The Formal Specification Language mCRL2

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

Technische Universiteit Eindhoven (TU/e)
P.O. Box 513, NL-5600 MB Eindhoven
The Netherlands
{J.F.Groote, A.H.J.Mathijssen, M.A.Reniers,
Y.S.Usenko, M.J.van.Weerdenburg}@tue.nl

Abstract. We introduce mCRL2, a specification language that can be used to specify and analyse the behaviour of distributed systems. This language is the successor of the μCRL specification language. The mCRL2 language extends a timed basic process algebra with the possibility to define and use abstract data types. The mCRL2 data language features predefined and higher-order data types. The process algebraic part of mCRL2 allows a faithful translation of coloured Petri nets and component based systems: we have introduced multiactions and we have separated communication and parallelism.

Keywords. specification language, abstract data types, process algebra, operational semantics

1 Introduction

In a typical computerised system, a number of components are running simultaneously. 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 easy, 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 among 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 paper, we introduce the mCRL2 specification language [1]. With this language, users can specify the behaviour of a distributed system and analyse it using automated techniques. The language mCRL2 is the successor of μCRL [2, 3] and timed μCRL [4, 5] and is inspired by [6] and [7].

The μCRL language extends a basic process algebra – based on the Algebra of Communicating Processes (ACP) [8] – 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 language for the specification and analysis of real-life systems.

Dagstuhl Seminar Proceedings 06351
Methods for Modelling Software Systems (MMOSS)
http://drops.dagstuhl.de/opus/volltexte/2007/862

The μCRL language has clear and well-defined syntax and semantics. Over the years, various tools have been developed for μCRL [9, 10], all with a strong foundation in formal theories (see [3, 11] for an overview). 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 [12, 13, 14]). In nearly all cases the analysis revealed errors in the system being analysed.

Recently, as reported in [1], researchers at the Eindhoven University of Technology (TU/e) started the development of the mCRL2 language and toolset. Based on user experiences with μCRL, their focus is to develop a more user-friendly language and tool interface.

1.1 Improvements over the μCRL language

On the data side, the most substantial improvement to the language 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 shorter and easier to read and write.

In μCRL, even all basic data types such as the Booleans and the naturals needed to be defined explicitly. As a consequence, different users could give widely different specifications of those. For tools, properties about such data types turned out to be a hurdle that was hard to overcome. By having standard data types, dedicated programming techniques can be employed for proving or disproving such properties.

A more elaborate motivation for the chosen adaptations to the data language can be found in [1]. In the rest of this subsection we will discuss the changes to the process algebraic part of the language.

One can distinguish three main streams of process specification formalisms: assertional specification formalisms, Petri nets and process algebras. We would all benefit if these formalisms would be integrated. In the past, we did not find any fundamental difficulties in relating assertional methods and μCRL [15, 16]. However, with Petri nets [17] we ran into a problem. Consider the coloured Petri net in Fig. 1. There are two places P_1 and P_2 and a transition labelled with n^2 in the middle. The tokens in this net contain natural numbers and the transition squares the number in each token that it processes. The standard semantics of such a net is that *atomically* a token leaves P_1, has its value squared and is put into P_2.

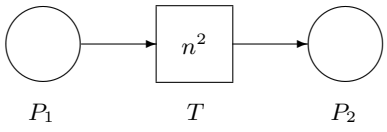


Fig. 1. A simple coloured Petri net

The natural structure-preserving translation of this Petri net into process algebra is in the form of the parallel composition $P_1 \parallel T \parallel P_2$. Using a standard synchronous communication mechanism as in ACP, a token can be read from P_1 into T, and in a subsequent step be forwarded from T to P_2. But now we have translated what was a single atomic step into two atomic steps.

There are two major drawbacks to such a translation: First, nice concepts from the world of Petri nets, such as state invariants, do not easily carry over, and second, doubling of states that results worsens the state space explosion problem that is the primary difficulty in analysing process behaviour.

To solve this problem, we have introduced so-called multiactions. In a multiaction zero or more actions occur simultaneously. As a consequence the transition in the above net can now be captured by a process that reads a token with a certain value n and delivers a token with the value n^2 in one multiaction. There is no straightforward and elegant way to achieve this in μCRL.

Allowing multiactions gives rise to a significant increase in the number of (multi)actions that can possibly happen. If we put n actions in parallel, the result contains $2^n - 1$ different multiactions. Often one only wants a small number of these multiactions to occur, which one cannot establish as easily with just the standard *blocking* (or *encapsulation*) operator. For this reason we have introduced a special *restriction* operator, which specifies which multiactions are actually allowed.

Furthermore, the way communication among parallel components is dealt with has been changed with respect to μCRL. In μCRL there is a *global* communication function, which means that there is only one such function specified and the (communication) behaviour of all processes depends solely on this function. This causes μCRL to be non-compositional. Therefore, mCRL2 uses a *local* communication mechanism, which allows one to specify the communication precisely where it is relevant and thus separates communication of distinct parts of a system.

For example, in the system of Fig. 2 the components A and B occur twice, but are connected in a different way. In the component on the left, actions a_1 of A and b_2 of B communicate, while in the component on the right a_1 communicates with b_3. Note that the action b_2 is also available on the right, where it must not communicate with a_1. This is not a problem with local communication as we can define the communication per subcomponent. With global communication, however, one needs to rename certain actions in one component to avoid conflicts with some unrelated component.

Also, we have chosen to separate the concepts of communication and parallelism, which gives more freedom in modelling (e.g. multiway communication only needs to be defined once instead of partially at every parallel operator). Consider, for example, a system with several components that need to synchronise. In mCRL2 this can be done by putting these components in parallel. The communication operator is used to express communication of precisely those actions. Without the separation of the parallel composition and communication, one must pair components into a new one, synchronising only the components

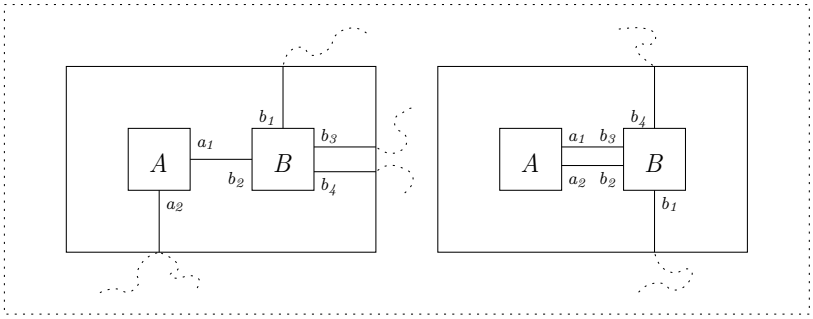


Fig. 2. Component-based system

of such a pair, and repeat this with the resulting components until only one is left.

1.2 Tool support

In general, the following steps are involved in the analysis of a system with mCRL2:

- A specification of the system's behaviour is written in the mCRL2 language.
- This specification is converted to a *Linear Process Specification* (LPS).[1] As we shall see, an LPS is an mCRL2 specification in a stricter format.
- The LPS can be modified/simplified using various manipulation tools and can be simulated using various simulation tools.
- A *Labelled Transition System* (LTS) or *state space* can be generated from the modified LPS. Subsequently, this LTS can be analysed for errors using model checking techniques.

For μCRL, these techniques have been defined, and implemented in the μCRL toolset. The μCRL toolset has been developed at and maintained by the Center for Mathematics and Computer Science (CWI) in Amsterdam since 1995 [9, 10]. Currently, we are adapting those tools and techniques to the setting of mCRL2.[2]

With mCRL2 we also want to *assist* the user in performing an analysis. For this reason we are developing a graphical user interface that allows users to manage their analyses in an intuitive way.

In this article, we restrict our focus to the theoretical side of the tool support: we show that most mCRL2 specifications can be converted to an LPS.

