

Process Algebra and mCRL2

IPA Basic Course on Formal Methods 2006

19th January 2006

Jan Friso Groote Aad Mathijssen Bas Ploeger Michel Reniers
Muck van Weerdenburg Jeroen van der Wulp

1 Introduction

In a typical distributed system, a number of components are running simultaneously in parallel. By working together, these components provide the functionalities that are required from the complete system. Although the behaviour of a single component can usually be specified and analysed relatively easily, the behaviour of the system as a whole is often too complex to be specified or analysed thoroughly. This is primarily due to (and inherent to) the parallelism between the system's components. An exhaustive analysis of all of the system's states and execution paths thus becomes a formidable task – even for a system with a relatively small number of components.

In this part of the IPA Basic Course on Formal Methods, we introduce and explain process algebra, which is a formalism that is well suited for the specification of system behaviour. This is done within the context of the mCRL2 specification language [4] and toolset [9]. With the toolset, users can specify the behaviour of a distributed system and analyse it using automated techniques. In Section 1.1 we give a short history of mCRL2. In Section 1.2 we give an overview of the remainder of this document.

1.1 History

As its name suggests, mCRL2 is the successor of the μ CRL specification language and toolset [3, 6, 7]. The μ CRL toolset has been developed at and maintained by the Centre for Mathematics and Computer Science (CWI) in Amsterdam since the beginning of the nineties of the previous century. The μ CRL language extends a basic process algebra – based on the Algebra of Communicating Processes (ACP) [1] – with the possibility to define and use abstract data types. The ability to use data within a process algebra specification is a valuable (perhaps even a necessary) enhancement when applying the toolset to a real-life system.

The μ CRL language has clear and well-defined syntax and semantics. Over the years, various tools have been developed for μ CRL, all with a strong foundation in formal theories. The toolset has been used in numerous case studies for the analysis of systems and protocols developed by both the industry and the academic world (see for example [2, 5, 12]). In nearly all cases the analysis revealed errors in the system being analysed.

Recently, researchers at the Eindhoven University of Technology (TU/e) started the development of the mCRL2 language and toolset. Based on user experiences with μ CRL, their focus is to develop a more user friendly language and tool interface. The most substantial improvement to the language on the data side is the introduction of predefined and higher-order data types, lambda calculus expressions and various other language constructs that are designed to make the data type definitions

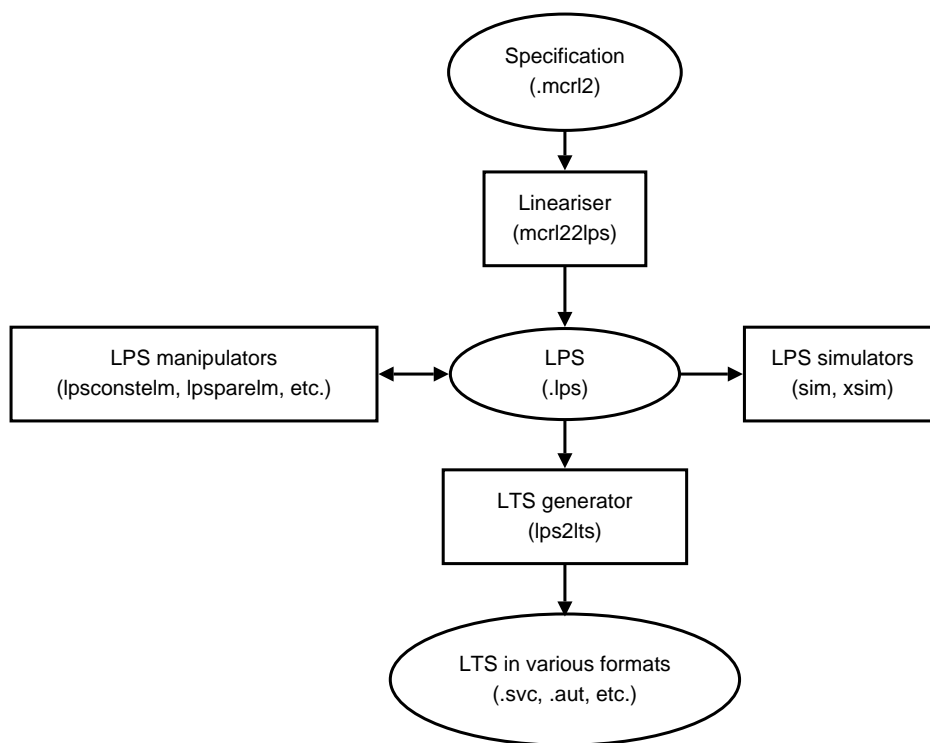


Figure 1: Overview of the mCRL2 toolset

shorter and easier to read and write. Regarding the process algebra, the most remarkable change is the introduction of multiactions which allow for a more straightforward conversion of Petri nets to mCRL2 specifications. Furthermore, the language is truly compositional, meaning that large systems can be specified in terms of smaller components. Finally, the mCRL2 toolset is being extended with a graphical user interface which improves the usability of the toolset and can be used alongside the traditional command line interface.

1.2 Overview

In general, the following steps are involved in the analysis of a system with mCRL2 (see also Figure 1):

- A specification of the system's behaviour is written in the mCRL2 language.
- This specification is converted to a *Linear Process Specification* (LPS) by the mCRL2 lineariser tool. As we shall see, an LPS is an mCRL2 specification in a stricter format.
- The LPS can be modified using various manipulation tools and can be simulated using various simulation tools.
- A *Labelled Transition System* (LTS) or *state space* is generated from the modified LPS.

Subsequently, this LTS can be analysed for errors using model checking techniques. This is a topic which is beyond the scope of this document. The remainder of this document contains the following sections:

- Language (Section 2);
- Linear Process Specifications (Section 3);
- Toolset (Section 4).

Every section contains exercises to practise the material during the course.

2 Language

This section describes the mCRL2 language. Roughly, the language consists of a process algebra part and data specification part. These parts are explained in Sections 2.1 and 2.2, respectively.

The precise syntax of the language can be found in Appendix A. Note that this syntax, which is used in this and the following sections, uses a rich text format. The toolset, however, uses a plain text format, which is related to the rich notation in Appendix B.

2.1 Process algebra

The most basic notion in the mCRL2 process language is an **action**. Actions represent atomic events. The following example illustrates how actions *send*, *receive* and *error* can be declared:

```
act    send, receive;
       error;
```

In general we write a, b, \dots to denote actions.

Process expressions, denoted by p, q, \dots , describe when certain actions can be executed. For example, “ a is followed by either b or c ”. We make this notion more formal by introducing operators.

2.1.1 Basic operators

Process expressions are compositions of actions using a number of operators. The most basic expressions are as follows:

- **Actions** a, b etc. as described above.
- **Deadlock** or inaction δ , which does not display any behaviour.
- **Alternative composition**, written as $p + q$. This represents a *non-deterministic choice* between p and q .
- **Sequential composition**, written $p \cdot q$. This expression first executes p and then q (assuming p terminates).

When writing process expressions we usually omit parentheses as much as possible. To do this, we give \cdot a higher precedence than $+$ and use the associativity of both operators. So, instead of writing $(a \cdot (b \cdot c)) + (d + e)$ we usually write $a \cdot b \cdot c + d + e$.

To give meaning to processes we use LTSs. In Figure 2 the LTS corresponding to the process expression $a \cdot b + c \cdot (d \cdot \delta + e)$ is shown. To be able to distinguish deadlock (δ) and successful termination (e.g. after an action a) we use closed nodes to indicate that a process has terminated and open nodes if a process has not (yet) done so. Each open node, usually called a state, corresponds to a process described by the (sub-)graph with this node as root. In Figure 2 we have labelled the states with the corresponding process expressions.

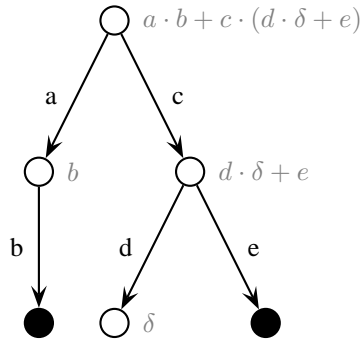


Figure 2: LTS of $a \cdot b + c \cdot (d \cdot \delta + e)$

In Figure 3 the LTSs for the basic operators are drawn. Figure 3(c) denotes the LTS for a general process, meaning that p_0 is the (only) initial state of p and p_1 to p_n ($n \geq 0$) are its terminating states. As shown in Figure 3(d), the alternative composition of two processes is created by merging their initial states. For the sequential composition $p \cdot q$ each of the terminating states of p is replaced with a copy of q (Figure 3(e)).

