
Contents

1 Analysis of Distributed Systems with mCRL2	3
<i>Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav S. Usenko and Muck van Weerdenburg Technische Universiteit Eindhoven (TU/e)</i>	
1.1 Introduction	3
1.2 The Language of mCRL2	5
1.2.1 The mCRL2 process language	5
1.2.2 The mCRL2 data language	10
1.3 Verification Methodology and the mCRL2 Toolset	13
1.4 Examples of Modelling and Verification in mCRL2	16
1.4.1 Dining Philosophers Problem	16
1.4.2 Alternating bit protocol	18
1.4.3 Sliding Window Protocol (SWP)	22
1.5 Historical context	26
Bibliography	29
1. Use of captions in section/subsection headings is not consistent	
2. Where should we put the (high-level) comparison between (timed) μ CRL and mCRL2?	
3. The second paragraph already described most relevant changes in the process language and is therefore only a good introduction to mCRL2 for those that are acquainted with μ CRL	

Chapter 1

Analysis of Distributed Systems with mCRL2

Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav S. Usenko and Muck van Weerdenburg

Technische Universiteit Eindhoven (TU/e)

1.1	Introduction	3
1.2	The Language of mCRL2	4
1.3	Verification Methodology and the mCRL2 Toolset	13
1.4	Examples of Modelling and Verification in mCRL2	15
1.5	Historical context	26

1.1 Introduction

The language mCRL2 [?] is a process algebra with data and time suitable for the specification and verification of a broad range of (typically distributed) systems. It is the successor of the language μ CRL [?, ?] and its timed version timed μ CRL [?, ?]. Together with their toolsets, μ CRL and mCRL2 have been successfully used in various academic and industrial case studies [?, ?, ?, ?]. Specific characteristics of the mCRL2 process algebra are local communication, multiactions and communication-independent parallelism. Local communication allows one to specify communications within subcomponents of the system without affecting the rest of the system. This makes the language particularly suitable for component-based and hierarchical systems. Global communication, which is typically used in other languages, only allows communication to be defined on a global level for the system as a whole. Multiactions enable the specification of (not necessarily related) actions that are to be executed together. Most process algebras only allow a single action to be executed atomically thus forcing an order on the execution of actions. By allowing multiple actions to be executed together, certain systems (such as low-level hardware) can be modelled in a much more straightforward manner. These multiactions also allow the separation of parallelism and communication. When two actions can execute at the same time a multi-action with those actions is the result. Communication can then be applied to these multiactions to make certain actions communicate with each other. Besides the process algebra, mCRL2 also has a higher-order data language.

Processes and actions can be parameterised with this data, which is essential for the description of most practical systems. Besides several built-in data types (e.g. integers, lists, sets) the user can define its own abstract data types in a straightforward manner. Typical functions, such as equality, are automatically generated for such user-defined data types. As the data language is higher order, functions are first-class citizens and can therefore be used just as easily as other data. For practical use we have developed a toolset for mCRL2. This toolset contains numerous tools for manipulation of specifications, state-space exploration and generation, simulation, verification, and visualisation. To be able to work efficiently a specification is transformed into a specific form called a linear process. Such a linear process can be seen as a compact symbolic representation of the (possibly infinite) state space of the system. Many optimisation and verification techniques have been developed for these linear processes (see Sect. ??). A typical practice in verification via model checking is to generate a state space of a specification and use that state space to check certain properties. The mCRL2 toolset supports this practice, but also offers an alternative. Given a specification and a (modal) formula expressing a desired (temporal) property, the toolset can generate a Parameterised boolean Equation System (PBES) [?, ?]. Such a PBES can then be solved by specific provers of the toolset. The use of PBESs is particularly interesting for systems that are too large or complex for model checking via explicit state spaces.

The languages (timed) μ CRL and mCRL2 and their corresponding toolsets are very similar. Below we list the major differences. For a more detailed discussion on these differences and their motivation we refer to [?] and [?].

Process The specific characteristics of mCRL2 as mentioned in the introduction are basically also the most important changes with respect to (timed) μ CRL.

Data In the data specifications of mCRL2 the use of functions as first class citizens is now supported and libraries involving standard data types such as the Booleans and the natural numbers have been introduced.

Toolset The toolsets offer the same type of functionality. Some of the tools that are available for μ CRL are not yet available for mCRL2 and mostly the timing features are not incorporated (yet).

Any process description in (timed) μ CRL is straightforwardly translated into a description of the same process in mCRL2.

In Sect. ?? we give an introduction to mCRL2. We define the syntax of the mCRL2 process language together with an intuitive explanation of the semantics. In Sect. ?? we discuss the data language that is used by the process language. In Sect. ?? we give an in-depth overview of the toolset. We further illustrate the possibilities offered by the mCRL2 language and toolset by means of some examples in Sect. ?. In Sect. ?? concluding remarks are presented.

1.2 The Language of mCRL2

1.2.1 The mCRL2 process language

Multiactions

The primary notion in the mCRL2 process language is an *action*, which represents an elementary activity or a communication of some system. The following example illustrates how action names `send`, `receive` and `error` can be declared. Actions can be *parameterised* with data. For example:

```
act  error;
     send :  $\mathbb{B}$ ;
     receive :  $\mathbb{B} \times \mathbb{N}$ ;
```

This declares parameterless action name `error`, action name `send` with a data parameter of sort \mathbb{B} (booleans), and action name `receive` with two parameters of sort \mathbb{B} and \mathbb{N} (natural numbers), respectively. For the above action name declaration, `error`, `send(true)` and `receive(false, 6)` are valid actions. The means offered by mCRL2 to define sorts and operations on sorts will be discussed in Sect. ??.

In general, we write `a, b, ...` to denote action names and \vec{d}, \vec{e}, \dots to denote vectors of data parameters. In the notation $\mathbf{a}(\vec{d})$, we assume that the vector of data parameters \vec{d} is of the type that is specified for the action name `a`. An action without data parameters can be seen as an action with an empty vector of data parameters. We often write α, β, \dots for multiactions.

Actions are allowed to occur simultaneously in mCRL2. In this case we speak about multiactions. A multiaction is a bag of actions that are constructed according to the following BNF:

$$\alpha ::= \tau \mid \mathbf{a}(\vec{d}) \mid \alpha \mid \beta ,$$

where `a` denotes an action name and \vec{d} a vector of data parameters.

The term τ represents the multiaction containing no (observable) actions. This so-called hidden or internal action cannot be observed. It is not very useful when specifying the behaviour of processes, but it is essential when it comes to analysing processes. The term $\mathbf{a}(\vec{d})$ represents a multiaction that contains only (one occurrence of) the action $\mathbf{a}(\vec{d})$ and the term $\alpha \mid \beta$ represents a multiaction containing the actions from either the multiaction α or β .

As multiactions are bags we have the usual equalities on multiactions, viz. commutativity and associativity of \mid . Note that τ acts as the unit element of \mid . That is, $\alpha \mid \tau = \alpha = \tau \mid \alpha$. Using the actions declared previously the following are considered multiactions: τ , $\text{error} \mid \text{error} \mid \text{send}(\text{true})$, $\text{send}(\text{true}) \mid \text{receive}(\text{false}, 6)$ and $\tau \mid \text{error}$.

Basic operators

Process expressions, denoted by p, q, \dots , describe when certain multiactions can be executed. For example, “ a is followed by either b or c ”. We make this notion more formal by introducing operators. The most basic expressions are as follows:

- *Multiactions* (α, β , etc.) as described above.
- *Deadlock* or inaction δ , which does not execute any multiactions, but only displays delay behaviour.
- *Alternative composition*, written as $p + q$. This expression non-deterministically chooses to execute either p or q . The choice is made upon performance of the first multiaction of p or q .
- *Sequential composition*, written $p \cdot q$. This expression first executes p and upon termination of p continues with the execution of q .
- *Conditional operator*, written $c \rightarrow p \diamond q$, where c is a data expression of sort \mathbb{B} . This process expression behaves as an if-then-else construct: if c is *true* then p is executed, else q is executed. The else part is optional. This operator is used to express that data can influence process behaviour.
- *Process references*, written $P(\vec{d}), Q(\vec{d})$, etc. are used to refer to processes declared by *process definitions* of the form $P(\vec{x}:\vec{D}) = p$. This process definition declares that the behaviour of the process reference $P(\vec{d})$ is given by $p[\vec{d}/\vec{x}]$, i.e. p in which all free occurrences of variables \vec{x} are replaced by \vec{d} .
- *Summation operator*, written as $\sum_{x:D} p$, where x is a variable of sort D and p is a process expression in which this variable may occur. In this case we say that x is bound in p . The corresponding behaviour is a non-deterministic choice among the processes $p[d/x]$ for all terms d of sort D . If $\{d_0, d_1, \dots, d_n, \dots\}$ are the terms of sort D , then $\sum_{x:D} p$ can be expressed as $p[d_0/x] + p[d_1/x] + \dots + p[d_n/x] + \dots$.
- *At operator*, written $p \circ t$, where t is a data expression of sort \mathbb{R} (real numbers). The expression $p \circ t$ indicates that any first multiaction of p happens at time t (in case $t < 0$, $p \circ t$ is equal to $\delta \circ \mathbf{0}$, the latter representing an immediate deadlock).

