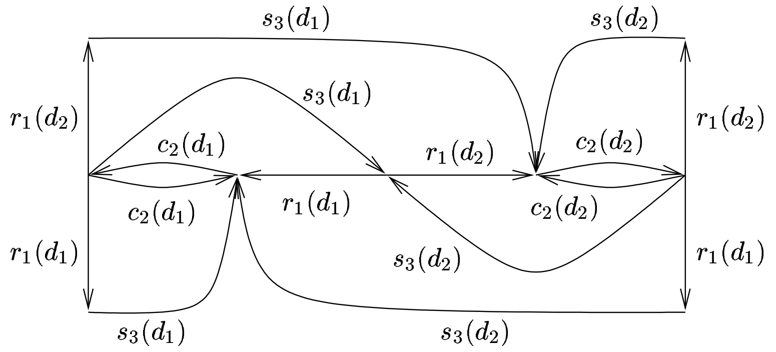
The two shortest execution traces of $\partial_{\{s_2(d_1)\}}(t)$ to a deadlock state are $r_1(d_1)\,c_3(d_1)\,r_1(d_1)$ and $r_1(d_2)\,c_3(d_2)\,r_1(d_1)$.



The four τ-transitions in the graph above are all silent, as they do not lose possible behaviours; namely, each τ-transition can be 'undone' by its reverse τ-transition.

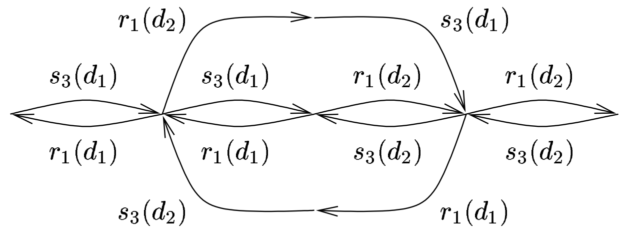


4.9 The node in the middle is the root node.



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 $r_1(d_1)\,c_2(d_1)\,r_1(d_2)$.



5.4

$$\begin{array}{lcl} \mathit{plusmod}(k,0) & = & k \\ \mathit{plusmod}(k,S(\ell)) & = & \mathit{succmod}(\mathit{plusmod}(k,\ell)) \\ \mathit{ordered}(k,\ell,m) & = & (k \leq \ell \wedge l < m) \vee (m < k \wedge k \leq \ell) \vee (\ell < m \wedge m < k) \end{array}$$

The algebraic formulation reads $\mathit{ordered}(k,\ell,\mathit{plusmod}(k,m))$.

5.6 Let $d, d', d'' \in \Delta$. An error trace is:

$$\textbf{(0)}\ r_{\mathrm{A}}(d)\, c_{\mathrm{B}}(d,0)\, j\, c_{\mathrm{C}}(d,0)\, s_{\mathrm{D}}(d)\ \textbf{(1)}\ r_{\mathrm{A}}(d')\, c_{\mathrm{B}}(d',S(0))\, j\, c_{\mathrm{C}}(d',S(0))\, s_{\mathrm{D}}(d')\ \textbf{(2)}\ r_{\mathrm{A}}(d)\, c_{\mathrm{B}}(d,0)\, j$$
$$c_{\mathrm{C}}(d,0)\ \textbf{(3)}\ c_{\mathrm{E}}(S(S(0)))\, j\, c_{\mathrm{F}}(S(S(0)))\ \textbf{(4)}\ r_{\mathrm{A}}(d'')\, c_{\mathrm{B}}(d'',S(S(0)))\, j\, c_{\mathrm{C}}(d'',S(S(0)))\, s_{\mathrm{D}}(d'')\ \textbf{(5)}\ s_{\mathrm{D}}(d).$$

(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 $[0, S(0)]$;

5.7
```
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,max-fill)) |> 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',max-fill)) |>
    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(next-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)
```

5.8
```
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,max-fill)) |> delta)
+ sum(k:Nat,sB(retrieve(k,buffer),k,next-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,
  <| ordered(first-in',k,plusmod(first-in',max-fill)) |>
    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
```

5.10 Each τ-transition loses the possibility to execute one of the *leader*(n)-actions at the end.

5.11 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.