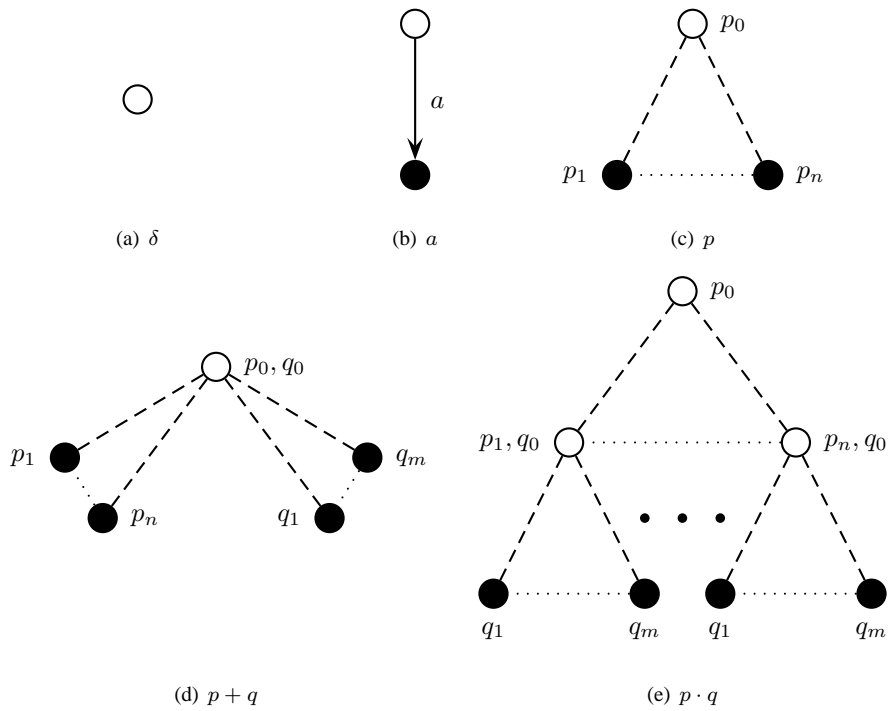


Figure 3: LTSs of the basic operators

Exercise 2.1 Draw the LTS for each of the following processes:

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

4. $\delta \cdot (a + b)$

Looking at Exercises 2.1.1 and 2.1.2, we see that after doing either an a or a b both processes have exactly the same behaviour (namely c). To express this equality with respect to the (observable) behaviour a process describes we use **(strong) bisimulation equivalence**. Which processes are equal, or *bisimilar*, according to this equivalence is stated as follows. For every action one process can perform, the other has to be able to perform the same action as well *and* the resulting processes should be bisimilar. Furthermore, if one process has terminated the other should have done so as well. For example, $a \cdot (b + c)$ can perform an a , which results in a process $b + c$, and $a \cdot b + a \cdot c$ can also perform an action a , but not such that the resulting process is equal to $b + c$. Thus, these processes are not *bisimilar*.

Other equivalences can also be used. With **trace equivalence**, for instance, two processes are equal when the sets of traces that the processes can execute are equal.¹ In this case $a \cdot (b + c)$ and $a \cdot b + a \cdot c$, which are not bisimilar, would be equivalent as for both we have that $\{\epsilon, a, a \cdot b, a \cdot c\}$, with ϵ the empty trace, is the set of traces.

We explain our choice for bisimilarity by means of the story “The Lady, or the Tiger?” by Frank Stockton [14], which tells about a barbaric king who puts his accused subjects in the middle of a public arena. In this arena, there are two closed doors, exactly alike. The fate, and thereby also the guilt, of the accused is decided by forcing him to open one of the two. Behind one of the doors is a hungry tiger which will tear him to pieces immediately when given the opportunity. The other conceals a beautiful lady to whom, once revealed, he will be married instantly.

When modelling such a trial, we treat the opening of the doors as an atomic action, as well as the resulting behaviour after opening each of the doors. The correct model for the above story is $open_door \cdot marry_lady + open_door \cdot confront_tiger$, as depicted in Figure 4(a). It expresses that whether you will be confronted with the lady or the tiger depends on the $open_door$ action. The model $open_door \cdot (marry_lady + confront_tiger)$ in Figure 4(b), however, is wrong, because after opening a door the accused person can somehow still choose between meeting the tiger and marrying the lady.

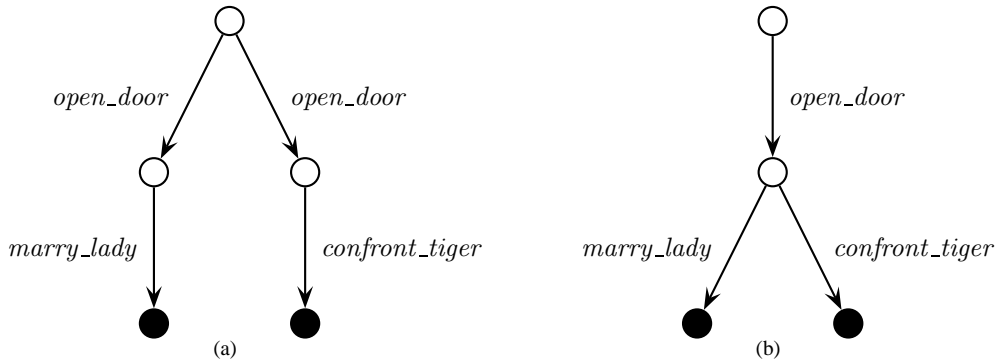


Figure 4: The lady, or the tiger?

We use **axioms** to express the properties of the operators. The axioms for the operators introduced so far are listed in Table 1. Here, x, y, z are variables that stand for unknown process expressions. With the axioms we can prove, for instance, $a + (\delta + a)$ equal to a using axioms A2, A6 and A3 as follows:

$$a + (\delta + a) \stackrel{A2}{=} (a + \delta) + a \stackrel{A6}{=} a + a \stackrel{A3}{=} a$$

¹We will not define the set of traces of a process formally.

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	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
A6	$x + \delta = x$
A7	$\delta \cdot x = \delta$

Table 1: Basic operator axioms

We introduce more axioms along the way. In Appendix C a complete overview of all axioms is given. Note that some axioms given in this section are slightly simplified versions of those given in Appendix C due to the incremental structure of this document.

Exercise 2.2 Prove the following propositions:

1. $a + (\delta + a) = a$ (without using axiom A2)
2. $\delta \cdot (a + b) = \delta \cdot a + \delta \cdot b$
3. $p + q = \delta$ implies $p = \delta$

In some occasions it helps to use $p \leq q$, which is defined as $p + q = q$. This relation is sometimes called *summand inclusion* and can be used to split a proof obligation in two (possibly) simpler parts (see Exercise 2.3.3).

Exercise 2.3 Prove the following propositions:

1. $p \leq p + q$
2. $\delta \leq p$
3. $p = q$ if, and only if, $p \leq q$ and $q \leq p$ (anti-symmetry)
4. $p + q = \delta$ implies $p = \delta$

2.1.2 Recursion

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 introduce **process references**, written as P . These are references to variables declared by process equations, that are introduced next. Using process expressions we can form **process equations**. Take for instance the following declaration:

```

act    switch, break;
proc  Off = switch · On;
        On = break · δ + switch · Off;

```

This declares process references (often just called processes) Off and On . Process Off can do a *switch* action, after which it behaves as process On . Process On can also do a *switch* action and return to process Off , 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 $Off;$

Exercise 2.4 Draw the LTS for Off , as described above.

Note that with recursion the composition of LTSs with the alternative operator as shown in Figure 3(d) is no longer valid. For example, given a specification $P = a \cdot P$, the composition $P + b$ would result in $Q = a \cdot Q + b$ instead of $a \cdot P + b$. The precise correct composition is beyond the scope of this document.

2.1.3 Parallel operators

Having covered the basics, we take a look at some additional operators that play an essential role in process algebra, namely the parallel operators:

- **Parallel composition** or merge $p \parallel q$, which *interleaves* and *synchronises* the actions of p with those of q .
- **Synchronisation operator** $p|q$, which synchronises the first actions of p and q and combines the rest of p and q like the parallel composition.
- **Multiaction** $a|\dots|b$, which is a special instance of the synchronisation operator where the arguments are single actions (or multiactions). The meaning of a multiaction is that all actions occurring in it happen at the same moment (i.e. truly in parallel). We often write α or β for multiactions.
- **Left merge** $p \ll q$, which is an auxiliary operator to allow for the axiomatisation of the parallel composition. (It allows only p to execute a first action and thereafter combines the remainder of p with q as the parallel composition does.)

The precedence of the operators introduced so far, in decreasing order, is as follows: $|$, \cdot , $\{\parallel, \ll\}$, $+$.

The related axioms are given in Table 2. Here α and β are multiactions and α_δ and β_δ are multiactions *or* deadlock.

Exercise 2.5 Prove the following propositions:

1. $(a + b) \parallel c = a \cdot c + b \cdot c + c \cdot (a + b) + a|c + b|c$
2. $a \parallel \delta = a \cdot \delta$
3. $p \parallel q = q \parallel p$

Does the following hold? Why (not)?