When writing process expressions we usually omit parentheses as much as possible. To do this, we define precedence rules for the operators. The precedence of the operators introduced so far, in decreasing order, is as follows: $\circ, \cdot, \rightarrow, \sum, +$. Furthermore, \cdot and $+$ are associative. So, instead of writing $(a \cdot (b \cdot c)) + (d + e)$ we usually write $a \cdot b \cdot c + d + e$.

Often processes have some recursive behaviour. A coffee machine, for example, will normally not stop (terminate) after serving only one cup of coffee. To facilitate this, we use process references and process definitions:

```
act   coin, break, coffee;
proc  Wait = coin · Serve;
       Serve = break ·  $\delta$  + coffee · Wait;
```

This declares process references (often just called processes) `Wait` and `Serve`. Process `Wait` can do a `coin` action, after which it behaves as process `Serve`. Process `Serve` can do a `coffee` action and return to process `Wait`, but it might also do a `break` action, which results in a deadlock.

A complete process specification needs to have an *initial process*. For example:

```
init  Wait;
```

Parameterised processes can be declared as follows:

```
proc  P( $c:\mathbb{B}, n:\mathbb{N}$ ) = error · P( $c, n$ )
       + send( $c$ ) · P( $\neg c, n + 1$ )
       + receive( $c, n$ ) · P( $false, \max(n - 1, 0)$ );
```

This declares the processes $P(c, n)$ with data parameters c and n of sort \mathbb{B} and \mathbb{N} , respectively. Note that the sorts of the data parameters are declared on the left-hand side of the definition. In the process references on the right-hand side the *values* of the data parameters are specified.

Summation is used to *quantify* over data types. Summations over a data type are particularly useful to model the receipt of an arbitrary element of a data type. For example the following process is a description of a single-place buffer, repeatedly reading a natural number using action name r , and then delivering that value via action name s .

```
act    $r, s : \mathbb{N}$ ;
proc  Buffer =  $\sum_{n:\mathbb{N}} r(n) \cdot s(n) \cdot \text{Buffer}$ ;
init  Buffer;
```

Time can be added to processes using the operator ϵ . We give a few examples of the use of the operator ϵ . To start with, we specify a simple clock:

```
act   tick;
proc  C( $t : \mathbb{R}$ ) = tick  $\epsilon$  ( $t + 1$ ) · C( $t + 1$ );
init  C(0);
```

For a positive value u of sort \mathbb{R} , the process $C(u)$ exhibits the single infinite trace $\text{tick } \epsilon (u + 1) \cdot \text{tick } \epsilon (u + 2) \cdot \text{tick } \epsilon (u + 3) \cdot \dots$.

As a different example, we show a model of a *drifting* clock (taken from [?]). This is a clock that is accurate within a bounded interval $[1 - \mathfrak{d}, 1 + \mathfrak{d}]$, where $\mathfrak{d} < 1$.

```

proc DC( $t : \mathbb{R}$ ) =  $\sum_{\epsilon : \mathbb{R}} (1 - \mathfrak{d} \leq \epsilon \wedge \epsilon \leq 1 + \mathfrak{d}) \rightarrow \text{tick}^c(t + \epsilon) \cdot \text{DC}(t + \epsilon)$ ;
init DC(0);

```

Parallel composition

Besides the basic operators, which are typically used to specify the behaviour of core components in the system, we also have *parallel composition* (or *merge*) to compose processes. We write $p \parallel q$ for the parallel composition of p and q . This means that the multiactions of p and q are *interleaved* and *synchronised*. The parallel composition binds stronger than the summation operator, but weaker than the conditional operator.

To illustrate, $a \parallel b$ will either first execute a followed by b , or first b followed by a , or a and b together. That is, $p \parallel q$ is equivalent to $a \cdot b + b \cdot a + a \mid b$. The process $a \cdot b \parallel (c + d)$ is equivalent to $a \cdot (b \cdot (c + d) + c \cdot b + d \cdot b + b \mid c + b \mid d) + c \cdot a \cdot b + d \cdot a \cdot b + a \mid c \cdot b + a \mid d \cdot b$.

Additional operators

Now that we are able to put various processes in parallel, we need ways to restrict the behaviour of this composition and to model the interaction between processes. For this purpose we introduce the following operators:

- *Restriction operator* $\nabla_V(p)$ (also known as *allow*), where V is a set consisting of (non-empty) multisets of action names specifying exactly which multiactions from p are allowed to occur. Restriction $\nabla_V(p)$ disregards the data parameters of the multiactions in p when determining if a multiaction should be allowed, e.g. $\nabla_{\{b \mid c\}}(a(0) + b(\text{true}, 5) \mid c) = b(\text{true}, 5) \mid c$.

Note that the empty multiaction τ is not allowed as an element of the set V , so τ is always allowed (for any V , $\nabla_V(\tau) = \tau$).

- *Blocking operator* $\partial_B(p)$ (also known as *encapsulation*), where B is a set of action names (does not include τ) that are *not* allowed to occur. Blocking $\partial_B(p)$ disregards the data parameters of the actions in p when determining if an action should be blocked, e.g. $\partial_{\{b\}}(a(0) + b(\text{true}, 5) \mid c) = a(0)$.
- *Renaming operator* $\rho_R(p)$, where R is a set of renamings of the form $a \rightarrow b$, meaning that every occurrence of action name a in p is replaced by action name b . This set R is required to be a function. Renaming $\rho_R(p)$ also disregards the data parameters, but when a renaming is applied the data parameters are retained, e.g. $\rho_{\{a \rightarrow b\}}(a(0) \mid b + a \mid c) = b(0) \mid b + b \mid c$.

Note that every action name may only occur once as a left-hand side of \rightarrow in R .

- *Communication operator* $\Gamma_C(p)$, where C is a set of allowed communications of the form $\mathbf{a}_0 \mid \cdots \mid \mathbf{a}_n \rightarrow \mathbf{c}$, with $n \geq 1$ and \mathbf{a}_i and \mathbf{c} action names. For each communication $\mathbf{a}_0 \mid \cdots \mid \mathbf{a}_n \rightarrow \mathbf{c}$, multiactions containing $\mathbf{a}_0(\vec{d}) \mid \cdots \mid \mathbf{a}_n(\vec{d})$ (for some \vec{d}) in p are replaced by $\mathbf{c}(\vec{d})$. Note that the data parameters are retained in action \mathbf{c} . For example, $\Gamma_{\{\mathbf{a} \mid \mathbf{b} \rightarrow \mathbf{c}\}}(\mathbf{a}(0) \mid \mathbf{b}(0)) = \mathbf{c}(0)$, but also $\Gamma_{\{\mathbf{a} \mid \mathbf{b} \rightarrow \mathbf{c}\}}(\mathbf{a}(0) \mid \mathbf{b}(1)) = \mathbf{a}(0) \mid \mathbf{b}(1)$. Furthermore, $\Gamma_{\{\mathbf{a} \mid \mathbf{b} \rightarrow \mathbf{c}\}}(\mathbf{a}(1) \mid \mathbf{a}(0) \mid \mathbf{b}(1)) = \mathbf{a}(0) \mid \mathbf{c}(1)$.

The left-hand sides of the communications in C should be disjoint (e.g. $C = \{\mathbf{a} \mid \mathbf{b} \rightarrow \mathbf{c}, \mathbf{a} \mid \mathbf{d} \rightarrow \mathbf{e}\}$ is not allowed) to ensure that the communication operator uniquely maps each multiaction to another.

With these auxiliary functions we can, for example, enforce communication between processes. A typical example is as follows.

```
act   send, read, c;
init   $\nabla_{\{\mathbf{c}\}}(\Gamma_{\{\text{send} \mid \text{read} \rightarrow \mathbf{c}\}}(\text{send} \parallel \text{read}));$ 
```

Here we have that $\text{send} \parallel \text{read}$ is equivalent to $\text{send} \cdot \text{read} + \text{read} \cdot \text{send} + \text{send} \mid \text{read}$. In the latter the communication operator replaces the $\text{send} \mid \text{read}$ by \mathbf{c} giving $\text{send} \cdot \text{read} + \text{read} \cdot \text{send} + \mathbf{c}$. With the restriction operator we allow only \mathbf{c} of these alternatives to execute. That is, the above process is equivalent to just \mathbf{c} .