6.1 The process declaration is captured by $Z(\mathrm{t})$, where

$$Z(b{:}Bool) = a{\cdot}Z(\neg b) \lhd b \rhd \delta + c{\cdot}Z(\neg b) \lhd \neg b \rhd \delta.$$

6.2

$$\begin{aligned}
&Y(n_1{:}Nat, \ldots, n_k{:}Nat) \ = \\
&\textstyle\sum_{i=1}^{k} a(f(n_i)){\cdot}Y(n_i := g(n_i)) \lhd h(n_i) \rhd \delta \\
&+\ \textstyle\sum_{j=1}^{k} b(f'(n_j)){\cdot}Y(n_j := g'(n_j)) \lhd h'(n_j) \rhd \delta \\
&+\ \textstyle\sum_{i,j=1}^{k} c(f(n_i)){\cdot}Y(n_i := f(n_i), n_j := f'(n_j)) \lhd h(n_i) \wedge h'(n_j) \wedge f(n_i) = f'(n_j) \wedge i \neq j \rhd \delta
\end{aligned}$$

6.3

$$\begin{array}{ll}
& X(n_1) \parallel X(n_2) \\
= & X(n_1) \mathbin{⌊\!⌊} X(n_2) + X(n_2) \mathbin{⌊\!⌊} X(n_1) + X(n_1) \mid X(n_2) \\
= & (a(n_1){\cdot}X(n_1+1) \lhd n_1 < 10 \rhd \delta + b(n_1){\cdot}X(n_1+2) \lhd n_1 > 5 \rhd \delta) \mathbin{⌊\!⌊} X(n_2) \\
+ & (a(n_2){\cdot}X(n_2+1) \lhd n_2 < 10 \rhd \delta + b(n_2){\cdot}X(n_2+2) \lhd n_2 > 5 \rhd \delta) \mathbin{⌊\!⌊} X(n_1) \\
+ & (a(n_1){\cdot}X(n_1+1) \lhd n_1 < 10 \rhd \delta + b(n_1){\cdot}X(n_1+2) \lhd n_1 > 5 \rhd \delta) \mid \\
& (a(n_2){\cdot}X(n_2+1) \lhd n_2 < 10 \rhd \delta + b(n_2){\cdot}X(n_2+2) \lhd n_2 > 5 \rhd \delta) \\
= & a(n_1){\cdot}(X(n_1+1) \parallel X(n_2)) \lhd n_1 < 10 \rhd \delta \;+\; b(n_1){\cdot}(X(n_1+2) \parallel X(n_2)) \lhd n_1 > 5 \rhd \delta \\
+ & a(n_2){\cdot}(X(n_1) \parallel X(n_2+1)) \lhd n_2 < 10 \rhd \delta \;+\; b(n_2){\cdot}(X(n_1) \parallel X(n_2+2)) \lhd n_2 > 5 \rhd \delta \\
+ & c(n_1){\cdot}(X(n_1) \parallel X(n_2)) \lhd n_1 < 10 \wedge n_2 > 5 \wedge n_1 = n_2 \rhd \delta \\
+ & c(n_1){\cdot}(X(n_1) \parallel X(n_2)) \lhd n_1 > 5 \wedge n_1 > 10 \wedge n_1 = n_2 \rhd \delta
\end{array}$$

6.4 We need to prove $\mathcal{I}_i(n) \Rightarrow \mathcal{I}_i(n+2)$ for $i = 1, 2$.
If n is even, then $\mathcal{I}_1(n) = \mathcal{I}_1(n+2) = \mathsf{t}$ and $\mathcal{I}_2(n) = \mathcal{I}_2(n+2) = \mathsf{f}$.
If n is odd, then $\mathcal{I}_1(n) = \mathcal{I}_1(n+2) = \mathsf{f}$ and $\mathcal{I}_2(n) = \mathcal{I}_2(n+2) = \mathsf{t}$.

6.5 We prove that $\mathcal{I}_i$ is an invariant for the LPE X in Definition 6.1; i.e., $\mathcal{I}_i(d) \wedge h(d, e_i) \Rightarrow \mathcal{I}_i(g(d, e_i))$, for $i = 1, 2$.
If $i = 1$, then we obtain $\mathsf{t} \wedge h(d, e_i) \Rightarrow \mathsf{t}$, which is true.
If $i = 2$, then we obtain $\mathsf{f} \wedge h(d, e_i) \Rightarrow \mathsf{f}$, which is true.

6.6 $\mathcal{I}(n) = \mathsf{t}$ for $n = 0, 2, 4, 6, 8$, and $\mathcal{I}(n) = \mathsf{f}$ for all other n.
$\mathcal{I}(n) = \mathsf{t}$ for $n = 0{-}6, 8$, and $\mathcal{I}(n) = \mathsf{f}$ for all other n.

6.7 $FC_X(n)$ is $\exists m{:}Nat(n = 3m \vee n = 3m+1)$.
$\phi(n) = \begin{cases} \mathsf{t} & \text{if } n = 3m \text{ or } n = 3m+2 \\ \mathsf{f} & \text{if } n = 3m+1 \end{cases}$
The matching criteria are fulfilled:

- $n = 3m+2 \Rightarrow \phi(n) = \phi(n+1) = \mathsf{t}$;
- $n = 3m \Rightarrow h'_a(\phi(n)) = \phi(n) = \mathsf{t}$;
  $n = 3m+1 \Rightarrow h'_d(\phi(n)) = \neg\phi(n) = \neg\mathsf{f} = \mathsf{t}$;
- $((n = 3m \vee n = 3m+1) \wedge \phi(n)) \Rightarrow n = 3m$;
  $((n = 3m \vee n = 3m+1) \wedge \neg\phi(n)) \Rightarrow n = 3m+1$;
- actions do not carry data parameters;
- $n = 3m \Rightarrow \phi(n+1) = \mathsf{f}$;
  $n = 3m+1 \Rightarrow \phi(n+1) = \mathsf{t}$.