4. $(p + q) \parallel r = p \parallel r + q \parallel r$

MA3'	$\alpha \beta = \beta \alpha$
MA4	$\alpha (\beta \gamma) = (\alpha \beta) \gamma$
CM1	$x \parallel y = x \parallel y + y \parallel x + x y$
CM2	$\alpha_\delta \parallel x = \alpha_\delta \cdot x$
CM3	$\alpha_\delta \cdot x \parallel y = \alpha_\delta \cdot (x \parallel y)$
CM4	$(x + y) \parallel z = x \parallel z + y \parallel z$
CM5	$(\alpha_\delta \cdot x) \beta_\delta = \alpha_\delta \beta_\delta \cdot x$
CM6	$\alpha_\delta (\beta_\delta \cdot x) = \alpha_\delta \beta_\delta \cdot x$
CM7	$(\alpha_\delta \cdot x) (\beta_\delta \cdot y) = \alpha_\delta \beta_\delta \cdot (x \parallel y)$
CM8	$(x + y) z = x z + y z$
CM9	$x (y + z) = x y + x z$
CD1	$\delta \alpha_\delta = \delta$
CD2	$\alpha_\delta \delta = \delta$

Table 2: Parallel operator axioms

2.1.4 Additional operators

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

- **Restriction operator** $\nabla_V(p)$, where V is a set of multiactions that specifies exactly which multiactions from p are allowed to occur.
- **Blocking operator** $\partial_H(p)$ (also known as *encapsulation*), where H is a set of action names that are *not* allowed to occur.
- **Renaming operator** $\rho_R(p)$, where R is a set of renamings of the form $a \rightarrow b$, meaning that every occurrence of action a in p is replaced by action b .
- **Communication operator** $\Gamma_C(p)$, where C is a set of allowed communications of the form $a_0 | \dots | a_n \rightarrow c$, with $n \geq 1$, meaning that every group of actions a_0, \dots, a_n within a multiaction is replaced by c .

Before we can give the axioms for these operators, we first need to introduce some special functions on multiactions that correspond to the above operators. We use conversions $_ \langle _ \rangle$ and $_ | _$ to convert multiactions to their corresponding multiset and back, respectively. We extend these conversions to sets V and C in the straightforward way given in Appendix C. We write multisets as $\langle a_1, \dots, a_n \rangle$ and denote them by m and n .

For the blocking operator we need to detect whether or not actions in a multiaction occur in the set of actions H . We do this by taking the intersection of the corresponding multiset with H , resulting in a set (e.g. $\langle a|b|b \rangle \cap \{b, c\} = \langle a, b, b \rangle \cap \{b, c\} = \{b\}$).

The renaming operator works by applying the function R to a multiaction with \bullet . For example, $R \bullet a|b = R(a)|R(b)$, where $R(a) = c$ if $a \rightarrow c \in R$ (otherwise $R(a) = a$). Note that every action may only occur once as a right-hand side of a \rightarrow in R .

Somewhat more complicated is the function $\gamma_C(m)$, which applies the communication described by C to a multiset m . It replaces every occurrence of a left-hand side of a communication it can find in m with the appropriate result. More precisely:

$$\begin{aligned} \gamma_C(m \cup n) &= \gamma_C(m) \cup \langle c \rangle && \text{if } n \rightarrow c \in C_{\langle \rangle} \\ \gamma_C(m) &= m && \text{if there is no such } n \end{aligned}$$

For example, $\gamma_{\{a|b \rightarrow c\}}(\langle a, a, b, c \rangle) = \langle a, c, c \rangle$ and $\gamma_{\{a|a \rightarrow a, b|c|d \rightarrow e\}}(\langle a, b, a, d, c, a \rangle) = \langle a, a, e \rangle$. To assure that γ_C does not have multiple solutions, communications in C should be defined such that left-hand sides are disjoint (e.g. $C = \{a|b \rightarrow c, a|d \rightarrow e\}$ is not allowed).

Axioms for the additional operators are listed in Table 3.

VD	$\nabla_V(\delta) = \delta$
V1'	$\nabla_V(\alpha) = \alpha$ if $\alpha_{\langle \rangle} \in V_{\langle \rangle}$
V2'	$\nabla_V(\alpha) = \delta$ if $\alpha_{\langle \rangle} \notin V_{\langle \rangle}$
V3	$\nabla_V(x + y) = \nabla_V(x) + \nabla_V(y)$
V4	$\nabla_V(x \cdot y) = \nabla_V(x) \cdot \nabla_V(y)$
DD	$\partial_H(\delta) = \delta$
D1	$\partial_H(\alpha) = \alpha$ if $\alpha_{\langle \rangle} \cap H = \emptyset$
D2	$\partial_H(\alpha) = \delta$ if $\alpha_{\langle \rangle} \cap H \neq \emptyset$
D3	$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$
D4	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$
RD	$\rho_R(\delta) = \delta$
R1	$\rho_R(\alpha) = R \bullet \alpha$
R3	$\rho_R(x + y) = \rho_R(x) + \rho_R(y)$
R4	$\rho_R(x \cdot y) = \rho_R(x) \cdot \rho_R(y)$
GD	$\Gamma_C(\delta) = \delta$
G1	$\Gamma_C(\alpha) = \gamma_C(\alpha_{\langle \rangle})_{\langle \rangle}$
G3	$\Gamma_C(x + y) = \Gamma_C(x) + \Gamma_C(y)$
G4	$\Gamma_C(x \cdot y) = \Gamma_C(x) \cdot \Gamma_C(y)$

Table 3: Additional operator axioms

Exercise 2.6 Prove the following propositions:

1. $\partial_{\{b\}}(a \cdot \rho_{\{c \rightarrow a\}}(b|c)) = a \cdot \delta$
2. $\Gamma_{\{a|c \rightarrow d\}}(a|b \parallel c) = a|b \cdot c + c \cdot a|b + b|d$
3. $\partial_{\{s_a, s_b\}}(\Gamma_{\{s_a|s_b \rightarrow s\}}(a \cdot s_a \parallel s_b \cdot b)) = a \cdot s \cdot b$
4. $\Gamma_{\{s_a|s_b \rightarrow s\}}(\nabla_{\{a, b, s_a|s_b\}}(a \cdot s_a \parallel s_b \cdot b)) = a \cdot s \cdot b$
5. $\nabla_{\{a, b, s\}}(\Gamma_{\{s_a|s_b \rightarrow s\}}(a \cdot s_a \parallel s_b \cdot b)) = a \cdot s \cdot b$

Exercise 2.7 Assume we have some component S that communicates via a channel K with an environment. We are only interested in the communication between S and K , so we specify S as follows:

proc $S = r_1 \cdot s_2 \cdot S;$

The communication behaviour of S is specified as repeatedly receiving something over *connection 1* and sending something over *connection 2*. (Note that these *somethings* are not modelled, as they are not relevant here.)

The channel can be specified as follows (only considering the connection to S):

proc $K = s_1 \cdot K + r_2 \cdot K;$

The composition of S and K would then be:

init $\nabla_{\{c_1, c_2\}}(\Gamma_{\{r_1|s_1 \rightarrow c_1, r_2|s_2 \rightarrow c_2\}}(K \parallel S));$

1. Simplify the initial process to a process expression with only basic operators.
2. What happens if the specification of channel K is replaced with the following trace equivalent specification of L , that allows *burst* communication over *connection 1*? (In other words, simplify the system again with L instead of K .)

proc $L = s_1 \cdot L + s_1 \cdot s_1 \cdot L + r_2 \cdot L;$

2.1.5 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 interaction 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 that denotes that some (unknown) internal behaviour happens.
- **Hiding operator** $\tau_I(p)$, which hides (or renames to τ) all actions in I in all multiactions in p .

Because one can only observe external actions, (the effects of) internal actions are only observable if such an action determines a choice. To reflect this in the equivalence on processes we need a somewhat weaker notion of bisimulation. This **branching bisimulation equivalence** differs in the fact that if one process can perform a τ , then the other does not necessarily has to be able to perform a τ as well. In this case, however, it is needed that the τ performed by the first process results in a process that is equivalent to the other. Also, in matching an action (both external and τ) it is allowed to first execute some internal actions. For example, $p = \tau \cdot a$ and $q = a$ are *branching bisimilar* because q can skip the τ of p and p is allowed to first do its τ before matching q 's a .

A somewhat stricter variant named **rooted branching bisimulation equivalence** is needed to make sure that processes are also equal within some context. This equivalence adds a rootedness condition to branching bisimulation equivalence stating that internal actions that can be performed from the initial state of a process should be considered as external actions. So $\tau \cdot a$ and a are not *rooted branching bisimilar*, but $a \cdot \tau \cdot b$ and $a \cdot b$ are.

The axioms are listed in Table 4. For the axioms of the hiding operator we use a function $\theta_I(\alpha)$ which

removes all actions in set I from multiaction α . It is defined by $\theta_I(a|\alpha) = \theta_I(\alpha)$ and $\theta_I(a) = \tau$ if $a \in I$ and $\theta_I(a|\alpha) = a|\theta_I(\alpha)$ and $\theta_I(a) = a$ otherwise. Also, $\theta_I(\tau) = \tau$.

Note that axioms V1' and V2' (from Table 3) are replaced by slightly different versions. Because τ is not observable, it cannot be blocked and will always be allowed. Axiom T2 expresses that a τ can only be removed from a choice if it does not reduce the possible behaviour. When a (first) action of y is executed, we have no way of determining which of the two occurrences of y it was. However, if we have the expression $\tau \cdot x + y$ instead and τ happens, we can observe that it is no longer possible to execute y and thus the executed τ is observable. This also explains why the initial x is required and there is no axiom $\tau \cdot x = x$; without it there could be another alternative in the context. Note that this corresponds to the rootedness condition.