Passing a data value from one process to another can also be achieved with the communication and restriction operations.

$$\nabla_{\{\mathbf{c}\}} \left(\Gamma_{\{\mathbf{s} \mid \mathbf{r} \rightarrow \mathbf{c}\}} \left(\mathbf{s}(3) \cdot x \parallel \sum_{n:\mathbb{N}} \mathbf{r}(n) \cdot \mathbf{Y}(n) \right) \right) = \mathbf{c}(3) \cdot \nabla_{\{\mathbf{c}\}}(\Gamma_{\{\mathbf{s} \mid \mathbf{r} \rightarrow \mathbf{c}\}}(x \parallel \mathbf{Y}(3)))$$

In this case the value 3 is passed from one parallel component to the other to become the value of the variable n in process \mathbf{Y} .

Abstraction

An important notion in process algebra is that of *abstraction*. Usually the requirements of a system are defined in terms of *external* behaviour (i.e. the interactions of the system with its environment), while one wishes to check these requirements on an implementation of the system which also contains *internal* behaviour (i.e. the interactions between the components of the system). So it is desirable to be able to abstract from the internal behaviour of the implementation. For this purpose the following constructs are available:

- *Internal action* or silent step τ , which is a special multiaction (i.e. the empty one) that denotes that some (unknown) internal behaviour happens.
- *Hiding operator* $\tau_I(p)$, which hides (or renames to τ) all actions with an action name in I in all multiactions in p . Hiding $\tau_I(p)$ disregards the data parameters of the actions in p when determining if an action should be hidden.

Assume we have the two simple processes P and Q that are to be run in parallel. Furthermore we wish that P executes an a and Q a b and that the a is followed by b . We could then write the following specification.

```

act   a, b, r, s, i;
proc  P = a · s;
        Q = r · b;
init   $\nabla_{\{a,b,i\}}(\Gamma_{\{r|s \rightarrow i\}}(P \parallel Q));$ 

```

The system described by this specification is equivalent to the process $a \cdot i \cdot b$. That is, first P executes its a action, then P and Q synchronise (producing i as a result) such that Q can execute its b action.

As actions r , s and i are only used to synchronise P and Q , we are not really interested to see them in the resulting process. Instead, we can hide the result of this synchronisation as in the following process:

```

init   $\nabla_{\{a,b\}}(\tau_{\{i\}}(\Gamma_{\{a|b \rightarrow i\}}(P \parallel Q)));$ 

```

This process is equivalent to $a \cdot \tau \cdot b$. Under certain equivalences (see [?, ?]) such as weak and branching bisimilarity, this process is also equivalent to $a \cdot b$ (as the τ is unobservable).

1.2.2 The mCRL2 data language

The mCRL2 data language is a functional language based on *higher-order abstract data types* [?, ?, ?]. As mentioned before, mCRL2 also has concrete data types: *standard data types* and sorts constructed from a number of *type formers*.

Basic data type definition mechanism

Basically, mCRL2 contains a simple and straightforward data type definition mechanism. Sorts (types), constructor functions, maps (functions) and their definitions can be declared. Sorts declared in such a way are called user-defined sorts. For instance, the following declares the sort A with constructor functions c and d . Also functions f and g , and constant h , are declared and (partially) defined:

```

sort   A;
cons  c, d : A;
map   f : A × A → A;
        g : A → A;
        h : A;
var   x : A;
eqn   f(c, x) = c;
        f(d, x) = x;
        g(c) = c;
        h = d;

```

In the equations *variables* are used to represent arbitrary data expressions. A sort is called a *constructor sort* when it has at least one constructor function. For example, A is a constructor sort. Constructor sorts correspond to inductive data types.

Standard data types

mCRL2 has the following standard data types:

- Booleans (\mathbb{B}) with constructor functions *true* and *false* and operators \neg , \wedge , \vee , and \Rightarrow . It is assumed that *true* and *false* are different.
- Unbounded positive integers (\mathbb{N}^+), natural numbers (\mathbb{N}), integers (\mathbb{Z}), and real numbers (\mathbb{R}) with relational operators $<$, \leq , $>$, \geq , unary negation $-$, binary arithmetic operators $+$, $-$, $*$, **div**, **mod** and arithmetic operations *max*, *min*, *abs*, *succ*, *pred*, *exp*. These functions are only available for appropriate sorts, e.g. **div** and **mod** are only defined for a denominator of sort \mathbb{N}^+ . Also conversion functions $A \wr B$ are provided for all sorts $A, B \in \{\mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}\}$.

The user of the language is allowed to add maps and equations for standard data types. This also enables the user to specify *inconsistent* theories where the equation *true* = *false* becomes derivable. In such a case, the data specification loses its meaning.

Type formers

There are a number of operators to construct types, such as structured types, function types, and for lists, sets and bags.

A *structured type* represents a sort together with constructor, projection, and recogniser functions in a compact way. For instance, a sort of machine states can be declared by:

```

struct off | standby | starting | running(mode :  $\mathbb{N}$ ) | broken?is_broken;

```

This declares a sort with constructor functions *off*, *standby*, *starting*, *running* and *broken*, projection function *mode* from the declared sort to \mathbb{N} and recogniser *is_broken* from this sort to \mathbb{B} . So $mode(running(n)) = n$ and $mode(c)$ is

left unspecified for all constructors c different from *running*; so $mode(c)$ is a natural number, but we don't know which one. Also, $is_broken(broken) = true$ and $is_broken(d) = false$ for all constructors d different from *broken*.

Second, we have the *function* type former. The sort of functions from A to B is denoted $A \rightarrow B$. Note that function types are first-class citizens: functions may return functions. It is assumed that parentheses associate to the right in function notations, e.g. $A \rightarrow B \rightarrow C$ means $A \rightarrow (B \rightarrow C)$.

We also have a *list* type former. The sort of (finite) lists containing elements of sort A is declared by $List(A)$ and has constructor functions $[] : List(A)$ and $\triangleright : A \times List(A) \rightarrow List(A)$. Other operators include \triangleleft , $++$ (concatenation), $.$ (element at), *head*, *tail*, *rhead* and *rtail* together with list enumeration $[e_0, \dots, e_n]$. The following expressions of type $List(A)$ are all equivalent: $[c, d, d]$, $c \triangleright [d, d]$, $[c, d] \triangleleft d$ and $[] ++ [c, d] ++ [d]$.

Possibly infinite *sets* and *bags* where all elements are of sort A are denoted by $Set(A)$ and $Bag(A)$, respectively. The following operations are provided for these sort expressions: set enumeration $\{d_0, \dots, d_n\}$, bag enumeration $\{d_0 : c_0, \dots, d_n : c_n\}$ (c_i is the multiplicity or count of element d_i), set/bag comprehension $\{x : s \mid c\}$, element test \in , bag multiplicity *count*, set complement \bar{s} and infix operators \subseteq , \subset , \cup , $-$, \cap with their usual meaning for sets and bags. Also conversion functions *Set2Bag* and *Bag2Set* are provided, where the latter one 'forgets' the multiplicity of the bag elements.

Sort references

Sort references can be declared. For instance, B is a synonym for A in

```
sort B = A;
```

Using sort references it is possible to define recursive sorts. For example, a sort of binary trees with numbers as their leaves can be defined as follows:

```
sort T = struct leaf(N) | node(T, T);
```

This declares sort T with constructor functions $leaf : \mathbb{N} \rightarrow T$ and $node : T \times T \rightarrow T$, without projection and recogniser functions.

Standard functions

For all sorts the equality operator \approx , inequality $\not\approx$, conditional *if* and quantifiers \forall and \exists are provided. For the user-defined data types the user has to provide equations giving meaning to \approx . For the standard data types and the type formers this operation is defined as expected. The inequality operator, the conditional and the quantifiers are defined for all sorts as expected.

So for instance, with n a variable over the natural numbers \mathbb{N} , the expression $n \approx n$ is equal to *true* and $n \not\approx n$ is equal to *false*. Using the above declaration

of sort A and map f , if $(true, c, d)$ is equal to c and $\forall x:A (f(x, c) \approx c)$ is equal to $true$.

Also, expressions of sort \mathbb{B} may be used as *conditions* in equations, for instance:

```
var    $x, y : A;$ 
eqn   $x \approx y \rightarrow f(x, y) = x;$ 
```

Furthermore, *lambda abstractions* and *where clauses* can be used. For example:

```
map   $h, h' : A \rightarrow A \rightarrow A;$ 
var   $x, y : A;$ 
eqn   $h(x)$                  $= \lambda y':A (\lambda z:A f(z, g(z)))(g(f(x, y')));$ 
       $h'(x)(y)$             $= f(z, g(z))$  whr  $z = g(f(x, y))$  end;
```

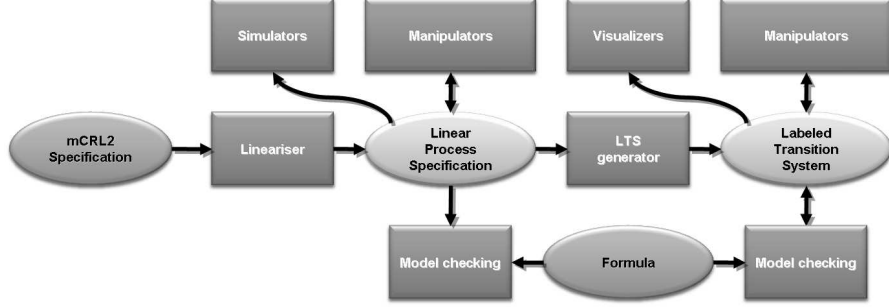
Here functions h and h' are equivalent, i.e. it is possible to show that $h = h'$ (using extensionality).