[1] Previously, in the literature, linear process specifications have been called linear process equations and linear process operators.

[2] Up-to-date information about the language mCRL2 and its toolset can be found at http://www.mcrl2.org.

1.3 Related work

As said, a reason for introducing multiactions in mCRL2 is to allow for a more straightforward translation of Petri nets to mCRL2. The material presented in this paper is not the first attempt to capture the features of Petri nets in a process algebraic setting. Over the years many attempts have been made to link Petri nets and process algebra.

In [18] and [19] an ACP approach is taken to combine these formalisms. The main difference between these approaches and mCRL2 lies in the way communication works and in the additional restriction operator of mCRL2. Communication in [18, 19] is global.

Although in [18] there is a local communication operator $\rho_{\overline{\gamma}}$, it still depends on a global communication function $\overline{\gamma}$ that defines which actions in a multiaction communicate. This requires renaming in case one wishes to use multiple instances of processes in different contexts in the same system, even though there might be no name clash. Also, the complete communication behaviour has to be specified globally, breaking the compositionality of the language.

Interworkings [20] and LOTOS [21] employ local communication. However, it is strictly linked to the parallel operator(s). In mCRL2 there is a separation of the concepts of communication and parallelism. This is also the case is in [19], where a general renaming operator is used. This operator takes any function on multiactions, thus communication is considered to be just simple renaming. With data, however, renaming and communication are two different concepts as communication can only take place if data parameters are agreed upon. For renaming on multiactions, as in [19], data parameters may differ, but in some way actions can still be related. We consider this as undesired, which is why our renaming is defined on action names only and we have a special operator for communication.

Both [18] and [19] do not have an equivalent to the restriction operator, which specifies which multiactions (disregarding data) are allowed to occur. Depending on the situation, it can be much more useful to specify which multiactions are allowed instead of which actions are not allowed. In a context without data, in [18] and [19], this operator can also be modelled by combining renaming/communication and blocking as follows: The process that only allows the multiactions consisting of precisely one a and one b action in process p can be described as follows:

$$\rho_{\overline{\gamma}}(\partial_{\{a,b\}}(\rho_{\overline{\gamma}}(p)))$$

with $\overline{\gamma}$ the identity on actions and $\overline{\gamma}(a|b) = c|d$ and $\overline{\gamma}(c|d) = a|b$ and c and d fresh actions w.r.t. process expression p. Here $\partial_{\{a,b\}}$ denotes the blocking of actions a and b. In a setting with data this becomes impossible because the actions a and b can have different data parameters, unless one allows communication to be defined not only on actions but also on data (e.g. $\overline{\gamma}(a(4)|b(true)) = c(4.5)$). Apart from data, also abstraction is not included in [18] and [19].

A more Petri net based approach is used in the Petri Box Calculus [22, 23] (also [24]). Here communication is done à la CCS [25], which means that every action has a counterpart and only these can communicate with each other. Such

a communication results in the *silent step* τ, interpreted as the empty multiset (as in mCRL2). This choice makes multiway communication more cumbersome. Because of this way of defining communication, one tends to model in such a way that the restriction operator is probably of less value which is perhaps why it is not included. Besides this they use a specific kind of syntax and semantics that preserves *history* (after executing an action the resulting process still contains this action, but only as being already executed).

1.4 Structure of the document

The document is structured as follows. In Sect. 2 we define the syntax of the mCRL2 process language together with a set of axioms. After that, we define the operational semantics of the process language in Sect. 3. We show that the axioms are sound and complete with respect to the semantics. In Sect. 4 we discuss the data language that is used by the process language. In Sect. 5 we show that most mCRL2 specifications can be represented by a linear process specification. Finally, we discuss future work in the Conclusions.

2 The mCRL2 process language

2.1 Actions

The most basic notion in the mCRL2 process language is an *action*. 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. 4.

In general, we write $a, b, \ldots$ to denote action names and $\boldsymbol{d}, \boldsymbol{e}, \ldots$ to denote vectors of data parameters. In the notation $a(\boldsymbol{d})$, we assume that the vector of data parameters $\boldsymbol{d}$ 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.

2.2 Multiactions

Multiactions represent a collection of actions that are assumed to occur at the same time (i.e., truly in parallel). Multiactions are constructed according to the following BNF:

$$\alpha ::= \tau \mid a(\boldsymbol{d}) \mid \alpha \sqcup \beta \ ,$$

where a denotes an action name and $\boldsymbol{d}$ a vector of data parameters. The constructor τ represents the multiaction containing no actions, the constructor $a(\boldsymbol{d})$ represents a multiaction that contains only (one occurrence of) the action $a(\boldsymbol{d})$, and the constructor $\alpha \sqcup \beta$ represents a multiaction containing the actions from both the multiactions α and β.

Using the actions declared previously the following are considered multiactions: τ, $\mathit{error} \sqcup \mathit{error} \sqcup \mathit{send}(\mathit{true})$, $\mathit{send}(\mathit{true}) \sqcup \mathit{receive}(\mathit{false}, 6)$ and $\tau \sqcup \mathit{error}$. We often write $\alpha, \beta, \ldots$ for multiactions.

On multiactions we define a notion of equality by means of the axioms from Table 1. We also define operators $\setminus$ and $\sqsubseteq$ on multiactions for later use. Here $\equiv$ denotes syntactic equality on action names and $\neq$ denotes provable inequality on data (vectors).

MA1	$\alpha \sqcup \beta = \beta \sqcup \alpha$
MA2	$(\alpha \sqcup \beta) \sqcup \gamma = \alpha \sqcup (\beta \sqcup \gamma)$
MA3	$\alpha \sqcup \tau = \alpha$
MD1	$\tau \setminus \alpha = \tau$
MD2	$\alpha \setminus \tau = \alpha$
MD3	$\alpha \setminus (\beta \sqcup \gamma) = (\alpha \setminus \beta) \setminus \gamma$
MD4	$(a(\boldsymbol{d}) \sqcup \alpha) \setminus a(\boldsymbol{d}) = \alpha$
MD5	$(a(\boldsymbol{d}) \sqcup \alpha) \setminus b(\boldsymbol{e}) = a(\boldsymbol{d}) \sqcup (\alpha \setminus b(\boldsymbol{e}))$ if $a \not\equiv b$ or $\boldsymbol{d} \neq \boldsymbol{e}$
MS1	$\tau \sqsubseteq \alpha = \mathit{true}$
MS2	$a(\boldsymbol{d}) \sqsubseteq \tau = \mathit{false}$
MS3	$a(\boldsymbol{d}) \sqcup \alpha \sqsubseteq a(\boldsymbol{d}) \sqcup \beta = \alpha \sqsubseteq \beta$
MS4	$a(\boldsymbol{d}) \sqcup \alpha \sqsubseteq b(\boldsymbol{e}) \sqcup \beta = a(\boldsymbol{d}) \sqcup (\alpha \setminus b(\boldsymbol{e})) \sqsubseteq \beta$ if $a \not\equiv b$ or $\boldsymbol{d} \neq \boldsymbol{e}$
MAN1	$\underline{\tau} = \tau$
MAN2	$\underline{a(\boldsymbol{d})} = a$
MAN3	$\underline{\alpha \sqcup \beta} = \underline{\alpha} \sqcup \underline{\beta}$

Table 1. Axioms for multiactions

In this paper, we assume that we have a similar set of operators and axioms for *multisets of action names*. In Table 1, we have also defined an operator $\underline{\alpha}$ that associates with a multiaction α the multiset of action names that are obtained by omitting all data parameters that occur in α.