Note that τ is not allowed to occur in sets V, H, R and C . Also, the conversion $\alpha_{\langle \rangle}$ removes any τ that occurs in α (e.g. $(a|\tau|b)_{\langle \rangle} = \langle a, b \rangle$). Finally, we now allow communications in C to be of the form $a|b$ (besides $a|b \rightarrow c$), meaning that a and b communicate to τ .

MA2'	$\alpha \tau = \alpha$
T1	$x \cdot \tau = x$
T2	$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$
TID	$\tau_I(\delta) = \delta$
TI1	$\tau_I(\alpha) = \theta_I(\alpha)$
TI3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$
TI4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$
V1	$\nabla_V(\alpha) = \alpha$ if $\alpha_{\langle \rangle} \in (V \cup \{\tau\})_{\langle \rangle}$
V2	$\nabla_V(\alpha) = \delta$ if $\alpha_{\langle \rangle} \notin (V \cup \{\tau\})_{\langle \rangle}$

Table 4: Abstraction axioms

Exercise 2.8 Prove the following propositions:

1. $\tau_{\{b,c\}}(a \cdot (b + c \cdot b) \cdot d) = a \cdot d$
2. $\tau_{\{b\}}(a|b) = a$

Exercise 2.9 Assume we have components C_a, C_b and C_c that are connected sequentially (i.e. there is a connection between C_a and C_b and a connection between C_b and C_c). They are specified as follows, with the purpose of implementing a leader election protocol:

The protocol works like this: if you have only one neighbour, then you can tell your neighbour that you give up your chance to be leader (with an s_i). If a neighbour tells you that he will no longer participate (via r_i), you no longer consider him to be your neighbour and if he was your only neighbour, you proclaim your leader position (l_i).

act $r_1, r_2, s_1, s_2, l_1, l_2, l_3;$
proc $C_a = s_1 + r_1 \cdot l_1;$
 $C_b = r_1 \cdot (s_2 + r_2 \cdot l_2) + r_2 \cdot (s_1 + r_1 \cdot l_2);$
 $C_c = s_2 + r_2 \cdot l_3;$

init $\tau_{\{c_1, c_2\}}(\partial_{\{r_1, r_2, s_1, s_2\}}(\Gamma_{\{r_1|s_1 \rightarrow c_1, r_2|s_2 \rightarrow c_2\}}(C_a \parallel C_b \parallel C_c)));$

Simplify the initial process.

2.2 Data

The mCRL2 data language is a functional language based on *higher-order abstract data types* [10, 11]. Sorts (types), constructors, functions and their definitions can be declared. For instance, the following declares the sort A with constructors c and d . Also functions f, g and h are declared and (partially) defined:

```

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

```

In the equations *variables* are used to represent unknown data expressions. Note that function types are first-class citizens: functions may return functions.

Sort references can be declared. For instance in

```

sort    B = A;

```

B is a synonym for A . Using sort references it is possible to define recursive sorts (see below).

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

```

var     x, y : A;
eqn     h(x)    = λy':A(λz:Af(z, g(z)))(g(f(x, y')));
          h(x)(y) = f(z, g(z)) whr z = g(f(x, y)) end;

```

Note that the two definitions of h are equivalent.

As mentioned above, mCRL2 also has concrete data types. These consist of *standard data types* and *functions* as well as *type constructors*. For the former, we have the following:

- Booleans (\mathbb{B}) with constants *true*, *false* and operators \neg , \wedge , \vee , \Rightarrow . For all sorts the equality operator \approx , inequality $\not\approx$, conditional *if* and quantifiers \forall and \exists are provided. So for instance the expression $c \approx c$ is equal to *true*, $c \not\approx c$ to *false*, *if* (*true*, c , d) to c , and $\forall_{x:A}.(f(x, c) \approx c)$ to *true* (using the above definition of f). Also, expressions of sort \mathbb{B} may be used as *conditions* in equations, for instance:

```

var     x, y : A;
eqn     x ≈ y → f(x, y) = x;

```

- Unbounded positive (\mathbb{N}^+), natural (\mathbb{N}), integer (\mathbb{Z}) and non-negative real numbers ($\mathbb{R}^{\geq 0}$) with relational operators $<$, \leq , $>$, \geq , unary negation $-$, binary arithmetic operators $+$, $-$, $*$, **div**, **mod** and arithmetic operations *max*, *min*, *abs*, *succ*, *pred*, *exp*. Also conversion functions $A \mathcal{A} B$ are provided for all sorts $A, B \in \{\mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}^{\geq 0}\}$.

There are a number of type constructors, of which the first is a *structured type*. This is a compact way of defining a sort and its distinct constructor functions, together with projection and recogniser functions for these constructors. For instance, a sort MS of machine states can be declared by:

```
sort     $MS = \mathbf{struct}$  off | standby | starting | running | broken;
```

This sort has constructors *off*, *standby*, *starting*, *running* and *broken* and no projection or recogniser functions. Note that the constructors are distinct, so e.g. $off \approx off$ is *true* and $off \approx standby$ is *false*.

A sort of binary trees with numbers as their leaves can be defined as follows:

```
sort     $T = \mathbf{struct}$  leaf(value :  $\mathbb{N}$ )?is_leaf | node(left :  $T$ , right :  $T$ )?is_node;
```

This declares sort T with constructors $leaf : \mathbb{N} \rightarrow T$ and $node : T \times T \rightarrow T$, projection functions $value : T \rightarrow \mathbb{N}$ and $left, right : T \rightarrow T$, and recognisers $is_leaf : T \rightarrow \mathbb{B}$ and $is_node : T \rightarrow \mathbb{B}$. So for example $value(leaf(n)) = n$ and $left(node(t, u)) = t$, and $is_leaf(leaf(n)) = true$ and $is_leaf(node(t, u)) = false$.

We also have a *list* type constructor. The following declares a list containing elements of sort A :

```
sort     $AL = List(A)$ ;
```

This list has constructors $[] : AL$ and $\triangleright : A \times AL \rightarrow AL$. 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 AL 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 $\{a_0, \dots, a_n\}$, bag enumeration $\langle a_0 : c_0, \dots, a_n : c_n \rangle$ (c_i is the multiplicity or count of element a_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.

Integration with the process language

Actions can be *parameterised* with data. For example:

```
act     $a$ ;  
         $b : \mathbb{B}$ ;  
         $c : \mathbb{B} \times \mathbb{N}^+$ ;
```

This declares parameterless action a , action b with a data parameter of sort \mathbb{B} , and action c with two parameters of sort \mathbb{B} and \mathbb{N}^+ respectively. For the above declaration, a , $b(true)$ and $c(false, 6)$ are valid actions.

For some operators this parameterisation involves small changes in the semantics:

- Restriction $\nabla_V(p)$, blocking $\partial_H(p)$ and hiding $\tau_I(p)$ *disregard* the data parameters of the multiactions in p when determining if an (multi)action should be blocked or hidden. For example, $\nabla_{\{b|c\}}(a(0) + b(true, 5)|c) = b(true, 5)|c$ and $\partial_{\{b\}}(a(0) + b(true, 5)|c) = a(0)$.
- Renaming $\rho_R(p)$ also disregards the data parameters, but when a renaming is applied the data parameters are *retained*. For example, $\rho_{\{a \rightarrow b\}}(a(0) + a) = b(0) + b$.

- Communication $\Gamma_C(p)$ has become *stricter*: for each communication $a_0|\dots|a_n \rightarrow c$, $n \geq 1$, multiactions $a_0(\dots)|\dots|a_n(\dots)$ in p are only replaced by $c(\dots)$ when the data parameters of all a_i are equal (both the number of parameters and their values). The data parameters are retained in action c . For example $\Gamma_{\{a|b \rightarrow c\}}(a(0)|b(0)) = c(0)$, but also $\Gamma_{\{a|b \rightarrow c\}}(a(0)|b(1)) = a(0)|b(1)$. Furthermore, $\Gamma_{\{a|b \rightarrow c\}}(a(1)|a(0)|b(1)) = a(0)|c(1)$.

This requires some small changes in the axioms for these operators. The new axioms can be found in Appendix C.

Next to actions, process references can be parameterised. For example:

$$\begin{aligned} \text{proc } P(d : \mathbb{B}, e : \mathbb{N}^+) &= a \cdot P(d, e) \\ &\quad + b(d) \cdot P(\neg d, e + 1) \\ &\quad + c(d, e) \cdot P(\text{false}, \text{max}(e - 1, 1)); \end{aligned}$$

This declares the process P with data parameters d and e of sort \mathbb{B} and \mathbb{N}^+ , respectively. Note that the sorts of the data parameters are declared in the left-hand side of the equation. In the process references on the right-hand side the *values* of the data parameters are specified.

Data can influence process behaviour by means of a **conditional** operator, written as $c \rightarrow p \diamond q$, 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: $c \rightarrow p$ is a valid expression that behaves as $c \rightarrow p \diamond \delta$. The operator binds stronger than \parallel and $\llbracket \cdot \rrbracket$, but weaker than \cdot . The corresponding axioms for the conditional operator are given in Table 5.