1.3 Verification Methodology and the mCRL2 Toolset

mCRL2 models can be verified according to different aspects. One might be interested if two mCRL2 models are equivalent according to some notion of process equivalence. Or validity of temporal properties, like absence of deadlock, can be examined with respect to the mCRL2 model. In this section we present the methodology that allows to perform such verifications on mCRL2 models. We also mention the tools from the mCRL2 toolset which allow one to perform steps of the verification procedures automatically.

The mCRL2 Toolset [?] has been developed at Technical University of Eindhoven to support formal reasoning about systems specified in mCRL2. It is based on term rewriting techniques and on formal transformation of process-algebraic and data terms. At the time of writing it allows one to generate, simulate, reduce and visualise state spaces; to search for deadlocks and particular actions; to perform symbolic optimisations for mCRL2 specifications (e.g. invariant and confluence reductions); and to verify modal formulas that contain data. Fig. ?? gives an overview is given of the toolset. The ovals represent the forms of data the tools (represented by rectangles) manipulate.

The *lineariser* transforms a restricted though practically expressive subset of mCRL2 specifications to Linear Process Specifications (LPSs). These LPSs are a compact symbolic representation of the labelled transition system of the specification. Due to its restricted form, an LPS is especially suited as input for tools; there is no need for such tools to take into account all the different

Figure 1.1: The mCRL2 Toolset (www.mcrl2.org)

operators of the complete language. (See [?] for the details of the linearisation process.)

An LPS¹ contains a single process definition of the *linear form*:

proc $P(x:D) = \sum_{i \in I} \sum_{y_i: E_i} c_i(x, y_i) \rightarrow \alpha_i(x, y_i) \cdot P(g_i(x, y_i));$
init $P(d_0);$

where data expressions of the form $d(x_1, \dots, x_n)$ contain at most free variables from $\{x_1, \dots, x_n\}$, I is a finite index set, and for $i \in I$

- $c_i(x, y_i)$ are boolean expressions representing the conditions,
- $\alpha_i(x, y_i)$ is a multiaction $\mathbf{a}_i^1(f_i^1(x, y_i)) | \dots | \mathbf{a}_i^{n_i}(f_i^{n_i}(x, y_i))$, where $f_i^k(x, y_i)$ (for $1 \leq k \leq n_i$) are the parameters of action name \mathbf{a}_i^k ,
- $g_i(x, y_i)$ is an expression of sort D representing the next state of the process definition P ;
- d_0 is a closed data expression;
- $\sum_{i \in I} p_i$ is a shorthand for $p_1 + \dots + p_n$, where $I = \{1, \dots, n\}$ (δ in case $n = 0$).

The form of the summand as described above is sometimes presented as the *condition-action-effect* rule. In a particular state d and for some data value e the multiaction $\alpha_i(d, e)$ can be done if condition $c_i(d, e)$ holds. The effect of the action on the state is given by the fact that the next state is $g_i(d, e)$.

Once an LPS has been generated from a mCRL2 specification, there are a number of possible steps forward. With *simulation* tools one can quickly

¹Here, for the sake of simplicity, we present an untimed version of the LPS which cannot terminate.

gain insight in the behaviour of the system. It is possible to manually select transitions but traces can also be automatically generated. Traces – either previously generated with the simulator or obtained via other tools – can be inspected with the simulator.

With the tools based on *theorem proving* it is possible to check invariants of a system or to detect confluence [?]. The *model checking* tools take an LPS and a modal formula – representing some functional requirement – and create a Parameterised Boolean Equation System (PBES) [?, ?] stating whether or not the formula holds for the specification. These formulae are written in a variant of the modal μ -calculus extended with regular expressions [?]; for example, to state that a system is deadlock free we can write $[true^*](true)true$ or, equivalently, $\nu X.\langle true \rangle true \vee [true]X$. That the number *leave* actions never exceeds the number of *enter* actions is expressed by $\nu X(n:\mathbb{N} := 0).[enter]X(n+1) \wedge [leave](n > 0 \wedge X(n-1))$. Together with an LPS such a formula can be automatically translated to a PBES and symbolically solved by different tools of the toolset (either via conversion to a Boolean Equation System (BES) or by direct manipulation of the PBES).

As symbolic model checking is not yet always successful and because certain tools and toolsets only work on LTSs, it is possible to use the *LTS generator* to construct the explicit state space of a specification. Besides creating such an LTS, it is also possible to automatically check for presence of deadlocks, and certain actions, and to generate a trace to each deadlock state. A variety of exploration techniques is available (such as breadth-first and depth-first searches as well as random simulation).

There are also several *LPS manipulation* tools [?]. These tools can manipulate LPS in such a way that other tools perform better on such an LPS. Typically they remove unused parts of an LPS or detect certain invariants that can be made explicit. Examples are tools that detect that certain process parameters are not used or that certain parameters always have the same value.

Besides a wide range of tools on LPSs, the toolset also contains some tools that *manipulate* LTSs, most notably LTS reduction tools that can minimise an LTS modulo certain equivalences (e.g. trace equivalence, and strong and branching bisimilarity).

With the *LTS visualisation* tools one can gain insight in systems whose size ranges from very small to the quite large. In Sect. ?? there are several examples of images that have been created using these tools. Each tool has a different way of showing a state space: either by using automatic positioning algorithms or by clustering states based on state information. These visualisation tools have proven to be useful in detecting simple properties as well as more complex properties such as symmetry.

In Sect. ?? we demonstrate in slightly more detail how we can use the toolset to validate systems.

1.4 Examples of Modelling and Verification in mCRL2

Similar to other modelling languages, in order to model a system in mCRL2 it is often important to follow the two steps:

- Determine how the system interfaces with the outside world, which actions of the system are visible to the outside world, and on which events the system reacts.
- Decompose the system into a number of parallel components and establish how the components interact.

In this section we demonstrate the language and the toolset with several simple examples.

1.4.1 Dining Philosophers Problem

A classical example of a concurrent system is the well-known *Dining Philosophers Problem* [?]. We illustrate how to model this problem in mCRL2 and give a procedure to find deadlocks and traces leading to these deadlocks using the tools from the mCRL2 toolset.

The dining philosophers problem tells the story of a group of philosophers who live together in a house. In that house there is a table at which each philosopher has its own plate. In between each pair of neighbouring plates there is precisely one fork. The dish they are served for dinner requires two forks to be eaten. In other words, each pair of neighbouring entities (philosophers) shares one resource (fork).

The mCRL2 model of the problem is presented below. For simplicity we only consider three philosophers.

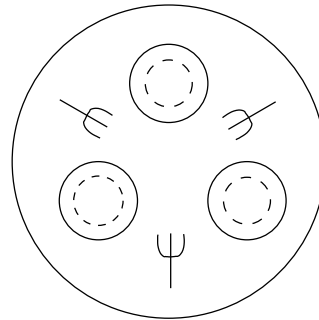


Figure 1.2: Dining table for three philosophers.


```

sort   PhilId = struct  $p_1 \mid p_2 \mid p_3$ ;
         ForkId = struct  $f_1 \mid f_2 \mid f_3$ ;

map     $lf, rf : PhilId \rightarrow ForkId$ ;
eqn     $lf(p_1) = f_1; lf(p_2) = f_2; lf(p_3) = f_3$ ;
          $rf(p_1) = f_2; rf(p_2) = f_3; rf(p_3) = f_1$ ;

act    get, put, up, down, lock, free:  $PhilId \times ForkId$ ;
         eat:  $PhilId$ ;

proc    $Phil(p:PhilId) = (\mathbf{get}(p, lf(p)) \parallel \mathbf{get}(p, rf(p))) \cdot \mathbf{eat}(p) \cdot$ 
          $(\mathbf{put}(p, lf(p)) \parallel \mathbf{put}(p, rf(p))) \cdot Phil(p)$ ;
          $Fork(f:ForkId) = \sum_{p:Phil} \mathbf{up}(p, f) \cdot \mathbf{down}(p, f) \cdot Fork(f)$ ;

init    $\partial_{\{\mathbf{get}, \mathbf{put}, \mathbf{up}, \mathbf{down}\}} (\Gamma_{\{\mathbf{get} \mid \mathbf{up} \rightarrow \mathbf{lock}, \mathbf{put} \mid \mathbf{down} \rightarrow \mathbf{free}\}} ($ 
          $Fork(f_1) \parallel Fork(f_2) \parallel Fork(f_3) \parallel Phil(p_1) \parallel Phil(p_2) \parallel Phil(p_3)$ 
          $));$ 

```

In this model the sort *PhilId* contains the philosophers. The n th philosopher is denoted by p_n . Similarly, the sort *ForkId* contains the forks (f_n denotes the n th fork). The functions lf and rf take a philosopher and return his left and right fork, respectively.