2.3 Basic operators

Process expressions, denoted by $p, q, \ldots$, 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.
- *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 $P(\boldsymbol{d})$, $Q(\boldsymbol{d})$, etc. are used to refer to processes declared by *process definitions* of the form $P(\boldsymbol{x}{:}\boldsymbol{D}) = p$. This process definition declares that the behaviour of the process reference $P(\boldsymbol{d})$ is given by $p[\boldsymbol{d}/\boldsymbol{x}]$, i.e., p in which all free occurrences of variables $\boldsymbol{x}$ are replaced by $\boldsymbol{d}$.
- *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. The corresponding behaviour is a non-deterministic choice among the processes $p[d/x]$ for all elements $d \in D$. For $D = \{d_0, d_1, \ldots, d_n, \ldots\}$ this can be expressed as $p[d_0/x] + p[d_1/x] + \cdots + p[d_n/x] + \cdots$.
- *At operator*, written $p \vartriangleleft t$, where t is a data expression of sort $\mathbb{R}^{\geq 0}$ (non-negative real numbers). The expression $p \vartriangleleft t$ indicates that the first multiaction of p happens at time t.
- *Initialisation* operator, written $t \gg p$, where t is a data expression of sort $\mathbb{R}^{\geq 0}$. The initialisation operator is an auxilary operator, i.e., it is hardly ever used in modelling a system. The expression $t \gg p$ restricts the behaviour of p to the part that starts after time t.

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: $\vartriangleleft$, $\cdot$, $\gg$, $\rightarrow$, $\sum$, $+$. Furthermore, $\cdot$ and $+$ are associative (made formal in Table 3). 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 $\mathit{coin}, \mathit{break}, \mathit{coffee}$;
proc $\mathit{Wait} = \mathit{coin} \cdot \mathit{Serve}$;
$\mathit{Serve} = \mathit{break} \cdot \delta + \mathit{coffee} \cdot \mathit{Wait}$;

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:

$$\begin{aligned}\textbf{proc}\ P(c:\mathbb{B}, n:\mathbb{N}) = {} & error \cdot P(c,n)\\ & + send(c) \cdot P(\neg c, n+1)\\ & + receive(c,n) \cdot P(false, max(n-1,0));\end{aligned}$$

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 in 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.

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

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

$$\begin{aligned}&\textbf{act}\quad tick;\\ &\textbf{proc}\ C(t:\mathbb{R}^{\geq 0}) = tick \triangleleft t \cdot C(t+1);\\ &\textbf{init}\quad C(0);\end{aligned}$$

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

As a different example, we show a model of a *drifting* clock (taken from [26]). This is a clock that is accurate within a bounded interval $[1-\mathfrak{d}, 1+\mathfrak{d}]$, where $\mathfrak{d} < 1$.

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

2.4 Axioms for equality of processes

A structured operational semantics in the form of strong bisimulation equivalence classes of labelled transition systems (the labels are multiactions) can be associated with each process expression by means of deduction rules in the style of Plotkin [27, 28], just as has been done for μCRL [3] and timed μCRL [5]. We give such a semantics in Sect. 3, but first we use *axioms* to express the properties of the operators.

$$\frac{}{p = p} \qquad \frac{p = q}{q = p} \qquad \frac{p = q \quad q = r}{p = r}$$

$$\frac{p_1 = q_1 \quad \cdots \quad p_n = q_n}{f(p_1, \ldots, p_n) = f(q_1, \ldots, q_n)}$$

$$\frac{p = q \ \in \ Ax}{p = q} \qquad \frac{P(\boldsymbol{x} : \boldsymbol{D}) = p \ \in \ PD}{P(\boldsymbol{x}) = p} \qquad \frac{p = q}{p[d/x] = q[d/x]}$$

$$\frac{d = e}{p[d/x] = p[e/x]}$$

Table 2. Derivation rules for mCRL2 processes

The derivation rules for mCRL2 processes from Table 2 are used to prove equalities between processes. Here, and in all other tables in this paper, p, q, and r are arbitrary process expressions, f is an operator from the signature of mCRL2, Ax refers to the collection of axioms of mCRL2 as presented in this paper, PD refers to the collection of user-defined process definitions, x is a data variable, d and e are data expressions, and $p[d/x]$ denotes the result of substituting d for x in p in a *capture-avoiding* way. Substitution should be capture-avoiding since $\sum_{x:D}$ is a binder. Data variables may occur in process and data expressions. Note that we do not have process variables. The last rule in the table links provable equality on process expressions with provable equality on data expressions.

The axioms for the operators introduced so far are listed in tables 3 and 4. Here, α and β are multiactions, and t and u are data expressions of sort $\mathbb{R}^{\geq 0}$. The function fv gives the data variables that occur freely in a process expression.

With the axioms we can prove, for instance, $a + (\delta + a)$ equal to a using axioms A2, A6 and A3 as follows:

$$a + (\delta + a) \overset{\text{A2}}{=} (a + \delta) + a \overset{\text{A6}}{=} a + a \overset{\text{A3}}{=} a.$$

A1	$p + q = q + p$	
A2	$p + (q + r) = (p + q) + r$	
A3	$p + p = p$	
A4	$(p + q) \cdot r = p \cdot r + q \cdot r$	
A5	$(p \cdot q) \cdot r = p \cdot (q \cdot r)$	
A6	$\alpha + \delta = \alpha$	
A7	$\delta \cdot p = \delta$	
C1	$\mathit{true} \to p \diamond q = p$	
C2	$\mathit{false} \to p \diamond q = q$	
SUM1	$\sum_{x:D} p = p$	if $x \notin \mathit{fv}(p)$
SUM2	$\sum_{x:D} p = \sum_{y:D} p[y/x]$	if $y \notin \mathit{fv}(p)$
SUM3	$\sum_{x:D} p = \sum_{x:D} p + p$	
SUM4	$\sum_{x:D}(p + q) = \sum_{x:D} p + \sum_{x:D} q$	
SUM5	$(\sum_{x:D} p) \cdot q = \sum_{x:D} p \cdot q$	if $x \notin \mathit{fv}(q)$

Table 3. Axioms for the basic operators

T1	$p \lhd 0 = \delta \lhd 0$	
T2	$c \to p = c \to p \diamond \delta \lhd 0$	
T3	$p = \sum_{x:\mathbb{R}_{\geq 0}} p \lhd x$	if $x \notin \mathit{fv}(p)$
T4	$p \lhd t \cdot q = p \lhd t \cdot (t \gg q)$	
TA1	$\alpha \lhd t \lhd u = (t \approx u) \to \alpha \lhd t \diamond \delta \lhd \min(t, u)$	
TA2	$\delta \lhd t \lhd u = \delta \lhd \min(t, u)$	
TA3	$(p + q) \lhd t = p \lhd t + q \lhd t$	
TA4	$(p \cdot q) \lhd t = p \lhd t \cdot q$	
TA5	$(\sum_{x:D} p) \lhd t = \sum_{x:D} p \lhd t$	if $x \notin \mathit{fv}(t)$
TI1	$t \gg \alpha \lhd u = t < u \to \alpha \lhd u \diamond \delta \lhd t$	
TI2	$t \gg \delta \lhd u = \delta \lhd \max(t, u)$	
TI3	$t \gg (p + q) = t \gg p + t \gg q$	
TI4	$t \gg (p \cdot q) = (t \gg p) \cdot q$	
TI5	$t \gg \sum_{x:D} p = \sum_{x:D} t \gg p$	if $x \notin \mathit{fv}(t)$

Table 4. Timed axioms for the basic operators

2.5 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.
- *Left merge* $p\lfloor\!\lfloor q$, which is an auxiliary operator to allow for the axiomatisation of the parallel composition. (It only allows p to execute a first action and thereafter combines the remainder of p with q as the parallel composition does.)
- *Before* operator $p \ll q$, which is an auxiliary operator for the axiomatisation of the left merge that describes the part of process p that starts before or at the time q gets definitely disabled.