C1	$\text{true} \rightarrow x \diamond y = x$
C2	$\text{false} \rightarrow x \diamond y = y$
C3	$c \rightarrow x = c \rightarrow x \diamond \delta$

Table 5: Conditional axioms

Exercise 2.10 Prove the following propositions:

1. $c \rightarrow p \diamond q = c \rightarrow p + \neg c \rightarrow q$
2. $(c \rightarrow p \diamond q) \cdot r = c \rightarrow p \cdot r \diamond q \cdot r$

We also extend process expressions with the ability to *quantify* over data types. For this we introduce the **summation** operator $\sum_{d:D} p$ where p is a process expression in which data variable d may occur. The corresponding behaviour is $p[d_0/d] + \dots + p[d_n/d]$, $n \geq 0$, for all elements $d_i \in D$. Here, $p[d_i/d]$ stands for p in which each free occurrence of d (i.e. not bound by another $\sum_{d:D}$) is replaced by d_i . Summation has the lowest precedence after $+$.

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 .

$$\begin{aligned} \text{act } r, s : \mathbb{N}; \\ \text{proc } \text{Buffer} &= \sum_{n:\mathbb{N}} r(n) \cdot s(n) \cdot \text{Buffer}; \end{aligned}$$

The axioms for summation are given in Table 6. Here X and Y are *function variables* from the data type D to process expressions.

SUM1	$\sum_{d:D} x = x$
SUM3	$\sum_{d:D} X(d) = \sum_{d:D} X(d) + X(e)$ with $e \in D$
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	$(\sum_{d:D} X(d)) \parallel x = \sum_{d:D} X(d) \parallel x$
SUM7	$(\sum_{d:D} X(d)) x = \sum_{d:D} X(d) x$
SUM7'	$x \sum_{d:D} X(d) = \sum_{d:D} x X(d)$
SUM11	$\sum_{d:D} X(d) = \sum_{d:D} Y(d)$, if $X(d) = Y(d)$ for all $d \in D$
V6	$\nabla_V(\sum_{d:D} X(d)) = \sum_{d:D} \nabla_V(X(d))$
D6	$\partial_H(\sum_{d:D} X(d)) = \sum_{d:D} \partial_H(X(d))$
R6	$\rho_R(\sum_{d:D} X(d)) = \sum_{d:D} \rho_R(X(d))$
G6	$\Gamma_C(\sum_{d:D} X(d)) = \sum_{d:D} \Gamma_C(X(d))$
TI6	$\tau_I(\sum_{d:D} X(d)) = \sum_{d:D} \tau_I(X(d))$

Table 6: Summation axioms

Note that \sum acts as a binder. In the axioms, a process variable x that is in the scope of a $\sum_{d:D}$ may only be instantiated with process expressions in which data variable d does not occur freely. For example, in SUM1 x may be instantiated with $a(1)$, but not with $a(d)$. However, using function variables we *are* able to bind variables, i.e. we can instantiate X with $\lambda_{d':D} p^2$ for any process expression p such that $X(d)$ becomes $p[d/d']$. As in the λ -calculus we allow α -conversion (renaming of bound variables), i.e. $\sum_{d:D} X(d) = \sum_{d':D} X(d')$.

Exercise 2.11 Prove the following propositions, where p, q are process expressions in which variable b of sort \mathbb{B} does not occur freely, e is a data expression of sort D , and F is a function from D to process expressions:

1. $\sum_{b:\mathbb{B}} b \rightarrow p \diamond q = p + q$
2. $\sum_{d:D} (d \approx e) \rightarrow F(d) = F(e)$ (sum elimination lemma)

Hint: use anti-symmetry of \leq (see Exercise 2.3.3).

2.3 Time

Time can be added to processes using the **at** operator \circlearrowleft . The expression $\alpha \circlearrowleft t$ indicates that multiaction α happens at time t , where t is a data expression of sort $\mathbb{R}^{\geq 0}$ and $t \geq 0$. This notion is extended to arbitrary process expressions as follows: in $p \circlearrowleft t$ the first multiactions of p happen at time t . The operator has higher precedence than \cdot , but lower than $|$.

We do not give the full details of this operator, because it is beyond the scope of this document. Instead, we give a few examples. To start with, we specify a simple clock:

act $tick;$
proc $C(t : \mathbb{R}^{\geq 0}) = tick \circlearrowleft t \cdot C(t + 1);$

For a value u of sort $\mathbb{R}^{\geq 0}$, the process $C(u)$ exhibits the single infinite trace $tick \circlearrowleft u \cdot tick \circlearrowleft (u + 1) \cdot tick \circlearrowleft (u + 2) \cdot \dots$.

²The λ in $\lambda_{d':D} p$ is not part of the actual language; it lives at the meta-level.

To make the behaviour a bit more interesting, we add a timeout and the possibility to reset the clock:

act *reset*;
proc $TRC(t : \mathbb{R}^{\geq 0}) = (t < 1000) \rightarrow tick^t \cdot TRC(t + 1)$
 $+ reset \cdot TRC(0);$

This clock can increment its counter by 1 consecutively for at most a thousand times, while at any moment can reset the counter to 0.

As a different example, we show a model of a *drifting* clock (taken from [17]). This is a clock that is accurate within a bounded interval $[1 - \mathfrak{d}, 1 + \mathfrak{d}]$, where $\mathfrak{d} < 1$.

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

3 Linear process specifications

For manipulating processes – either by hand or using tools – it is useful to transform them to a basic form. This basic form is called a **linear process specification** or LPS for short. The most important characteristics of a linear process are that there is one single equation and that there is precisely one action in front of the recursive invocation of the process variable at the right-hand side:

Definition 3.1 A linear process specification (LPS) is a process specification of which the process part is of the form

proc $P(d:D) = \sum_{i \in I} \sum_{e_i : E_i} c_i(d, e_i) \rightarrow a_i(f_i(d, e_i)) \cdot t_i(d, e_i) \cdot P(g_i(d, e_i));$

where I a finite index set, c_i a condition, f_i the parameter of action a_i occurring at time t_i and g_i the next state.³ Note that the first summand is a *meta-level* operation: $\sum_{i \in I} p_i$ is a shorthand for $p_1 + \dots + p_n$, where n is the size of I .

We call data parameter d the **state** parameter. In general we only strictly adhere to the form above with one state parameter d and one sum variable e_i per summand in theoretical considerations. In practice we can use any number or leave out both. Also, if the condition is *true* it is usually left out.

The form as described above is sometimes described as the *condition-action-effect* rule. In a particular state d the action a_i can be done at time t_i if condition c_i holds. The effect of the action is given by the function g_i .

Example 3.2 The process *Buffer* on page 14 is *not* linear because there are two actions in front of the reference to *Buffer* in the right-hand side of the process definition. The LPS for the buffer has the following form:

proc $P(n:\mathbb{N}, b:\mathbb{B}) = \sum_{m:\mathbb{N}} b \rightarrow r(m) \cdot P(m, \neg b)$
 $+ \neg b \rightarrow s(n) \cdot P(n, \neg b);$

init $P(0, true);$

Note that the linear form is less readable than the much more concise form we started with.

³The general form of an LPS also contains terminating summands and multiactions. We do not include these here because they only add complexity and provide no essential insight in the concept.

LPSs can be seen as a (symbolic) representation of the state space of a model. The state space often has a number of states that is exponential in the number of parallel processes. The fact that many process descriptions lead to systems with a huge number of states is commonly known as the *state space explosion problem*. Because of this, it often takes a large amount of time to generate a state space and a large amount of space to store it.

In general, it is relatively straightforward to transform processes to linear form. For two parallel processes it generally boils down to joining the state variables and concatenating the summands. In [15] it is described how this can be done for all timed μ CRL processes. Using these techniques, a model of hundreds of parallel processes can be transformed to a single LPS relatively easily and within a small amount of time; even if the corresponding state space is infinitely large.

Example 3.3 Consider the following process equations that describe two buffers in sequence. The first one reads from channel 1 and delivers at channel 2. The second one reads from 2 and sends to 3. The subsequent (non-linear) equation defines a system as the parallel composition of both processes where they pass the value on via channel 2:

$$\begin{aligned}
\text{proc } P_{12}(n:\mathbb{N}, b:\mathbb{B}) &= \sum_{m:\mathbb{N}} b \rightarrow r_1(m) \cdot P_{12}(m, \neg b) \\
&\quad + \neg b \rightarrow s_2(n) \cdot P_{12}(n, \neg b); \\
P_{23}(n:\mathbb{N}, b:\mathbb{B}) &= \sum_{m:\mathbb{N}} b \rightarrow r_2(m) \cdot P_{23}(m, \neg b) \\
&\quad + \neg b \rightarrow s_3(n) \cdot P_{23}(n, \neg b); \\
\text{System} &= \nabla_{\{r_1, s_3, c_2\}} (\Gamma_{\{r_2 | s_2 \rightarrow c_2\}} (P_{12}(0, \text{true}) \parallel P_{23}(0, \text{true})));
\end{aligned}$$

The process *System* behaves exactly the same as the process $P_{13}(0, \text{true}, 0, \text{true})$ defined by the following LPS that has been derived from the three equations above:

$$\begin{aligned}
\text{proc } P_{13}(n_1:\mathbb{N}, b_1:\mathbb{B}, n_2:\mathbb{N}, b_2:\mathbb{B}) &= \sum_{m:\mathbb{N}} b_1 \rightarrow r_1(m) \cdot P_{13}(m, \neg b_1, n_2, b_2) \\
&\quad + (\neg b_1 \wedge b_2) \rightarrow c_2(n_1) \cdot P_{13}(n_1, \neg b_1, n_1, \neg b_2) \\
&\quad + \neg b_2 \rightarrow s_3(n_2) \cdot P_{13}(n_1, b_1, n_2, \neg b_2);
\end{aligned}$$

Exercise 3.4 Give an LPS that is behaviourally equivalent to the process P defined by $P = a \cdot b \cdot c \cdot P$.