7.2 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 $j \notin p[i]$.

7.3 By uniqueness of the root, $empty(p[busy(p, s)])$ implies $s[j] = 1$ for $j \neq busy(p, s)$. Hence,

$$\phi(p, s[busy(p, s)] := 1) = \mathsf{f}.$$

8.2 Assume a process graph. 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 $\emptyset \subset split_{\surd}(P) \subset P$.

8.3 The maximal confluent sets of the four process graphs are:
both τ-transitions;
$\emptyset$;
both τ-transitions;
$\emptyset$.

8.4 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 and s_2.

References

[1] P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: application to the bounded retransmission protocol. In W.R. Cleaveland, ed., *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] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[4] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54(1/2):70–120, 1982.

[5] H.P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics 103. North-Holland, 1984. Second edition.

[6] 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.

[7] T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.

[8] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. ACM/Addison Wesley, 1989.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] M.A. Bezem and A. Ponse. Two finite specifications of a queue. *Theoretical Computer Science*, 177(2):487–507, 1997.

[15] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[16] Efficient annotated terms M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. Technical Report SEN-R0003, CWI, 2000.

[17] 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.

[18] G. Bruns. *Distributed Systems Analysis with CCS.* Prentice Hall, 1997.

[19] 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.

[20] 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.

[21] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 2000.

[22] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[23] 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.

[24] H. Davenport. *The Higher Arithmetic.* Cambridge University Press, 1952.

[25] 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'2000)*, Limassol, Cyprus, LNCS. Springer-Verlag, To appear.

[26] 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.

[27] 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.

[28] 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.

[29] W.J. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2000.

[30] 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.

[31] 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.

[32] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[33] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[34] J.F. Groote. *Process Algebra and Structured Operational Semantics.* PhD thesis, University of Amsterdam, 1991.

[35] 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.

[36] J.F. Groote and I.A. van Langevelde. The SVC file format. Technical Report, CWI, 2000. To appear.

[37] 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.

[38] 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.

[39] 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'2000)*, Bratislava, Slovakia, LNCS 1893, pp. ????. Springer-Verlag, 2000.

[40] 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, *Proceedings of the International Workshop on Semantics of Specification Languages*, Utrecht, The Netherlands, Workshops in Computing, pp. 231–250. Springer-Verlag, 1993.

[41] 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.

[42] J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization of parallel pCRL. Technical Report SEN-R0019, CWI, 2000.

[43] J.F. Groote and J. Springintveld. Focus points and convergent process operators: a proof strategy for protocol verification. In A. Arnold, ed., *Models and Proofs, Proceedings AMAST Workshop on Real-Time Systems and Operation Inter-PRC*, Bordeaux, France, 1995.

[44] 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.

[45] J.F. Groote and J.J. van Wamel. The parallel composition of uniform processes with data. To appear in *Theoretical Computer Science*.

[46] 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.

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

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

[49] IEEE Computer Society. *IEEE Standard for a High Performance Serial Bus.* Std. 1394-1995, August 1996.

[50] 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.

[51] H.P. Korver. *Protocol Verification in μCRL.* PhD Thesis, University of Amsterdam, 1994.

[52] H.P. Korver and M.P.A. Sellink. Example verifications using alphabet axioms. *Formal Aspects of Computing*, 10(1):43–58, 1998.

[53] 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.

[54] 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.

[55] J. Loeckx, H.-D. Ehrich, and M. Wolf. *Specification of Abstract Data Types.* Wiley/Teubner, 1996.

[56] 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.

[57] R. Milner. *A Calculus of Communicating Systems.* LNCS 92, Springer-Verlag, 1980.

[58] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

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

[60] 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.

[61] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[62] 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.

[63] 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.

[64] 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.

[65] J.M.T. Romijn. A timed verification of the IEEE 1394 leader election protocol. In S. Gnesi and D. Latella, eds, *Proceedings 4th Workshop on Formal Methods for Industrial Critical Systems (FMICS'99)*, Trento, Italy, pp. 3–29. Full version to appear in *Formal Methods in System Design.*

[66] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[67] 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.

[68] 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.

[69] 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.

[70] A.S. Tanenbaum. *Computer Networks.* Prentice Hall, 1981.

[71] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[72] J.J. van Wamel. *Verification Techniques for Elementary Data Types and Retransmission Protocols*. PhD thesis, University of Amsterdam, 1995.

[73] A. Wouters. Manual for the μCRL toolset. Technical Report, CWI, 2000. To appear.