The corresponding axioms are given in Table 5. Observe that the axioms S1-S3, in combination with axiom SMA, are generalisations of the axioms MA1-MA3 to arbitrary process expressions. The synchronisation operator binds stronger than all other binary operators and parallel composition and left merge bind stronger than the sum operator but weaker than the conditional operator: $|$, $\leftthreetimes$, $\cdot$, $\{\gg, \ll\}$, $\rightarrow$, $\{\parallel, \lfloor\!\lfloor\}$, $\sum$, $+$. One might expect that the synchronisation operator binds equally strong as parallel composition and left merge, but due to the strong relationship between the synchronisation operator and multiset union, as expressed by axiom SMA, we prefer to let it bind strongest. This way one can always write the synchronisation operator instead of multiset union. In the rest of this paper, we always use the synchronisation operator.

2.6 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 blocked, e.g., $\nabla_{\{b|c\}}(a(0) + b(true, 5)|c) = b(true, 5)|c$.
  The axioms are given in Table 6. In this table, we use $\underline{\alpha}$ to denote the multiset of action names that is obtained from multiaction α by omitting the data parameters of all actions in α (see Table 1). We define $ma \underline{\in} V$ as follows:

  $$ma \underline{\in} V \text{ iff } ma = mb \text{ for some } mb \in V$$

  Here, $=$ denotes equality on multisets of action names. It has axioms similar to M1-M3. It is used to abstract from the order of the elements in the multiset

SMA	$\alpha \mid \beta = \alpha \sqcup \beta$
M	$p \parallel q = p \between q + q \between y + p \mid q$
LM1	$\alpha \between p = (\alpha \ll p) \cdot p$
LM2	$\delta \between p = \delta \ll p$
LM3	$\alpha \cdot p \between q = (\alpha \ll q) \cdot (p \parallel q)$
LM4	$(p + q) \between r = p \between r + q \between r$
LM5	$(\sum_{x:D} p) \between q = \sum_{x:D} p \between q$ if $x \notin fv(q)$
LM6	$p \lhd t \between q = (p \between q) \lhd t$
S1	$p \mid q = q \mid p$
S2	$(p \mid q) \mid r = p \mid (q \mid r)$
S3	$p \mid \tau = p$
S4	$\alpha \mid \delta = \delta$
S5	$(\alpha \cdot p) \mid \beta = \alpha \mid \beta \cdot p$
S6	$(\alpha \cdot p) \mid (\beta \cdot q) = \alpha \mid \beta \cdot (p \parallel q)$
S7	$(p + q) \mid r = p \mid r + q \mid r$
S8	$(\sum_{x:D} p) \mid q = \sum_{x:D} p \mid q$ if $x \notin fv(q)$
S9	$p \lhd t \mid q = (p \mid q) \lhd t$
TB1	$p \ll \alpha = p$
TB2	$p \ll \delta = p$
TB3	$p \ll q \lhd t = \sum_{u:\mathbb{R}^{\geq 0}} u \leq t \rightarrow p \lhd u \ll q$
TB4	$p \ll (q + r) = p \ll q + p \ll r$
TB5	$p \ll q \cdot r = p \ll q$
TB6	$p \ll \sum_{x:D} q = \sum_{x:D} p \ll q$ if $x \notin fv(p)$

Table 5. Axioms for the parallel composition operators

V1	$\nabla_V(\alpha) = \alpha$	if $\underline{\alpha} \subseteq V \cup \{\tau\}$	V4 $\nabla_V(p + q) = \nabla_V(p) + \nabla_V(q)$
V2	$\nabla_V(\alpha) = \delta$	if $\underline{\alpha} \not\subseteq V \cup \{\tau\}$	V5 $\nabla_V(p \cdot q) = \nabla_V(p) \cdot \nabla_V(q)$
V3	$\nabla_V(\delta) = \delta$		V6 $\nabla_V(\sum_{x:D} p) = \sum_{x:D} \nabla_V(p)$
			V7 $\nabla_V(p \lhd t) = \nabla_V(p) \lhd t$

Table 6. Axioms for the restriction operator

of action names and from redundant empty multisets. Note that the empty multiaction τ is not allowed as an element of the set V, but is always allowed to occur (see axioms V1 and V2).

– *Blocking operator* $\partial_B(p)$ (also known as *encapsulation*), where B is a set of action names 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., $\partial_{\{b\}}(a(0) + b(true, 5)|c) = a(0)$. For the blocking operator we need to detect whether or not action names in a multiaction occur in the set of action names B. The axioms are given in Table 7.

B1 $\partial_B(\tau) = \tau$		B5 $\partial_B(\delta) = \delta$
B2 $\partial_B(a(\boldsymbol{d})) = a(\boldsymbol{d})$	if $a \notin B$	B6 $\partial_B(p + q) = \partial_B(p) + \partial_B(q)$
B3 $\partial_B(a(\boldsymbol{d})) = \delta$	if $a \in B$	B7 $\partial_B(p \cdot q) = \partial_B(p) \cdot \partial_B(q)$
B4 $\partial_B(\alpha\|\beta) = \partial_B(\alpha)\|\partial_B(\beta)$		B8 $\partial_B(\sum_{x:D} p) = \sum_{x:D} \partial_B(p)$
		B9 $\partial_B(p \lhd t) = \partial_B(p) \lhd t$

Table 7. Axioms for the blocking operator

– *Renaming operator* $\rho_R(p)$, where R is a set of renamings of the form $a \to b$, meaning that every occurrence of action name a in p is replaced by action name b. Renaming $\rho_R(p)$ also disregards the data parameters, but when a renaming is applied the data parameters are retained, e.g., $\rho_{\{a \to b\}}(a(0) + a) = b(0) + b$. Note that every action name may only occur once as a left-hand side of a $\to$ in R. The axioms are given in Table 8.

R1 $\rho_R(\tau) = \tau$	
R2 $\rho_R(a(\boldsymbol{d})) = b(\boldsymbol{d})$	if $a \to b \in R$ for some b
R3 $\rho_R(a(\boldsymbol{d})) = a(\boldsymbol{d})$	if $a \to b \notin R$ for all b
R4 $\rho_R(\alpha\|\beta) = \rho_R(\alpha)\|\rho_R(\beta)$	
R5 $\rho_R(\delta) = \delta$	
R6 $\rho_R(p + q) = \rho_R(p) + \rho_R(q)$	
R7 $\rho_R(p \cdot q) = \rho_R(p) \cdot \rho_R(q)$	
R8 $\rho_R(\sum_{dx:D} p) = \sum_{x:D} \rho_R(p)$	
R9 $\rho_R(p \lhd t) = \rho_R(p) \lhd t$	

Table 8. Axioms for the renaming operator

– *Communication operator* $\Gamma_C(p)$, where C is a set of allowed communications of the form $a_0 \mid \cdots \mid a_n \to c$, with $n \geq 1$ and a_i and c action names. For each

communication $a_0|\cdots|a_n \to c$, multiactions containing $a_0(\boldsymbol{d})|\cdots|a_n(\boldsymbol{d})$ (for some $\boldsymbol{d}$) in p are replaced by $c(\boldsymbol{d})$. Note that the data parameters are retained in action c. For example $\Gamma_{\{a|b\to c\}}(a(0)|b(0)) = c(0)$, but also $\Gamma_{\{a|b\to c\}}(a(0)|b(1)) = a(0)|b(1)$. Furthermore, $\Gamma_{\{a|b\to c\}}(a(1)|a(0)|b(1)) = a(0)|c(1)$. The axioms are given in Table 9.