Exercise 3.5 Give an LPS that is behaviourally equivalent to the process P defined by $P = a \cdot (P + Q)$ and $Q = b \cdot Q$.

Exercise 3.6 Consider the process $P = a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8 \cdot a_9 \cdot a_{10} \cdot P$. How many summands does an LPS with the same behaviour as

$$\nabla_{\{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}\}} (P \parallel P \parallel P \parallel P \parallel P \parallel P \parallel P \parallel P \parallel P \parallel P)$$

have? How many states does this process have?

4 mCRL2 toolset

We introduce the tools found in the current distribution of the mCRL2 toolset. For each tool we sketch how it can be used and what for. Tool names are typeset in teletype as are command line options to tools.

Many tools in the mCRL2 toolset offer a variety of options that can be used to change the behaviour of such a tool. In general the `--help` command line option can be used to view a compact description of available options.

4.1 Linearisation

As depicted in Figure 1 on page 2, analysis starts with a model of a system expressed in the mCRL2 language. The next step is to generate an LPS for the model. This is accomplished by using the tool `mcr122lps`. It also provides a number of useful options that may prove useful during the case study later on.

4.2 LPS manipulation

The following tools can be used to manipulate LPSs:

- `lpsinfo`: print basic information about an LPS;
- `lpspp`: pretty print an LPS;
- `lpsconstelm`: eliminate process parameters which are constant in the reachable state space;
- `lpsparelm`: eliminate unused parameters from an LPS;
- `lpsdataelm`: remove unused parts of the data specification of an LPS;
- `lpsrewr`: evaluate conditions through rewriting and remove unused alternatives.

Most of these tools perform either transformation or conversion operations except for `lpsinfo` and `lpspp`. The other tools are useful for simplifying LPSs, which can have a substantial effect on the final size of the state space, or the time/space needed for generation.

To get some more feeling for the kind of transformations that these tools perform consider the following example.

Example 4.1 Consider the following LPS:

```
act    a;  
        c :  $\mathbb{B}$ ;  
proc    $P(b:\mathbb{B}, x:\mathbb{N}, y:\mathbb{N}, z:\mathbb{N}) = a \cdot P(\text{false}, 0, 0, y + z)$   
         $+ (y \neq z \vee \neg b) \rightarrow c(b) \cdot P(\text{true}, y, y, y + z + 1)$ ;  
init    $P(\text{false}, 0, 0, 0)$ ;
```

The tool `lpsparelm` eliminates process parameter x , while `lpsconstelm` also eliminates y .

4.3 Simulation

Using simulation a model can be explored without generating the state space. It is used to get more intuition about the possible behaviour of the model under scrutiny. Using the interactive simulation tools `sim` (command line) and `xsim` (GUI), a user can explore a model (expressed as an LPS) by manually performing individual (multi)actions. After an action is performed the simulator shows a representation of the new state (a state vector), and the actions that are possible from this state.

A trace is a single run through the system, in this case explicitly specified by the user. The simulator can load and store traces at any time. Traces are important because they can show presence of behaviour and as such can be used as a proof or counter example. The tool `tracepp` can be used to pretty print traces.

4.4 State space generation

To generate a state space from an LPS the tool `lps2lts` can be used. The state space contains the behaviour of a model. It can be stored in two different file formats: SVC and Aldebaran (AUT). The Aldebaran format uses plain text which may result in large files. SVC files are more compressed but they require more time to generate. To use the visualisation tools introduced in the next section the SVC format should be used.

The tool `ltsconvert` can be used to convert between different formats of LTSes. The tool `ltsmin` can be used to minimise the state space modulo (branching) bisimulation equivalence.

4.5 Visualisation

Visualisation of a state space can help to provide insight in model behaviour. For instance, it can be used to trace errors in the specification of a model. But it can also provide means to gain insight into the *structure* of the behaviour and to identify *symmetries*.

A number of visualisation tools are available for use. The most straightforward visualisation of a state space is a direct visualisation as a labelled directed graph. The tool `ltsconvert` with the `-o dot` option can be used for this purpose. Other available visualisation tools are `NoodleView` (based on the `StateVis` tool [13]) and `FSMView` [16], for 2D and 3D visualisation respectively. These tools use clustering techniques, which allows them to be used effectively on large state spaces. Both tools operate on state spaces in FSM format, which can be obtained from an SVC file using `ltsconvert`.

4.6 Tool overview

Table 7 is included for easy reference of what tools are available.

Name	Description
<code>mcrl22lps</code>	generate an LPS of an mCRL2 specification
<code>lpsinfo</code>	print some basic information about an LPS
<code>lpspp</code>	pretty print an LPS
<code>lpsconstelm</code>	remove constant process parameters from an LPS
<code>lpsparelm</code>	remove unused process parameters from an LPS
<code>lpsdataelm</code>	remove unnecessary parts of the data specification of an LPS
<code>lpsrewr</code>	remove unused alternatives in conditions of an LPS
<code>sim/xsim</code>	manually explore the behaviour of an LPS
<code>tracepp</code>	pretty print a trace
<code>lps2lts</code>	generate the state space for an LPS
<code>ltsconvert</code>	convert an LTS to a different format
<code>ltsmin</code>	minimise an LTS
<code>NoodleView/FSMView</code>	visualise a state space through clustering in 2D/3D

Table 7: Quick reference of tools and their functionality

5 Case study

At ASML, the world leading wafer stepper manufacturer situated in Veldhoven (near Eindhoven), a new generation of wafer steppers is currently under construction. Wafers are being used to produce integrated circuits (ICs). Transistors and other components are etched onto a wafer using a mask and

a light beam. The components on ICs become increasingly smaller. This leads to the need for light with a higher frequency, i.e. in the ultraviolet bandwidth. But as the atmosphere absorbs ultraviolet light, the whole process must take place in vacuum.

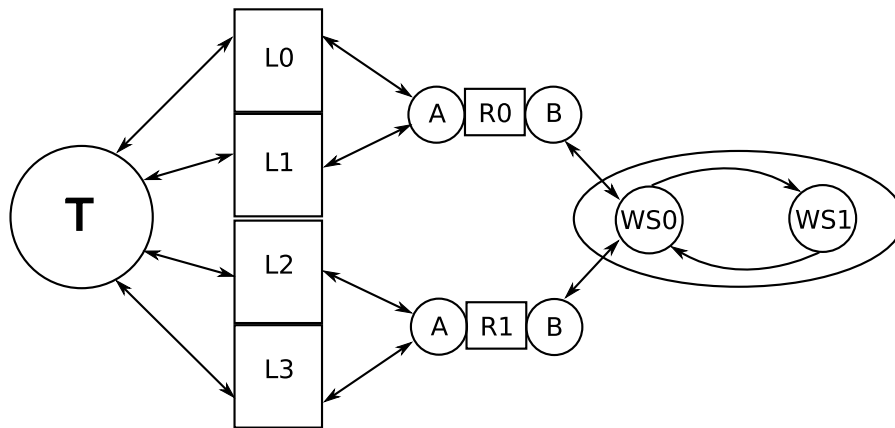


Figure 5: Layout of the wafer stepper

In Figure 5 a layout of a wafer stepper is shown. Fresh wafers enter the system via the tray (T). They must travel to one of the load locks (L0-3) that have a capacity of one wafer each. The load locks form the bridge between atmospheric pressure outside (tray-side) and the vacuum inside the system. There they wait to be transported by a robot (R0-1). Robot R0 transports wafers from and to locks L0 and L1, R1 does the same for locks L2 and L3. Both robots have two arms A and B that can pick up and hold a single wafer each. The arms are mounted on opposite sides of the robot such that arm A faces the locks when arm B faces a wafer stage (WS0-1) and vice versa. To transport wafers a robot just turns around its axis. At wafer stage WS0 a wafer is measured. Once it has been measured a wafer is exposed at wafer stage WS1, after which the wafer is ready. Both wafer stages can only accommodate a single wafer and wafers are moved by simultaneously swapping the contents of WS0 and WS1. Finally, if a wafer is ready, it leaves the system via the same path through one of the load locks back to the tray.

Exercise 5.1 Show that the system without scheduling constraints contains deadlocks.

By limiting the amount of wafers that can be in certain parts of the system it is possible to remove deadlocks. For instance, the number of fresh wafers that is in a lock at the same time can be limited to three. Note that this particular constraint does not remove deadlocks.

Exercise 5.2 Add appropriate constraints that prevent from deadlock yet still allow all fresh wafers that are initially on the tray to be exposed and leave the system.