Actions $\mathbf{get}(p_n, f_m)$ and $\mathbf{put}(p_n, f_m)$ are performed by the philosopher process. They model that the philosopher p_n gets or puts down fork f_m . The corresponding actions $\mathbf{up}(p_n, f_m)$ and $\mathbf{down}(p_n, f_m)$ are performed by the fork process. They model the fork f_m being taken or put down by philosopher p_n . The action $\mathbf{eat}(p_n)$ models philosopher p_n eating. For communication purposes we have actions **lock** and **free**. These will represent a fork actually being taken, respectively put down by a philosopher.

The process $Phil(p_n)$ models the behaviour of the n th philosopher. It first takes the forks on his left and right (in any order; possibly even at the same time), then eats, then puts both forks back (again in any order) and repeats its own behaviour.

The process $Fork(f_n)$ models the behaviour of the n th fork (i.e. the fork on the left of the n th philosopher). For any philosopher p it can perform $\mathbf{up}(p, f_n)$ meaning that the fork is being taken by philosopher p . Then it performs $\mathbf{down}(p, f_n)$ meaning that the same philosopher puts the fork down. Finally it repeats its own behaviour again.

The whole system consists of three **Phil** and three **Fork** processes in parallel. By enforcing communication between actions **get** and **up** and between **put** and **down**, we ensure that forks agree on being taken or put down by the philosophers. The result of these communications are **lock** and **free**, respectively. Note that the the communication operator only ensures that communication happens when possible. The blocking operator makes sure that nothing else

happens (i.e. by disallowing the loose occurrences of the actions `get`, `put`, `up`, and `down`).

This model of the problem can be analysed by generating the underlying LTS (consisting of 93 states and 431 transitions) and searching for the deadlocks while doing this. After linearising the specification and exploring the LTS we can find that the trace `lock(3, 3) · lock(2, 2) · lock(1, 1)` leads to a deadlock. (The LTS generator has a special option to detect deadlocks and report (shortest) traces to them.)

By using the mCRL2 simulator one can confirm that this trace indeed leads to a deadlock (either by manually executing this trace, or – especially in the case of longer traces – by loading the trace generated by the LTS generator). The detected deadlock represents the situation when each of the philosophers has taken one fork and waits for another one without being able to put the first one back. Note that due to the true-concurrency nature of mCRL2, this deadlock can also be reached by a single multiaction. That is, with the multiaction `lock(3, 3) | lock(2, 2) | lock(1, 1)` denoting that all philosophers take their left fork at exactly the same time. (In cases where this is not desired the restriction operator can be used to allow only single actions.)

The typical first attempt to a solution is stated as follows: let the philosophers pick the forks in a fixed order (first left, then right). That is, replace process `Phil` by the following:

```
proc Phil(p : PhilId) = get(p, lf(p)) · get(p, rf(p)) · eat(p) ·
                        put(p, lf(p)) · put(p, rf(p)) · Phil(p);
```

The Labelled Transition System of this specification consists of 35 states and 97 transitions and is depicted in Fig. ???. Of course, this system still contains a deadlock state; the above trace is still a valid trace in this system and thus leads to a deadlock.

One of the solutions is to cross the arms of one of the philosophers (or just tell him to pick the forks in reverse order). We change process `Phil` into the following:

```
proc Phil(p : PhilId) = (p ≈ p1) → get(p, rf(p)) · get(p, lf(p)) · eat(p) ·
                        put(p, rf(p)) · put(p, lf(p)) · Phil(p)
                        ◇ get(p, lf(p)) · get(p, rf(p)) · eat(p) ·
                        put(p, lf(p)) · put(p, rf(p)) · Phil(p);
```

This results in a LTS (depicted in Fig. ??) consisting of 36 states and 104 transitions without any deadlock states.

1.4.2 Alternating bit protocol

We now provide a model of a simple communication protocol, namely the alternating bit protocol (ABP) [?, ?]. The alternating bit protocol ensures

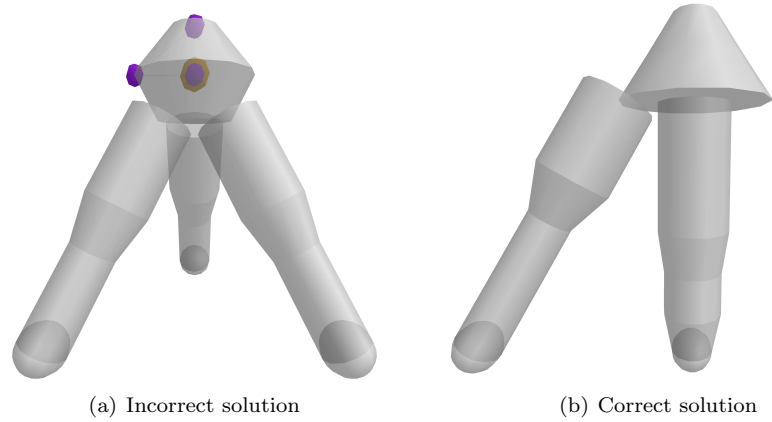


Figure 1.3: State spaces of two solutions to the dining philosophers problem

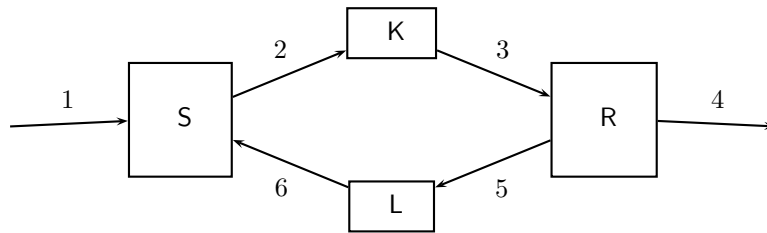


Figure 1.4: Alternating bit protocol

successful transmission of data through lossy channels. The protocol is depicted in Fig. ???. The processes K and L are channels that transfer messages from sender to receiver and vice versa, respectively. However, the data can be lost in transmission, in which case the processes K and L deliver an error. The sender process S and receiver R must take care that despite this loss of data, transfer between 1 and 4 is reliable, in the sense that messages sent at 1 are received at 4 exactly once in the same order in which they were sent.

In order to model the protocol, we assume some sort D with data elements to be transferred. We model the external behaviour of the Alternating Bit Protocol using the following process equation which defines a simple buffer B and we expect that the modelled protocol is branching bisimulation equivalent to this buffer:

$$\text{proc } B = \sum_{d:D} r_1(d) \cdot s_4(d) \cdot B;$$

Action $r_1(d)$ represents “read datum d from channel 1”, and action $s_4(d)$

represents “send datum d on channel 4”. Note that the external behaviour is actually the simplest conceivable behaviour for a data transfer protocol.

In order to develop the sender and the receiver we must have a good understanding of the exact behaviour of the channels K and L . As will be explained below, the process K sends pairs consisting of a message and a bit, and the process L only forwards bits. We can introduce a data sort *Bits*, but it is more efficient to use the booleans to represent bits. The processes choose internally whether data is delivered or lost using the action i . If it is lost an error message \perp is delivered:

$$\begin{aligned} \text{proc } K &= \sum_{d:D} \sum_{b:\mathbb{B}} r_2(d, b) \cdot (i \cdot s_3(d, b) + i \cdot s_3(\perp)) \cdot K; \\ L &= \sum_{b:\mathbb{B}} r_5(b) \cdot (i \cdot s_6(b) + i \cdot s_6(\perp)) \cdot L; \end{aligned}$$

Note that the action i cannot be omitted. If it would be removed, the choice between delivering the correct data or the error is made while interacting with the receiver of the message. The receiver can henceforth determine whether the data will be lost or not. This is not what we want to model here. We want to model that whether or not data are lost is determined internally in K and L . Because the factors that cause the message to be lost are outside our model, we use a non-deterministic choice to model data loss.

We model the sender and receiver using the protocol proposed in [?, ?]. The first aspect of the protocol is that the sender must guarantee that despite data loss in K , data eventually arrive at the receiver. For this purpose, it iteratively sends the same messages to the sender. The receiver sends an acknowledgement to the sender whenever it receives a message. If a message is acknowledged the sender knows that the message is received and it can proceed with the next message.

A problem of this protocol is that a datum may be sent more than once, and the receiver has no way of telling whether the datum stems from a single message which is resent, or whether it stems from two messages that contain the same data. In order to distinguish between these options extra control information must be added to the message. A strong point made in [?, ?] is that adding a single bit already suffices for the job. For consecutive messages the bit is alternated for each subsequent new datum to be transferred. If a datum is resent, the old bit is used. This explains the name Alternating Bit Protocol.