C1 $\Gamma_C(\alpha) = \gamma_C(\alpha)$	C4 $\Gamma_C(p \cdot q) = \Gamma_C(p) \cdot \Gamma_C(q)$
C2 $\Gamma_C(\delta) = \delta$	C5 $\Gamma_C(\sum_{x:D} p) = \sum_{x:D} \Gamma_C(p)$
C3 $\Gamma_C(p + q) = \Gamma_C(p) + \Gamma_C(q)$	C6 $\Gamma_C(p \lhd t) = \Gamma_C(p) \lhd t$

Table 9. Axioms for the communication operator

The function $\gamma_C(\alpha)$ applies the communications described by C to a multiaction α. It replaces every occurrence of a left-hand side of a communication it can find in α with the appropriate result. More precisely:

$$
\begin{array}{ll}
\gamma_\emptyset(\alpha) & = \alpha \\
\gamma_{C_1 \cup C_2}(\alpha) & = \gamma_{C_1}(\gamma_{C_2}(\alpha)) \\
\gamma_{\{a_0 \mid \cdots \mid a_n \to b\}}(\alpha) & = b(\boldsymbol{d}) \mid \gamma_{\{a_0 \mid \cdots \mid a_n \to b\}}(\alpha \setminus (a_1(\boldsymbol{d})|\cdots|a_n(\boldsymbol{d}))) \\
 & \quad \text{if } a_1(\boldsymbol{d})|\cdots|a_n(\boldsymbol{d}) \sqsubseteq \alpha \text{ for some } \boldsymbol{d} \\
\gamma_{\{a_0 \mid \cdots \mid a_n \to b\}}(\alpha) & = \alpha \qquad \text{otherwise}
\end{array}
$$

For example, $\gamma_{\{a|b\to c\}}(a|a|b|c) = a|c|c$ and $\gamma_{\{a|a\to a, b|c|d\to e\}}(a|b|a|d|c|a) = a|a|e$. The left-hand sides of the communications in C should be disjoint (e.g. $C = \{a|b \to c, a|d \to e\}$ is not allowed) to satisfy the desired property that $\gamma_{C_1}(\gamma_{C_2}(\alpha)) = \gamma_{C_2}(\gamma_{C_1}(\alpha))$ which guarantees that γ_C does not have multiple solutions.

2.7 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 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. The axioms are listed in Table 10.

H1 $\tau_I(\tau) = \tau$		H5 $\tau_I(\delta) = \delta$
H2 $\tau_I(a(\boldsymbol{d})) = \tau$	if $a \in I$	H6 $\tau_I(p + q) = \tau_I(p) + \tau_I(q)$
H3 $\tau_I(a(\boldsymbol{d})) = a(\boldsymbol{d})$	if $a \notin I$	H7 $\tau_I(p \cdot q) = \tau_I(p) \cdot \tau_I(q)$
H4 $\tau_I(\alpha \mid \beta) = \tau_I(\alpha) \mid \tau_I(\beta)$		H8 $\tau_I(\sum_{x:D} p) = \sum_{x:D} \tau_I(p)$
		H9 $\tau_I(p \lhd t) = \tau_I(p) \lhd t$

Table 10. Axioms for the abstraction operator

Although, at this stage, we do not include axioms that explicitly express that τ is an internal step and cannot be observed in certain cases (see [29, 30]), we note that the multiactions themselves already exhibit part of such abstraction properties. In specific, this is expressed in axiom S3 (and axiom MA3), which is the result from the fact that multiactions are actually bags and τ is the empty multiaction.

3 Operational semantics of the mCRL2 process language

3.1 Relating process expressions with the semantics

To define the semantics of the mCRL2 process language we interpret the process expressions defined in Sect. 2 to processes without data variables. This is done in a similar fashion as in [31].

As the process language depends on data, the semantics is parameterised by a data algebra $\mathcal{A}$ and a valuation η that maps data variables to data values. Furthermore, there is a parameter PD, the set of process definitions corresponding to the **proc** section of an mCRL2 specification.

We need some notation. We write $D_{\mathcal{A}}$ for the interpretation of sort D in data algebra $\mathcal{A}$. We also write $\eta[v/x]$ to denote a valuation that maps x to v and all other variables y to $\eta(y)$. Finally, we write $[\![d]\!]_\eta$ for the interpretation of data expression d.

The process operators in the semantics are very similar to those in the syntax. We will therefore also use the same notation. The difference is that the semantics does not have operators for alternative composition and conditional, and the summation operator on data types is replaced by a summation operator $\sum$ on sets of processes. This summation operator behaves as the alternative composition of all processes in the set.

The correspondence between the syntax and semantics is defined with the interpretation $[\![\_]\!]_\eta$ as follows. Note that we only provide the definition for the interesting cases; it is straightforward for the other operators.

$$\begin{array}{ll}
[\![\tau]\!]_\eta & = \tau \\
[\![a(\boldsymbol{d})|\alpha]\!]_\eta & = a([\![\boldsymbol{d}]\!]_{\boldsymbol{\eta}}) \sqcup [\![\alpha]\!]_\eta \\
[\![p+q]\!]_\eta & = \sum\{[\![p]\!]_\eta, [\![q]\!]_\eta\} \\
[\![b \to p]\!]_\eta & = \begin{cases} [\![p]\!]_\eta & \text{if } [\![b]\!]_\eta = [\![true]\!]_\eta \\ \delta@[\![0]\!]_\eta & \text{if } [\![b]\!]_\eta \neq [\![true]\!]_\eta \end{cases} \\
[\![b \to p \diamond q]\!]_\eta & = \begin{cases} [\![p]\!]_\eta & \text{if } [\![b]\!]_\eta = [\![true]\!]_\eta \\ [\![q]\!]_\eta & \text{if } [\![b]\!]_\eta \neq [\![true]\!]_\eta \end{cases} \\
[\![\sum_{x:D} p]\!]_\eta & = \sum\{[\![p]\!]_{\eta[v/x]} \mid v \in D_{\mathcal{A}}\}
\end{array}$$

We write PD_η for the set of interpreted process definitions from PD. That is, if $P(\boldsymbol{x} : \boldsymbol{D}) = q$ is in PD, then PD_η contains $P(\boldsymbol{v}) = [\![q]\!]_{\eta[\boldsymbol{v}/\boldsymbol{x}]}$, for all $\boldsymbol{v} \in \boldsymbol{D_{\mathcal{A}}}$.

We write $\mathbb{P}$ for the set of processes and $\mathbb{A}$ for the set of multiactions. The semantics of the mCRL2 process language is expressed by the action termination predicate $\longrightarrow \checkmark \subseteq \mathbb{P}\times\mathbb{A}\times\mathbb{R}^{\geq 0}_{\mathcal{A}}$, the action transition relation $\longrightarrow \subseteq \mathbb{P}\times\mathbb{A}\times\mathbb{R}^{\geq 0}_{\mathcal{A}}\times\mathbb{P}$ and the delay predicate $\rightsquigarrow \subseteq \mathbb{P}\times\mathbb{R}^{\geq 0}_{\mathcal{A}}$. The delay predicate indicates that a process can wait until at least the indicated time before not being able to let time pass without actually doing anything.

3.2 Auxiliary functions

We first introduce some auxiliary functions needed to define the semantics. Note that these functions work on interpreted multiactions or multiset of action names. For the blocking operator ∂_B we need to check whether or not a (interpreted) multiaction α contains an action name that is in B. We do this by converting α into a set such that we can take the intersection with B. This is done by first using $\_$ to remove the data from α and then applying $_{\{\}}$, which is defined as follows.

$$\begin{array}{ll}
\tau_{\{\}} & = \emptyset \\
a_{\{\}} & = \{a\} \\
(\alpha \sqcup \beta)_{\{\}} & = \alpha_{\{\}} \cup \beta_{\{\}}
\end{array}$$

With $\bullet$ we apply renaming R of the renaming operator ρ_R to a multiaction as follows:

$$\begin{array}{lll}
R \bullet \tau & = \tau & \\
R \bullet a(\boldsymbol{v}) & = b(\boldsymbol{v}) & \text{if } a \to b \in R \text{ for some } b \\
R \bullet a(\boldsymbol{v}) & = a(\boldsymbol{v}) & \text{if } a \to b \notin R \text{ for all } b \\
R \bullet (\alpha \sqcup \beta) & = (R \bullet \alpha) \sqcup (R \bullet \beta) &
\end{array}$$

Finally, we need to hide actions with names from the set I of the hiding operator τ_I, which we do with the $\theta_I()$ functions.

$$\begin{array}{lll}
\theta_I(\tau) & = \tau & \\
\theta_I(a(\boldsymbol{v})) & = \tau & \text{if } a \in I \\
\theta_I(a(\boldsymbol{v})) & = a(\boldsymbol{v}) & \text{if } a \notin I \\
\theta_I(\alpha \sqcup \beta) & = \theta_I(\alpha) \sqcup \theta_I(\beta) &
\end{array}$$

Note that we also need definitions of $\_$, $\in$ and γ_C, but these are essentially the same as in Sect. 2.