References

- [1] J.C.M. Baeten, W.P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.
- [2] W. Fokkink, J.F. Groote, J. Pang, B. Badban and J.C. van de Pol, "Verifying a sliding window protocol in μCRL ", *Proc. 10th Int'l Conf. Algebraic Methodology and Software Technology*, C. Rattray et al., eds., LNCS 3116, Springer Verlag, 2004, pp. 148-163.

- [3] J.F. Groote, “The syntax and semantics of timed μCRL ”, Technical report SEN-R9709, CWI, Amsterdam, 1997.
- [4] J.F. Groote, A. Mathijssen, M. van Weerdenburg and Y. Usenko, “From μCRL to mCRL2: motivation and outline”, *Proc. Workshop Algebraic Process Calculi: The First Twenty Five Years and Beyond*, BRICS NS-05-3, 2005, pp. 126-131.
- [5] J.F. Groote, J. Pang and A.G. Wouters, “Analysis of a distributed system for lifting trucks”, *Journal of Logic and Algebraic Programming*, vol. 55, no. 1-2, 2003, pp. 21-56.
- [6] J.F. Groote and A. Ponse, “The syntax and semantics of μCRL ”, *Algebra of Communicating Processes, Workshops in Computing*, A. Ponse, et al., eds., 1994, pp. 26-62.
- [7] J.F. Groote and M. Reniers, “Algebraic process verification”, *Handbook of Process Algebra*, J.A. Bergstra et al., eds., Elsevier Science, 2001, pp. 1151-1208.
- [8] J.F. Groote and J.J. van Wamel, “Algebraic Data Types and Induction in μCRL ”, Technical Report P9409, University of Amsterdam, Amsterdam, 1994.
- [9] mCRL2 Toolset, www.mcl2.org.
- [10] K. Meinke, “Higher-Order Equational Logic for Specification, Simulation and Testing”, *The 1995 Workshop on Higher-Order Algebra, Logic and Term Rewriting (HOA '95)*, LNCS 1074, Springer, 1996, pp. 124-143.
- [11] B. Möller, A. Tarlecki and M. Wirsing, “Algebraic Specification of Reachable Higher-Order Algebras”, *Recent Trends in Data Type Specification*, LNCS 332, Springer, 1988, pp. 154-169.
- [12] J. Pang, W. Fokkink, R. Hofman and R. Veldema, “Model Checking a Cache Coherence Protocol for a Java DSM Implementation”, *Proc. 2003 International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, IEEE Computer Society Press, 2003.
- [13] A.J. Pretorius and J.J. van Wijk, “Multidimensional Visualization of Transition Systems”, *Proc. 9th Int'l Conf. Information Visualization (IV05)*, London, IEEE CS Press, 2005, pp. 323-328.
- [14] F. Stockton, “The Lady, or the Tiger?”, *An Anthology of Famous American Stories*, New York, Modern Library, 1953, pp. 248-253.
- [15] Y.S. Usenko, *Linearization in μCRL* , PhD thesis, Eindhoven, 2002.
- [16] F. van Ham, H. van de Wetering and J.J. van Wijk, “Interactive visualization of state transition systems”, *IEEE Trans. Visualization and Computer Graphics*, vol. 8, no. 3, 2002, pp. 319-329.
- [17] T.A.C. Willemse, *Semantics and Verification in Process Algebras with Data and Timing*, PhD thesis, Eindhoven, 2003.

A Formal syntax

The following describes the formal syntax of the mCRL2 language. It is given in a rich text format for readability. In Appendix B a translation of rich text to plain text is given, which is needed for using the toolset. In both formats a %-sign indicates the beginning of a comment that extends to the end of the line.

In the following, b stands for a sort name, f for a function name, x for a data variable name, a for an action name and P for a process variable name. They are all strings matching the pattern “[$a-z$][$a-z0-9$]*”. N stands for a number that matches the pattern “[$1-9$][$0-9$]*”. Suggestive dots (\dots , \dots) are used to indicate repeating patterns with one or more occurrence. Furthermore, $|$ distinguishes alternatives (not to be mistaken with the sync operator $|$), $(pattern)^+$ indicates one or more occurrences of $pattern$, and $(pattern)^*$ indicates zero or more occurrences of $pattern$. As opposed to real EBNF, we do not use quotes to separate the terminals from the non-terminals.

Sort expressions s :

$$\begin{aligned} s & ::= b \mid s \rightarrow s \mid \mathbb{B} \mid \mathbb{N}^+ \mid \mathbb{N} \mid \mathbb{Z} \mid \mathbb{R}^{\geq 0} \mid \mathbf{struct} \ scs \mid \dots \mid scs \mid List(s) \mid Set(s) \mid Bag(s) \\ scs & ::= f \mid f(spj, \dots, spj) \mid f?f \mid (spj, \dots, spj)?f \\ spj & ::= s \mid s \times \dots \times s \rightarrow s \mid f : s \mid f : s \times \dots \times s \rightarrow s \end{aligned}$$

Here scs and spj stand for the constructors and the projection functions of a structured sort.

Declarations:

$$\begin{aligned} sd & ::= b \mid b = s \\ fd & ::= f, \dots, f : s \\ vd & ::= x, \dots, x : s \\ ed & ::= d = d \mid c \rightarrow d = d \\ ad & ::= a \mid a : s \times \dots \times s \\ pvd & ::= P \mid P(vd, \dots, vd) \end{aligned}$$

Here, sd stands for sort declaration, fd for function declaration, vd for data variable declaration, ed for equation declaration, ad for action declaration, pvd for process variable declaration, d for a data expression and c for a data expression of sort \mathbb{B} .

Data expressions d :

$$\begin{aligned} d & ::= x \mid f(d, \dots, d) \mid d(d) \mid N \mid \neg d \mid -d \mid \bar{d} \mid \#d \mid d \oplus d \\ & \quad \mid [] \mid [d, \dots, d] \mid \{ \} \mid \{d, \dots, d\} \mid \{d : d, \dots, d : d\} \mid \{x : s \mid d\} \\ & \quad \mid \lambda_{vd, \dots, vd} d \mid \forall_{vd, \dots, vd} d \mid \exists_{vd, \dots, vd} d \mid d \mathbf{whr} \ x = d, \dots, x = d \mathbf{end} \\ \oplus & ::= * \mid \cdot \mid \cap \mid \mathbf{div} \mid \mathbf{mod} \mid + \mid - \mid \cup \mid ++ \mid \triangleleft \mid \triangleright \\ & \quad \mid < \mid \leq \mid > \mid \geq \mid \subset \mid \subseteq \mid \in \mid \approx \mid \not\approx \mid \wedge \mid \vee \mid \Rightarrow \end{aligned}$$

The unary operators have the highest priority, followed by the infix operators \oplus , followed by λ , \forall and \exists , followed by $\mathbf{whr} \mathbf{end}$. The descending order of precedence of the infix operators is: $\{*, \cdot, \cap\}$, $\{\mathbf{div}, \mathbf{mod}\}$, $\{+, -, \cup\}$, $\{++, \triangleleft, \triangleright\}$, $\{<, \leq, >, \geq, \subset, \subseteq, \in\}$, $\{\approx, \not\approx\}$, $\{\wedge, \vee\}$, \Rightarrow . Of these operators $*, \cdot, \cap$, \mathbf{div} , \mathbf{mod} , $+$, $-$, \cup and $++$ associate to the left and \approx , $\not\approx$, \wedge , \vee and \Rightarrow associate to the right.

Process expressions p :

$$\begin{aligned} p & ::= a \mid \delta \mid \tau \mid p + p \mid p \cdot p \mid P \mid p|p \mid p \parallel p \mid p \parallel p \\ & \quad \mid \nabla_{\{as, \dots, as\}}(p) \mid \partial_{\{a, \dots, a\}}(p) \mid \tau_{\{a, \dots, a\}}(p) \mid \rho_{\{ar, \dots, ar\}}(p) \mid \Gamma_{\{ac, \dots, ac\}}(p) \\ & \quad \mid a(d, \dots, d) \mid P(d, \dots, d) \mid c \rightarrow p \diamond p \mid c \rightarrow p \mid \sum_{vd, \dots, vd} p \mid p^c t \\ as & ::= a \mid \dots \mid a \\ ar & ::= a \rightarrow a \\ ac & ::= a \mid as \mid a \mid as \rightarrow a \end{aligned}$$

Here, c and t stand for data expressions of sort \mathbb{B} and $\mathbb{R}^{\geq 0}$, respectively. as represents an action sequence, ar an action renaming, and ac an action communication. The descending order of precedence of the operators is: $|, \epsilon, \cdot, \rightarrow, \{ \parallel, \ll \}, \sum, +$. Of these operators $+$, \parallel , \ll , \cdot and $|$ associate to the right.

Specification elements:

$$\begin{aligned}
 spec_elt ::= & \mathbf{sort} (sd;)^+ \\
 & | \mathbf{cons} (fd;)^+ \\
 & | \mathbf{map} (fd;)^+ \\
 & | \mathbf{var} (vd;)^+ \mathbf{eqn} (ed;)^+ \\
 & | \mathbf{eqn} (ed;)^+ \\
 & | \mathbf{act} (ad;)^+ \\
 & | \mathbf{proc} (pvd = p;)^+
 \end{aligned}$$

Specification:

$$spec ::= (spec_elt)^* \mathbf{init} p; (spec_elt)^*$$

B Table of mCRL2 symbols

In the toolset, a plain text format is used as opposed to the rich text format of section 2. A mapping from rich text to plain text symbols is provided in Table 8.