After receiving a message at the receiver, its accompanying bit is sent back in the acknowledgement. When the bit differs from the bit associated with the last message, the receiver knows that this concerns a new datum and forwards it to gate 4. upon reception of \perp , the receiver does not know whether this regards an old or new message, and it sends the old bit to indicate that resending is necessary.

Upon receipt of an unexpected bit or an error message, the sender knows that the old datum must be resent. Otherwise, it can proceed to read new data from 1 and forward it.

First, we specify the sender S 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 $b \in \{0, 1\}$:

$$\begin{aligned} \text{proc } S(b:\mathbb{B}) &= \sum_{d:D} r_1(d) \cdot T(d, b); \\ T(d:D, b:\mathbb{B}) &= s_2(d, b) \cdot (r_6(b) \cdot S(-b) + (r_6(-b) + r_6(\perp)) \cdot T(d, b)); \end{aligned}$$

In state $S(b)$, the sender reads a datum d from channel 1. Next, the system proceeds to state $T(d, b)$, in which it sends this datum on channel 2, with the bit/boolean b attached to it. Then the sender expects to receive the acknowledgement b through channel 6, ensuring that the pair (d, b) has reached the receiver unscathed. If the correct acknowledgement b is received, then the system proceeds to state $S(-b)$, in which it is going to send out a datum with the bit $\neg b$ attached to it. If the acknowledgement is either the wrong bit $\neg b$ or the error message \perp , then the system sends the pair (d, b) on channel 2 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 $b \in \{0, 1\}$:

$$\begin{aligned} \text{proc } R(b:\mathbb{B}) &= \sum_{d:D} r_3(d, b) \cdot s_4(d) \cdot s_5(b) \cdot R(-b) \\ &+ (\sum_{d:D} r_3(d, \neg b) + r_3(\perp)) \cdot s_5(-b) \cdot R(b); \end{aligned}$$

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

1. If in $R(b)$ the receiver reads a pair (d, b) from channel 3, then this constitutes new information, so the datum d is sent on channel 4, after which acknowledgement b is sent to the sender via channel 5. Next, the receiver proceeds to state $R(-b)$, in which it is expecting to receive a datum with the bit $\neg b$ attached to it.
2. If in $R(b)$ the receiver reads a pair $(d, \neg b)$ or an error message \perp from channel 3, then this does not constitute new information. So then the receiver sends acknowledgement $\neg b$ to the sender via channel 5 and remains in state $R(b)$.

The desired concurrent system is obtained by putting $S(true)$, $R(true)$, K and L in parallel, blocking send and read actions over internal channels, and abstracting away from communication actions over these channels and from the action i . That is, the Alternating Bit Protocol (ABP) is defined by the process term

$$\text{proc } \text{ABP} = \nabla_I(\Gamma_G(S(true) \parallel K \parallel L \parallel R(true)));$$

where $I = \{r_1, s_4, c_2, c_3, c_5, c_6, i\}$ and $G = \{r_2 \mid s_2 \rightarrow c_2, r_3 \mid s_3 \rightarrow c_3, r_5 \mid s_5 \rightarrow c_5, r_6 \mid s_6 \rightarrow c_6\}$.

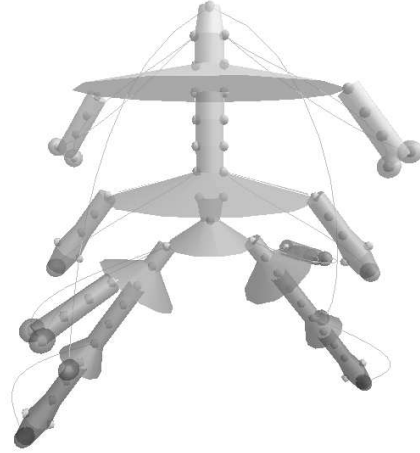


Figure 1.5: Visualisation of the state space of the Alternating Bit Protocol with $|D| = 2$

After linearisation and LTS generation of this specification we can visualise the system as shown in Fig. ??.

The interesting question here is whether the scheme with alternating bits works correctly. Or, in other words, do the buffer B and the alternating bit protocol ABP behave the same when only the actions r_1 and s_4 are visible? This question is concisely stated by asking whether the following equation holds in branching bisimulation semantics:

$$B = \tau_{\{c_2, c_3, c_5, c_6, i\}}(ABP).$$

Using the toolset this can easily be shown to hold by generating the LTS of both processes and automatically comparing them modulo branching bisimulation equivalence.

1.4.3 Sliding Window Protocol (SWP)

In the ABP, the sender sends out a datum and then waits for an acknowledgement 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 (SWP) [?] (see also [?]), this situation has been resolved as the sender can send out multiple data elements before it requires an acknowledgement. This protocol is so effective that it is one of the core protocols of the Internet.

The most complex SWP described in [?] was modelled in 1991 using techniques as described in [?]. This model revealed a deadlock. When confronted with this, the author of [?] indicated that this problem remained undetected

for a whole decade, despite the fact that the protocol had been implemented a number of times. There is some evidence that this particular deadlock occurs in actual implementations of internet protocols, but this has never been systematically investigated. In recent editions of [?] this problem has been removed.

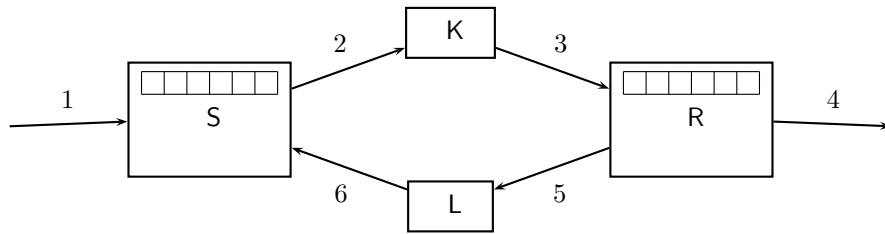


Figure 1.6: Sliding window protocol

We concentrate on a variant of the SWP which is unidirectional, to keep the model to its essential minimum. The essential feature of the SWP is that it contains buffers in the sender and the receiver to keep copies of the data in transit. This is needed to be able to resend this data if it turns out after a while that the data did not arrive correctly. Both buffers have size n . This means that there can be at most $2n$ data elements under way. This suggests that the external behaviour of the SWP is a bounded first-in-first-out (fifo) queue of length $2n$.

As in the ABP, elements from a nonempty data domain D are sent from a sender to a receiver. The external behaviour of the SWP is a fifo queue defined by the following equation, where the sliding window protocol with buffer size n behaves as $\text{FIFO}([], 2n)$:

$$\begin{aligned} \text{proc } \text{FIFO}(l:\text{List}(D), m:\mathbb{N}^+) \\ &= \#l < m \rightarrow \sum_{d:D} r_1(d) \cdot \text{FIFO}(l \triangleright d, m) \\ &+ \#l > 0 \rightarrow s_4(\text{head}(l)) \cdot \text{FIFO}(\text{tail}(l), m); \end{aligned}$$

Note that $r_1(d)$ can be performed until the list l contains m elements, because in that situation the sending and receiving windows will be filled. Furthermore, $s_4(\text{head}(l))$ can only be performed if l is not empty.

We now give a model of the sliding window protocol which implements the bounded buffer on top of two unreliable channels K and L . The setup is similar to that of the alternating bit protocol. See Fig. ??.

The channels differ from those of the ABP because they do not deliver an error in case of data loss. An indication of an error is necessary for the ABP to work correctly. But it is unrealistic. A better model of a channel is one where data are lost without an explicit indication. In this case we still assume that the channels can only carry a single data item, and have no buffer capacity. But the channels can be replaced by others, for instance with bounded, or unbounded capacity. As long as the channels now and then transfer a message, the sliding window protocol can be used to transform these unreliable channels into a fifo queue.

```

proc  K =  $\sum_{d:D, k:\mathbb{N}} r_2(d, k) \cdot (i \cdot s_3(d, k) + i) \cdot K;$ 
        L =  $\sum_{k:\mathbb{N}} r_5(k) \cdot (i \cdot s_6(k) + i) \cdot L;$ 

```

The sender and the receiver in the SWP both maintain a buffer of size n containing the data being transmitted. The buffers are represented by a function from \mathbb{N} to D indicating which data element occurs at which position. Only the first n places of these functions are used. In the receiver we additionally use a buffer of booleans of length n to recall which of the first n positions in the buffer contain valid data.

```

sort  DBuf =  $\mathbb{N} \rightarrow D;$ 
        BBuf =  $\mathbb{N} \rightarrow \mathbb{B};$ 

```

The SWP uses a numbering scheme to number the messages that are sent via the channels. It turns out that if the sequence numbers are issued modulo $2n$, messages are not confused and are transferred in order. Each message with sequence number j is put at position $j|_n$ (j modulo n) in the buffers.