3.3 Structured operational semantics

We give the operational semantics using Structured Operational Semantics (SOS) in the style of Plotkin [27, 28]. The deduction rules of the basic operators are given in Tables 11-17. In Tables 18-21 the operational semantics is given for the parallel operators. Finally, the semantics for the additional operators is given in Tables 22-26.

$$\frac{}{\delta \rightsquigarrow_t}$$

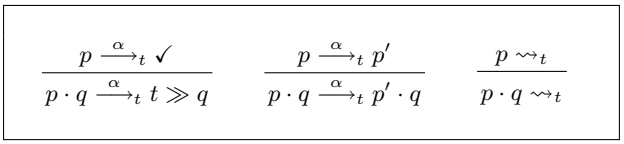
Table 11. Deduction rules for deadlock

$$\frac{}{\alpha \xrightarrow{\alpha}_t \surd} \quad t > 0 \qquad \frac{}{\alpha \rightsquigarrow_t}$$

Table 12. Deduction rules for multiactions

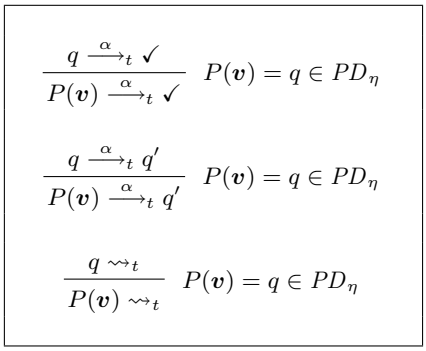
$$\frac{p \xrightarrow{\alpha}_t \surd}{\sum\{p\} \cup S \xrightarrow{\alpha}_t \surd} \qquad \frac{p \xrightarrow{\alpha}_t p'}{\sum\{p\} \cup S \xrightarrow{\alpha}_t p'} \qquad \frac{p \rightsquigarrow_t}{\sum\{p\} \cup S \rightsquigarrow_t}$$

Table 13. Deduction rules for summation operator



$$\frac{p \xrightarrow{\alpha}_t \checkmark}{p \cdot q \xrightarrow{\alpha}_t t \gg q} \qquad \frac{p \xrightarrow{\alpha}_t p'}{p \cdot q \xrightarrow{\alpha}_t p' \cdot q} \qquad \frac{p \rightsquigarrow_t}{p \cdot q \rightsquigarrow_t}$$

Table 14. Deduction rules for sequential composition

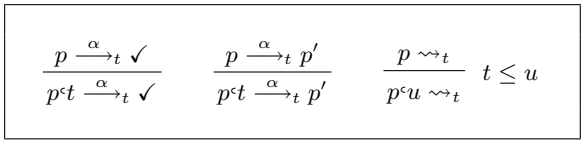


$$\frac{q \xrightarrow{\alpha}_t \checkmark}{P(\boldsymbol{v}) \xrightarrow{\alpha}_t \checkmark} \quad P(\boldsymbol{v}) = q \in PD_\eta$$

$$\frac{q \xrightarrow{\alpha}_t q'}{P(\boldsymbol{v}) \xrightarrow{\alpha}_t q'} \quad P(\boldsymbol{v}) = q \in PD_\eta$$

$$\frac{q \rightsquigarrow_t}{P(\boldsymbol{v}) \rightsquigarrow_t} \quad P(\boldsymbol{v}) = q \in PD_\eta$$

Table 15. Deduction rules for process references



$$\frac{p \xrightarrow{\alpha}_t \checkmark}{p \triangleleft t \xrightarrow{\alpha}_t \checkmark} \qquad \frac{p \xrightarrow{\alpha}_t p'}{p \triangleleft t \xrightarrow{\alpha}_t p'} \qquad \frac{p \rightsquigarrow_t}{p \triangleleft u \rightsquigarrow_t} \quad t \leq u$$

Table 16. Deduction rules for the at operator

$$\frac{p \xrightarrow{\alpha}_t \surd}{u \gg p \xrightarrow{\alpha}_t \surd}\ u < t \qquad \frac{p \xrightarrow{\alpha}_t p'}{u \gg p \xrightarrow{\alpha}_t p'}\ u < t$$

$$\frac{p \rightsquigarrow_t}{u \gg p \rightsquigarrow_t} \qquad \frac{}{u \gg p \rightsquigarrow_t}\ t \leq u$$

Table 17. Deduction rules for the initialisation operator

$$\frac{p \xrightarrow{\alpha}_t \surd, q \rightsquigarrow_t}{\begin{array}{l} p \parallel q \xrightarrow{\alpha}_t t \gg q \\ q \parallel p \xrightarrow{\alpha}_t t \gg q \end{array}} \qquad \frac{p \xrightarrow{\alpha}_t \surd, q \xrightarrow{\beta}_t \surd}{p \parallel q \xrightarrow{\alpha \sqcup \beta}_t \surd} \qquad \frac{p \rightsquigarrow_t, q \rightsquigarrow_t}{p \parallel q \rightsquigarrow_t}$$

$$\frac{p \xrightarrow{\alpha}_t p', q \rightsquigarrow_t}{\begin{array}{l} p \parallel q \xrightarrow{\alpha}_t p' \parallel t \gg q \\ q \parallel p \xrightarrow{\alpha}_t t \gg q \parallel p' \end{array}} \qquad \frac{p \xrightarrow{\alpha}_t p', q \xrightarrow{\beta}_t \surd}{\begin{array}{l} p \parallel q \xrightarrow{\alpha \sqcup \beta}_t p' \\ q \parallel p \xrightarrow{\alpha \sqcup \beta}_t p' \end{array}} \qquad \frac{p \xrightarrow{\alpha}_t p', q \xrightarrow{\beta}_t q'}{p \parallel q \xrightarrow{\alpha \sqcup \beta}_t p' \parallel q'}$$

Table 18. Deduction rules for parallel composition

$$\frac{p \xrightarrow{\alpha}_t \surd, q \xrightarrow{\beta}_t \surd}{p|q \xrightarrow{\alpha \sqcup \beta}_t \surd} \qquad \frac{p \rightsquigarrow_t, q \rightsquigarrow_t}{p|q \rightsquigarrow_t}$$

$$\frac{p \xrightarrow{\alpha}_t p', q \xrightarrow{\beta}_t \surd}{\begin{array}{l} p|q \xrightarrow{\alpha \sqcup \beta}_t p' \\ q|p \xrightarrow{\alpha \sqcup \beta}_t p' \end{array}} \qquad \frac{p \xrightarrow{\alpha}_t p', q \xrightarrow{\beta}_t q'}{p|q \xrightarrow{\alpha \sqcup \beta}_t p' \parallel q'}$$