Symbol	<i>Rich</i>	Plain
arrow	\rightarrow	->
cross	\times	#
diamond	\diamond	<>
standard sorts	$\mathbb{B}, \mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}^{\geq 0}$	Bool,Pos,Nat,Int,Time
equality and inequality	$\approx, \not\approx$	==,!=
logical operators	$\neg, \wedge, \vee, \Rightarrow$!,&&, ,=>
relational numeric operators	\leq, \geq	<=,>=
relational set operators	\in, \subseteq, \subset	in,<=,<
set operators	\neg, \cup, \cap	!,+,*
list operators	$\triangleright, \triangleleft, ++$	>,< ,++
lambda abstraction	$\lambda_{x:s}d$	lambda x:s.d
universal quantification	$\forall_{x:s}d$	forall x:s.d
existential quantification	$\exists_{x:s}d$	exists x:s.d
deadlock	δ	delta
internal action	τ	tau
left merge	\parallel	-
at	\cdot	@
sum	$\sum_{x:s}p$	sum x:s.p
allow	$\nabla_{\{a b\}}(p)$	allow({a b},p)
block	$\partial_{\{a\}}(p)$	block({a},p)
hide	$\tau_{\{a\}}(p)$	hide({a},p)
rename	$\rho_{\{a \rightarrow b\}}(p)$	rename({a -> b},p)
communication	$\Gamma_{\{a b \rightarrow c\}}(p)$	comm({a b -> c},p)

Table 8: Mapping from rich to plain text

C Axioms

In Table 9 all axioms for (untimed) mCRL2 are given. Table 10 contains the auxiliary functions used in these axioms.

MA2' $\alpha \tau = \alpha$	CM1 $x \parallel y = x \parallel y + y \parallel x + x y$
MA3' $\alpha \beta = \beta \alpha$	CM2 $\alpha_\delta \parallel x = \alpha_\delta \cdot x$
MA4 $\alpha (\beta \gamma) = (\alpha \beta) \gamma$	CM3 $\alpha_\delta \cdot x \parallel y = \alpha_\delta \cdot (x \parallel y)$
A1 $x + y = y + x$	CM4 $(x + y) \parallel z = x \parallel z + y \parallel z$
A2 $x + (y + z) = (x + y) + z$	CM5 $(\alpha_\delta \cdot x) \beta_\delta = \alpha_\delta \beta_\delta \cdot x$
A3 $x + x = x$	CM6 $\alpha_\delta (\beta_\delta \cdot x) = \alpha_\delta \beta_\delta \cdot x$
A4 $(x + y) \cdot z = x \cdot z + y \cdot z$	CM7 $(\alpha_\delta \cdot x) (\beta_\delta \cdot y) = \alpha_\delta \beta_\delta \cdot (x \parallel y)$
A5 $(x \cdot y) \cdot z = x \cdot (y \cdot z)$	CM8 $(x + y) z = x z + y z$
A6 $x + \delta = x$	CM9 $x (y + z) = x y + x z$
A7 $\delta \cdot x = \delta$	CD1 $\delta \alpha_\delta = \delta$
T1 $x \cdot \tau = x$	CD2 $\alpha_\delta \delta = \delta$
T2 $x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$	
VD $\nabla_V(\delta) = \delta$	DD $\partial_H(\delta) = \delta$
V1 $\nabla_V(\alpha) = \alpha$ if $\mu(\alpha_{\langle \rangle}) \in (V \cup \{\tau\})_{\langle \rangle}$	D1 $\partial_H(\alpha) = \alpha$ if $\mu(\alpha_{\langle \rangle}) \cap H = \emptyset$
V2 $\nabla_V(\alpha) = \delta$ if $\mu(\alpha_{\langle \rangle}) \notin (V \cup \{\tau\})_{\langle \rangle}$	D2 $\partial_H(\alpha) = \delta$ if $\mu(\alpha_{\langle \rangle}) \cap H \neq \emptyset$
V3 $\nabla_V(x + y) = \nabla_V(x) + \nabla_V(y)$	D3 $\partial_H(x + y) = \partial_H(x) + \partial_H(y)$
V4 $\nabla_V(x \cdot y) = \nabla_V(x) \cdot \nabla_V(y)$	D4 $\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$
V6 $\nabla_V(\sum_{d:D} x) = \sum_{d:D} \nabla_V(x)$	D6 $\partial_H(\sum_{d:D} x) = \sum_{d:D} \partial_H(x)$
RD $\rho_R(\delta) = \delta$	TID $\tau_I(\delta) = \delta$
R1 $\rho_R(\alpha) = R \bullet \alpha$	TI1 $\tau_I(\alpha) = \theta_I(\alpha)$
R3 $\rho_R(x + y) = \rho_R(x) + \rho_R(y)$	TI3 $\tau_I(x + y) = \tau_I(x) + \tau_I(y)$
R4 $\rho_R(x \cdot y) = \rho_R(x) \cdot \rho_R(y)$	TI4 $\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$
R6 $\rho_R(\sum_{d:D} x) = \sum_{d:D} \rho_R(x)$	TI6 $\tau_I(\sum_{d:D} x) = \sum_{d:D} \tau_I(x)$
GD $\Gamma_C(\delta) = \delta$	SUM1 $\sum_{d:D} x = x$
G1 $\Gamma_C(\alpha) = \gamma_C(\alpha_{\langle \rangle}) $	SUM3 $\sum_{d:D} X(d) = \sum_{d:D} X(d) + X(e)$ with $e \in D$
G3 $\Gamma_C(x + y) = \Gamma_C(x) + \Gamma_C(y)$	SUM4 $\sum_{d:D} (X(d) + Y(d)) = \sum_{d:D} X(d) + \sum_{d:D} Y(d)$
G4 $\Gamma_C(x \cdot y) = \Gamma_C(x) \cdot \Gamma_C(y)$	SUM5 $(\sum_{d:D} X(d)) \cdot x = \sum_{d:D} X(d) \cdot x$
G6 $\Gamma_C(\sum_{d:D} x) = \sum_{d:D} \Gamma_C(x)$	SUM6 $(\sum_{d:D} X(d)) \parallel x = \sum_{d:D} X(d) \parallel x$
C1 $true \rightarrow x \diamond y = x$	SUM7 $(\sum_{d:D} X(d)) x = \sum_{d:D} X(d) x$
C2 $false \rightarrow x \diamond y = y$	SUM7' $x (\sum_{d:D} X(d)) = \sum_{d:D} x X(d)$
C3 $c \rightarrow x = c \rightarrow x \diamond \delta$	SUM11 $\sum_{d:D} X(d) = \sum_{d:D} Y(d)$, if $X(d) = Y(d)$ for all $d \in D$

Table 9: Axioms for (untimed) mCRL2

$\theta_I(\tau)$	=	τ	
$\theta_I(a(\dots))$	=	τ	if $a \in I$
$\theta_I(a(\dots))$	=	$a(\dots)$	if $a \notin I$
$\theta_I(a(\dots) \alpha)$	=	$\theta_I(\alpha)$	if $a \in I$
$\theta_I(a(\dots) \alpha)$	=	$a(\dots) \theta_I(\alpha)$	if $a \notin I$
$R \bullet \tau$	=	τ	
$R \bullet a(\dots)$	=	$R(a)(\dots)$	
$R \bullet a(\dots) \alpha$	=	$R(a)(\dots) (R \bullet \alpha)$	
$\langle a_0 \dots a_n \rangle_{\langle \rangle}$	=	$\langle a_0, \dots, a_n \rangle \setminus \{ \tau \}$	(Note: $\langle a, \tau, \tau \rangle \setminus \{ \tau \} = \langle a \rangle$)
$\{ \alpha_0, \dots, \alpha_n \}_{\langle \rangle}$	=	$\{ \alpha_0_{\langle \rangle}, \dots, \alpha_n_{\langle \rangle} \}$	
$\{ \alpha_0 \rightarrow c_0, \dots, \alpha_n \rightarrow c_n \}_{\langle \rangle}$	=	$\{ \alpha_0_{\langle \rangle} \rightarrow c_0, \dots, \alpha_n_{\langle \rangle} \rightarrow c_n \}$	
$\langle \rangle_{ }$	=	τ	
$\langle a_0, \dots, a_n \rangle_{ }$	=	$a_0 \dots a_n$	
$\mu(\langle \rangle)$	=	$\langle \rangle$	
$\mu(\langle a(\dots) \rangle \cup m)$	=	$\langle a \rangle \cup \mu(m)$	
$\gamma_C(m \cup n)$	=	$\gamma_C(m) \cup \langle c(d_1, \dots, d_n) \rangle$	if $\mu(n) \rightarrow c \in C_{\langle \rangle}$ and the data parameters of all actions in n have the same data arguments d_1, \dots, d_n
$\gamma_C(m \cup n)$	=	$\gamma_C(m)$	if $\mu(n) \in C_{\langle \rangle}$ and the data parameters of all actions in n have the same data arguments d_1, \dots, d_n
$\gamma_C(m)$	=	m	if there is no such n

Table 10: Auxiliary functions for the axioms