We use the following auxiliary functions to describe the sliding window protocol. The function *empty* below represents a boolean buffer that is false everywhere, indicating that there is no valid data in the buffer. We use notation $q[i:=d]$ to say that position i of buffer q is filled with datum d . Similarly, $b[i:=c]$ is buffer b where c is put at position i .

The most involved function is $nextempty_{mod}(i, b, m, n)$. It yields the first position in buffer b starting at $i|_n$ that contains *false*. If the first m positions from $i|_n$ of b are all *true*, it yields the value $(i+m)|_{2n}$. The variable m is used to guarantee that the function *nextempty* is well defined if b is *true* at all its first n positions. The variables have the following sorts: $d:D$, $i, j, m:\mathbb{N}$, $n:\mathbb{N}^+$, $q:DBuf$, $c:\mathbb{B}$ and $b:BBuf$.

```

eqn  empty =  $\lambda j:\mathbb{N}.false;$ 
         $q[i:=d] = \lambda j:\mathbb{N}.if(i \approx j, d, q(j));$ 
         $b[i:=c] = \lambda j:\mathbb{N}.if(i \approx j, c, b(j));$ 
        nextemptymod( $i, b, m, n$ ) =
           $if(b(i|_n) \wedge m > 0, nextempty_{mod}((i+1)|_{2n}, b, m-1, n), i|_{2n});$ 

```


Below we model the sender process S . The variable ℓ contains the sequence number of the oldest message in sending buffer q and m is the number of items in the sending buffer. If a datum arrives via gate 1, it is put in the sending buffer q , provided there is place. There is the possibility to send the k th datum via gate 2 with sequence number $(\ell+k)|_{2n}$ and there is a possibility that an acknowledgement arrives via gate 6. This acknowledgement is the index of the first message that has not yet been received by the receiver.

$$\begin{aligned} \text{proc } S(\ell, m: \mathbb{N}, q: DBuf, n: \mathbb{N}^+) & \\ &= (m < n) \rightarrow \sum_{d:D} r_1(d) \cdot S(\ell, m+1, q[(\ell+m)|_n := d], n) \\ &+ \sum_{k:\mathbb{N}} (k < m) \rightarrow s_2(q((\ell+k)|_n), (\ell+k)|_{2n}) \cdot S(\ell, m, q, n) \\ &+ \sum_{k:\mathbb{N}} r_6(k) \cdot S(k, (m-k+\ell)|_{2n}, q, n); \end{aligned}$$

The receiver is modelled by the process $R(\ell', q', b, n)$ where ℓ' is the sequence number of the oldest message in the receiving buffer q' . A datum can be received via gate 3 from channel K and is only put in the receiving buffer q' if its sequence number k is in the receiving window. If sequence numbers and buffer positions would not be considered modulo $2n$ and n this could be stated by $\ell' \leq k < \ell' + n$. The condition $(k - \ell')|_{2n} < n$ states exactly this, taking the modulo boundaries into account.

The second summand in the receiver says that if the oldest message position is valid (i.e. $b(\ell'|_n)$ holds), then this message can be delivered via gate 4. Moreover, the oldest message is now $(\ell'+1)|_{2n}$ and the message at position $\ell'|_n$ becomes invalid.

The last summand says that the index of the first message that has not been received at the receiver is sent back to the sender as an acknowledgement that all lower numbered message have been received.

$$\begin{aligned} \text{proc } R(\ell': \mathbb{N}, q': DBuf, b: BBuf, n: \mathbb{N}^+) & \\ &= \sum_{d:D, k:\mathbb{N}} r_3(d, k) \cdot ((k-\ell')|_{2n} < n \\ &\quad \rightarrow R(\ell', q'[(k|_n) := d], b[(k|_n) := true], n) \\ &\quad \diamond R(\ell', q', b, n)) \\ &+ b(\ell'|_n) \rightarrow s_4(q'(\ell'|_n)) \cdot R((\ell'+1)|_{2n}, q', b[(\ell'|_n) := false], n) \\ &+ s_5(\text{nextempty}_{mod}(\ell', b, n, n)) \cdot R(\ell', q', b, n); \end{aligned}$$

The behaviour of the SWP is characterised by:

$$\text{proc } SWP(q, q': DBuf, n: \mathbb{N}^+) = \nabla_H(\Gamma_G(S(0, 0, q, n) \parallel K \parallel L \parallel R(0, q', \text{empty}, n)));$$

where the set $H = \{c_2, c_3, c_5, c_6, i, r_1, s_4\}$ and $G = \{r_2 \mid s_2 \rightarrow c_2, r_3 \mid s_3 \rightarrow c_3, r_5 \mid s_5 \rightarrow c_5, r_6 \mid s_6 \rightarrow c_6\}$. The contents of q and q' can be chosen arbitrarily without affecting the correctness of the protocol. This is stressed by not instantiating these variables. The sliding window protocol behaves as a bounded first-in-first-out buffer for any $n > 0$ and $q, q': DBuf$:

$$\text{FIFO}([], 2n) = \tau_I(\text{SWP}(q, q', n))$$

where $I = \{c_2, c_3, c_4, c_5, i\}$.

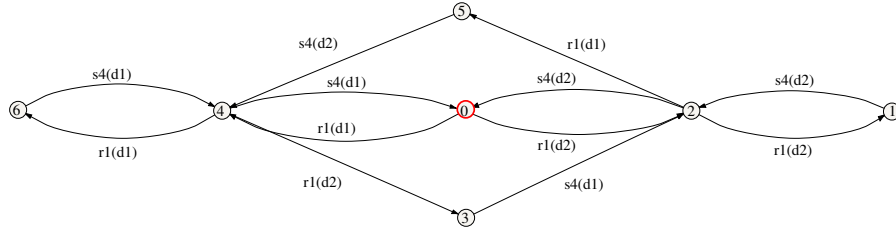


Figure 1.7: The LTS corresponding to the SWP with window size 1 and 2 data elements.

In Fig. ?? the minimised modulo branching bisimulation equivalence LTS that corresponds to the SWP with window size 1 and 2 data elements is presented. It is not difficult to see that the 2-place buffer has a bisimilar LTS.

Proving correctness of SWP for any window size and for an arbitrary data domain is a more difficult problem that involves techniques based on theorem proving, like the cones and foci method [?]. In [?] it is proven that the equivalence between SWP and the queue holds also in this case. Due to the tricky nature of modulo calculation, this proof is quite involved.

1.5 Historical context

Work on process algebra's can be traced back to observations by Milner and Bekiç around 1973 [?, ?]. They observed that actions are the basic notion to describe behaviour. This led to the development of three main process algebraic formalisms (CCS [?], ACP [?] and CSP [?]). These formalisms did not contain data and primarily served as vehicles to study the mathematical and semantical properties of such languages.

A major next step was the extension of process algebras to overcome the lack of practical expressibility that process algebras have. The most well known is LOTOS [?, ?]. It contains equational abstract data types, a module mechanism, interrupts and mechanisms to yield return values. LOTOS became an international standard in 1990. A languages that is very similar to LOTOS is PSF [?]. A later member to be added to the family of these

process algebras is FDR [?], although it does not use abstract data types.

In an attempt to specify a universal process algebraic language (the *Common Representation Language*, *CRL*), it was realized that these extended process algebras became too complex, which would hamper further development. In response a micro Common Representation Language (μ CRL [?, ?]) was defined. This language has essentially the same expressibility of its larger brothers (LOTOS, PSF and CRL), but does not contain unnecessary decorum.

For approximately one decade the language μ CRL has been used to develop theory, proof methodology, algorithms and an analysis toolset. This led to a variety of axiomatization and expressivity results [?]. The cones and foci theorem [?], confluence and τ -priorisation results [?] enabled the verification of more complex protocols, culminating in the proof-checked algebraic verification of the sliding window protocol [?]. In response to modelling of several industrial case studies, a toolset for μ CRL was developed [?], which had as an interesting side effect work on visualizing huge state spaces [?] and parameterised boolean equation systems to verify modal formulas with data and time [?, ?].

While work using μ CRL was progressing it was felt that the abstract data types became more and more a nuisance. The problem was not with expressivity as every system could easily be expressed using the abstract data types. The problem was that abstract data types are less suitable as a means to communicate and to build up knowledge. Using abstract data types it is required to specify datatypes such as booleans and natural numbers for each specification. This means that when studying a specification, these user defined standard datatypes must be read and understood, because they can (and will) differ at subtle points from user defined standard data types defined by others. Furthermore, no standard meta results on the datatypes can be accumulated (e.g. associativity of addition, distributivity of modulo operations), because they do not hold for all possible equational specifications of these operators. The use of abstract data types also led to rather long μ CRL specifications, generally starting with a dozen pages of data types. A disadvantage that we only saw after defining and using mCRL2 is the lack of sets and functions. This meant that most data types were encoded in lists (e.g. arrays are modelled as lists), which is not efficient for state space generation, awkward to read and unpleasant to prove properties about. It is worth noting that the clarity of abstract equational data types has substantial advantages. They are easier to learn, a perfect means of communication when it comes to discussions about details and relatively straightforward to implement efficiently [?].