Table 19. Deduction rules for the synchronisation operator

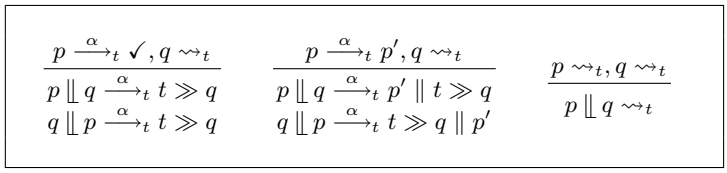


Table 20. Deduction rules for the left merge operator

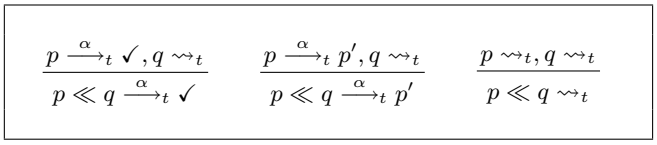


Table 21. Deduction rules for the before operator

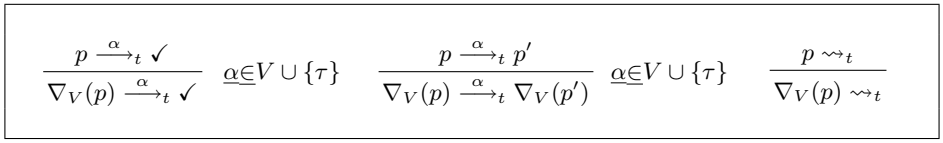


Table 22. Deduction rules for the restriction operator

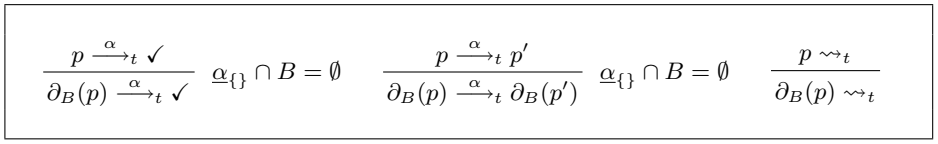


Table 23. Deduction rules for the blocking operator

$$\frac{p \xrightarrow{\alpha}_t \checkmark}{\rho_R(p) \xrightarrow{R\bullet\alpha}_t \checkmark} \qquad \frac{p \xrightarrow{\alpha}_t p'}{\rho_R(p) \xrightarrow{R\bullet\alpha}_t \rho_R(p')} \qquad \frac{p \rightsquigarrow_t}{\rho_R(p) \rightsquigarrow_t}$$

Table 24. Deduction rules for the renaming operator

$$\frac{p \xrightarrow{\alpha}_t \checkmark}{\Gamma_C(p) \xrightarrow{\gamma_C(\alpha)}_t \checkmark} \qquad \frac{p \xrightarrow{\alpha}_t p'}{\Gamma_C(p) \xrightarrow{\gamma_C(\alpha)}_t \Gamma_C(p')} \qquad \frac{p \rightsquigarrow_t}{\Gamma_C(p) \rightsquigarrow_t}$$

Table 25. Deduction rules for the communication operator

$$\frac{p \xrightarrow{\alpha}_t \checkmark}{\tau_I(p) \xrightarrow{\theta_I(\alpha)}_t \checkmark} \qquad \frac{p \xrightarrow{\alpha}_t p'}{\tau_I(p) \xrightarrow{\theta_I(\alpha)}_t \tau_I(p')} \qquad \frac{p \rightsquigarrow_t}{\tau_I(p) \rightsquigarrow_t}$$

Table 26. Deduction rules for the hiding operator

3.4 Bisimilarity

We define a notion of equivalence, called strong bisimilarity [32], on the processes in our semantics. This equivalence is a congruence for the operators in our language.

Definition 3.1 (Strongly bisimilar). *Let p and q be two process expressions, let PD be a set of process definitions and $\mathcal{A}$ a data algebra. We call p and q* (strongly) bisimilar*, notation $p \leftrightarrows q$, if there exists a symmetric relation R, called a* strong bisimulation relation*, relating $[\![p]\!]_\eta$ and $[\![q]\!]_\eta$ for all valuations η and such that for all r and s with $(r, s) \in R$, the following holds.*

- *If $r \xrightarrow{\alpha}_t r'$ for some multiaction α, time t and process r', then there exists a s' such that $s \xrightarrow{\alpha}_t s'$ and $(r', s') \in R$.*
- *If $r \xrightarrow{\alpha}_t \checkmark$ for some multiaction α and time t, then also $s \xrightarrow{\alpha}_t \checkmark$.*
- *If $r \rightsquigarrow_t$ for some time t, then also $s \rightsquigarrow_t$.*

Theorem 3.2 (Equivalence). *Strong bisimilarity is an equivalence.*

Proof. It is straightforward to check that strong bisimilarity is reflexive, symmetric and transitive.

Theorem 3.3 (Congruence). *Strong bisimilarity is a congruence for all operators of mCRL2.*

Proof. Because our semantics fits the *path* format [33], we know that strong bisimilarity (on the processes in the semantics) is a congruence for all operators in our semantics. Note that technically this result is meaningless for summation, since its argument is not a process but a set of processes. However, it is easy to show that strong bisimilarity is also a congruence for summation.

Now, using Definition 3.1, we can easily show that strong bisimilarity is also a congruence for all (syntactic) mCRL2 operators.

To express the relation between the semantics and the axiomatisation given in Sect. 2, we give the following two theorems that express that strong bisimulation and axiomatic derivability coincide.

Theorem 3.4 (Soundness). *The axiomatisation of mCRL2 is sound with respect to strong bisimilarity. That is, for all process expressions p and q we have that $p = q$ implies $p \leftrightarrows q$.*

Proof. Straightforward but laborious, and therefore not given here.

For completeness of our axiomatisation we only consider expressions without process references which we call *process-closed* expressions. Also, completeness of the process language depends on completeness of the data language. We there for call it relative completeness.

Theorem 3.5 (Completeness). *The axiomatisation of mCRL2 is relatively complete for process-closed expressions with respect to strong bisimilarity. That is, for all process-closed expressions p and q we have that $p \leftrightarrows q$ implies $p = q$.*

Proof. We only give a sketch of the proof. As our language is quite related to timed μCRL, we reuse the completeness proof of [5]. In this proof, the language pCRL$_t$ forms the basis. Therefore we translate our process expressions to pCRL$_t$ expressions. The main differences between our semantics and that of pCRL$_t$ are that we have multiactions and non-urgency. Multiactions α are translated to actions $ma(\overline{\alpha})$, where $\overline{\alpha}$ is a representation of α in the data language used in the pCRL$_t$ setting. To be able to translate non-urgency we add an additional, and eliminable, operator $\ggg$ that corresponds to the mCRL2 $\gg$ operator to get pCRL$_t^{\ggg}$.

The actual completeness proof uses the following properties:

1. Every process expression can be written to a basic form. This is quite standard and does not deviate much from similar lemmas in, for example, [5]. We call process expressions in basic form *basic expressions*.

2. The axioms of $\mathrm{pCRL}_t^{\ggg}$ are sound. This follows from the soundness of the axioms of pCRL_t and the soundness of the $\ggg$ axioms. The proof for the latter is similar to the proofs of $\gg$ in pCRL_t.
3. The axiomatisation of $\mathrm{pCRL}_t^{\ggg}$ is complete. This follows from the fact that $\ggg$ is eliminable and the completeness of pCRL_t.
4. The translation of basic expressions to $\mathrm{pCRL}_t^{\ggg}$ preserves bisimilarity. That is, if two bisimilar mCRL2 process expressions are translated, they are translated to bisimilar $\mathrm{pCRL}_t^{\ggg}$ expressions.
5. The translation of $\mathrm{pCRL}_t^{\ggg}$ expressions back to mCRL2 expressions preserves axiomatic derivability. That is, if we translate derivably equal $\mathrm{pCRL}_t^{\ggg}$ expressions back to mCRL2, they are translated to derivably equal mCRL2 expressions.
6. The translation of $\mathrm{pCRL}_t^{\ggg}$ to mCRL2 is the inverse of the translation of mCRL2 to $\mathrm{pCRL}_t^{\ggg}$ (modulo axiomatic derivability).