With these arguments, it was decided to extend μ CRL with concrete data types and give the resulting language the somewhat uninspiring name mCRL2. Under influence of a.o. the work bij Dijkstra c.s. [?] it was felt that a proper specification language should not only contain basic datatypes but also commonly used mathematical objects such as functions, sets and quantifiers. Furthermore, in the style of abstract datatypes, it was considered necessary

that these datatypes should not be a finite approximation, but be the ‘real thing’. So, numbers have an infinite range, sets can both be finite and infinite and quantifiers can both have finite and infinite domains. Among the process algebraic inspired languages mCRL2 is unique in this respect (other ‘rich’ formalisms are Cold [?], Z [?], and languages used in proof checkers such as Coq [?] and PVS [?]). Besides a major extension of the data types, mCRL2 differs in some minor aspects from μ CRL, such as the semantics of time and the availability of multi-actions.

A similar movement as from μ CRL to mCRL2 can be observed in the development of E-LOTOS [?] from LOTOS. The most notable difference is that E-LOTOS is developed with the implementability of the language as guiding principle, whereas mCRL2 has been developed with speciifiability in mind. So, typically mCRL2 contains quantifiers and sums over infinite data types. E-LOTOS has no quantifiers and restricts sums to finite data types. mCRL2 has infinite sets and natural numbers with unbounded range (which has a performance penalty). E-LOTOS maps the numbers on machine based numbers with a finite range. Reversely, E-LOTOS allows exceptions, while loops and assignments that are not available in mCRL2.

What are the future developments for mCRL2? We expect that we see a steady improvement of the language and the tools. More importantly, are the integration of all behavioural aspects in unified formalisms. E.g. behaviour, data, modal logics, time, performance, throughput, continuous interaction and feedback in one mathematical framework. But what we consider the most important development is that attention will shift from the language to its use. We expect articles on specification style. E.g. design by confluence to reduce the state spaces to be analysed, design with synchronized increases of clocks, to enable abstract interpretation techniques. But we also expect that attention will move to the design and study of system behaviour per se using mCRL2 (or any similar formalism) as a self evident means.

Bibliography

- [1] Bahareh Badban, Wan Fokkink, Jan Friso Groote, Jun Pang, and Jaco van de Pol. Verification of a sliding window protocol in $\hat{\mu}\text{CRL}$. *Formal Asp. Comput.*, 17(3):342–388, 2005.
- [2] 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.
- [3] Marc Bezem and Jan Friso Groote. Invariants in process algebra with data. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR*, volume 836 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 1994.
- [4] S. C. C. Blom, W. J. Fokkink, J. F. Groote, I. van Langevelde, B. Lissner, and J. C. van de Pol. μCRL : A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification: 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.
- [5] Stefan Blom, Jens R. Calame, Bert Lissner, Simona Orzan, Jun Pang, Jaco van de Pol, Muhammad Torabi Dashti, and Anton Wijs. Distributed analysis with $u\text{crl}$: A compendium of case studies. In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 683–689. Springer, 2007.
- [6] Stefan Blom and Jaco van de Pol. State space reduction by proving confluence. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 596–609. Springer, 2002.
- [7] J. J. Brunekreef. Sliding window protocols. In S. Mauw and G. J. Veltink, editors, *Algebraic Specification of Communication Protocols*. Cambridge University Press, 1993.
- [8] V. G. Cerf and R. E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22:637–648, 1974.
- [9] Taolue Chen, Bas Ploeger, Jaco van de Pol, and Tim A. C. Willemse. Equivalence checking for infinite systems using parameterized boolean

- equation systems. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2007.
- [10] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 394–397. Springer Verlag, 1996.
- [11] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.
- [12] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4, 2002.
- [13] J. F. Groote. The syntax and semantics of timed μ CRL. Technical Report SEN-R9709, CWI, Amsterdam, 1997.
- [14] J. F. Groote, A. Mathijssen, M. van Weerdenburg, and Y. S. Usenko. From μ CRL to mCRL2: motivation and outline. In *Proc. Workshop Essays on Algebraic Process Calculi (APC 25)*, volume 162 of *Electronic Notes in Theoretical Computer Science*, pages 191–196, 2006.
- [15] J. F. Groote and A. Ponse. The syntax and semantics of μ CRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing, pages 26–62. Springer-Verlag, 1995.
- [16] J. F. Groote and M. A. Reniers. Algebraic process verification. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier Science Publishers B.V., Amsterdam, 2001.
- [17] J. F. Groote and M. P. A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1-2):47–81, 1996.
- [18] Jan Friso Groote and Bert Lissner. Computer assisted manipulation of algebraic process specifications. *SIGPLAN Notices*, 37(12):98–107, 2002.
- [19] Jan Friso Groote and Radu Mateescu. Verification of temporal properties of processes in a setting with data. In Armando Martin Haebeler, editor, *Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98*, volume 1548 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999.
- [20] Jan Friso Groote, Michel A. Reniers, and Yaroslav S. Usenko. Time abstraction in timed μ CRL a la regions. In *IPDPS. IEEE*, 2006.
- [21] Jan Friso Groote and Tim A. C. Willemse. Parameterised boolean equation systems (extended abstract). In Philippa Gardner and Nobuko

- Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference*, volume 3170 of *Lecture Notes in Computer Science*, pages 308–324. Springer, 2004.
- [22] J.F. Groote and J. Springintveld. Focus points and convergent process operators: a proof strategy for protocol verification. *JLAP*, 49(1-2):31–60, 2001.
- [23] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [24] ISO - International Organization for Standardization. *Information processing systems - Open systems interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour, IS 8807*. ISO, 1989.
- [25] W. C. Lynch. Reliable full duplex file transmission over half-duplex telephone lines. *Communications of the ACM*, 11(6):407–410, 1968.
- [26] Aad Mathijssen and A. Johannes Pretorius. Verified design of an automated parking garage. In Lubos Brim, Boudewijn R. Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Applications and Technology, 11th International Workshop, FMICS 2006 and 5th International Workshop PDMC 2006*, volume 4346 of *Lecture Notes in Computer Science*, pages 165–180, 2007.
- [27] mCRL2 toolset showcases. <http://www.mcrl2.org/wiki/index.php/showcases>.
- [28] The mCRL2 website. <http://www.mcrl2.org/>.
- [29] K. Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100(2):385–417, june 1992.
- [30] K. Meinke. Higher-order equational logic for specification, simulation and testing. In *The 1995 Workshop on Higher-Order Algebra, Logic and Term Rewriting (HOA '95)*, volume 1074 of *Lecture Notes in Computer Science*, pages 124–143. Springer-Verlag, 1996.
- [31] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [32] B. Möller, A. Tarlecki, and M. Wirsing. Algebraic specification of reachable higher-order algebras. In *Recent Trends in Data Type Specification*, volume 332 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 1988.
- [33] Bas Ploeger and Lou Somers. Analysis and verification of an automatic document feeder. In Yookun Cho, Roger L. Wainwright, Hisham Hadad, Sung Y. Shin, and Yong Wan Koo, editors, *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC)*, pages 1499–1505, 2007.

- [34] M. A. Reniers, J. F. Groote, M. B. van der Zwaag, and J. van Wamel. Completeness of timed μ CRL. *Fundamenta Informaticae*, 50(3-4):361–402, 2002.
- [35] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.
- [36] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1981.
- [37] G. J. Tretmans and H. Brinksma. TorX: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, 2003.
- [38] Y. S. Usenko. *Linearization in μ CRL*. PhD thesis, Technische Universiteit Eindhoven (TU/e), 2002.
- [39] D. A. van Beek, Ka L. Man, Michel A. Reniers, J. E. Rooda, and Ramon R. H. Schiffelers. Syntax and consistent equation semantics of hybrid χ . *J. Log. Algebr. Program.*, 68(1-2):129–210, 2006.
- [40] Marko van Eekelen, Stefan ten Hoedt, René Schreurs, and Yaroslav S. Usenko. Analysis of a session-layer protocol in mCRL2. verification of a real-life industrial implementation. Technical Report ICIS-R07014, Radboud University Nijmegen, 2007.
- [41] R. J. van Glabbeek. The linear time - branching time spectrum II. In E. Best, editor, *CONCUR'93, International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, 1993.
- [42] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.
- [43] Anton Wijs and Wan Fokkink. From χ -t to μ CRL: Combining performance and functional analysis. In *10th International Conference on Engineering of Complex Computer Systems (ICECCS 2005), 16-20 June 2005, Shanghai, China*, pages 184–193. IEEE Computer Society, 2005.
- [44] T. A. C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. PhD thesis, Technische Universiteit Eindhoven (TU/e), 2003.