By property 1, completeness for all mCRL2 process expressions follows from completeness for all basic expressions. Thus we show that for all bisimilar basic expression we can derive that they are equal in the axiomatisation. To do so, we first translate them to $\mathrm{pCRL}_t^{\ggg}$. By property 4, we know that this gives us bisimilar $\mathrm{pCRL}_t^{\ggg}$ expressions. Then by properties 2 and 3 we can prove that these expressions are derivably equal in $\mathrm{pCRL}_t^{\ggg}$. Using the translation from $\mathrm{pCRL}_t^{\ggg}$ to mCRL2 we obtain mCRL2 expressions that are derivably equal by property 5. Finally, because the translation back is the inverse of the first translation (property 6), we have the same basic expressions of which we now know that they are derivably equal in mCRL2.

4 The mCRL2 data language

The mCRL2 data language is a functional language based on *higher-order abstract data types* [34, 35, 36]. As mentioned before, mCRL2 also has concrete data types: *standard data types* and sorts constructed from a number of *type constructors*.

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 are declared and (partially) defined:

sort A;
cons $c, d : A$;
map $f : A \times A \to A$;
$g : A \to A$;
var $x : A$;
eqn $f(c, x) = c$;
$f(d, x) = x$;
$g(c) = c$;

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.

Equations are used to derive equalities between data expressions. The derivation rules are the standard rules for equational logic (reflexivity, symmetry, transitivity, congruence and substitution), plus extensionality and induction on constructor sorts.

Standard data types. In mCRL2, a number of sorts and functions on those are predefined. The following standard data types are defined:

- 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 numbers ($\mathbb{N}^+$), natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$), non-negative real numbers ($\mathbb{R}^{\geq 0}$) 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 *A2B* are provided for all sorts $A, B \in \{ \mathbb{N}^+, \mathbb{N}, \mathbb{Z}, \mathbb{R}^{\geq 0}, \mathbb{R} \}$.

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, e.g. by adding the axiom $\mathit{true} = \mathit{false}$. In such a case, the data specification looses its meaning.

Type constructors. There are a number of type constructors, of which the first is a *structured type*. This is a compact way of defining a sort together with constructor, projection, and recogniser functions. 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 $\mathit{mode}(\mathit{running}(n)) = n$ and $\mathit{mode}(c)$ 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 constructor. The sort of functions from A to B is denoted $A \to B$. 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., $A \to B \to C$ means $A \to (B \to C)$.

We also have a *list* type constructor. The sort of (finite) lists containing elements of sort A is declared by $List(A)$ and has constructor functions $[\,] : List(A)$ and $\triangleright : A \times List(A) \to List(A)$. Other operators include $\triangleleft$, $+\!\!+$ (concatenation), . (element at), *head*, *tail*, *rhead* and *rtail* together with list enumeration $[\,e_0, \ldots, e_n\,]$. 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 $\{\,d_0, \ldots, d_n\,\}$, bag enumeration $\{\,d_0 : c_0, \ldots, d_n : c_n\,\}$ (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 $\overline{s}$ and infix operators $\subseteq$, $\subset$, $\cup$, $-$, $\cap$ with their usual meaning for sets and bags. Also conversion functions *Set2Bag* and *Bag2Set* are provided.

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** $leaf(\mathbb{N}) \mid node(T, T)$;

This declares sort T with constructor functions $leaf : \mathbb{N} \to T$ and $node : T \times T \to T$, and 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 constructors 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:

$$\begin{array}{ll}\textbf{var} & x, y : A;\\ \textbf{eqn} & x \approx y \;\rightarrow\; f(x,y) = x;\end{array}$$

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

$$\begin{array}{lll}\textbf{map} & h, h' : A \to A \to A; & \\ \textbf{var} & x, y : A; & \\ \textbf{eqn} & h(x) & = \lambda_{y':A}(\lambda_{z:A} f(z, g(z)))(g(f(x, y')));\\ & h'(x)(y) & = f(z, g(z)) \;\textbf{whr}\; z = g(f(x,y)) \;\textbf{end};\end{array}$$

Note that the functions h and h' are equivalent, i.e., one can derive that $h = h'$.

5 Linear process specifications

5.1 Definition and examples

The transition system corresponding to an mCRL2 specification often has a number of states that is exponential in the number of parallel processes. The fact that many process definitions 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.

On the other hand, general specifications are not well suited for manipulation – either by hand or using tools. Therefore it is useful to transform specifications to a basic form called *linear process specification* or LPS for short. An LPS can be seen as a (symbolic) representation of the transition system of a model. It uses the basic mCRL2 operators only, and in a very specific way.

Definition 5.1. *A* linear process specification *(LPS) is a process specification that contains a single process definition of the* linear form

$$\begin{array}{ll}\textbf{proc}\; P(x{:}D) = & \sum_{i\in I} \sum_{y_i:E_i} c_i(x,y_i) \to \alpha_i(x,y_i) \lhd t_i(x,y_i) \cdot P(g_i(x,y_i))\\ & + \sum_{j\in J} \sum_{y_j:E_j} c_j(x,y_j) \to \alpha_{\delta\, j}(x,y_j) \lhd t_j(x,y_j);\end{array}$$

where data expressions of the form $d(x_1, \ldots, x_n)$ contain at most free variables from $\{ x_1, \ldots, x_n \}$, I and J are disjoint and finite index sets, and for $i \in I$ and $j \in J$ the following are:

- *$c_i(x, y_i)$ and $c_j(x, y_j)$ are boolean expressions representing the conditions,*
- *$\alpha_i(x, y_i)$ is a multiaction $a_i^1(f_i^1(x, y_i))|\cdots|a_i^{n_i}(f_i^{n_i}(x, y_i))$, where $f_i^k(x, y_i)$ (for $1 \le k \le n_i$) are the parameters of action name a_i^k,*

- $\alpha_{\delta j}(x, y_j)$ *is either* δ *or a multiaction* $a_j^1(f_j^1(x, y_j))|\cdots|a_j^{n_j}(f_j^{n_j}(x, y_j))$*, where* $f_j^k(x, y_j)$ *(for* $1 \leq k \leq n_j$*) are the parameters of action name* a_j^k*, respectively,*
- $t_i(x, y_i)$ *and* $t_j(x, y_j)$ *are expressions of sort* $\mathbb{R}^{\geq 0}$ *representing the time stamps of multiactions* $\alpha_i(x, y_i)$ *and* $\alpha_{\delta j}(x, y_j)$*, respectively,*
- $g_i(x, y_i)$ *is an expression of sort* D *representing the next state of the process definition* P*;*

and contains an initialisation of the form

init $P(d_0)$;

where d_0 *is a closed data expression.*

Note that the summands $\sum_{i\in I}$ *and* $\sum_{j\in J}$ *are* meta-level *operations:* $\sum_{i\in I} p_i$ *is a shorthand for* $p_1 + \cdots + p_n$*, where* $I = \{1, \ldots, n\}$*.*

We call data parameter x the *state* parameter. In general we only strictly adhere to the form above with one state parameter x and one sum variable y_i per summand in theoretical considerations. In practice we can use any number or leave out all.

The form of the first 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 at time $t_i(d, e)$ 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 $g_i(x, y_i)$.

We illustrative the above with some examples. The process specification *Buffer* (also presented in Sect. 2)

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

is *not* linear because it has 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:

$$\begin{aligned}\textbf{proc}\ P(n{:}\mathbb{N}, b{:}\mathbb{B}) = &\textstyle\sum_{m:\mathbb{N}} b \to r(m){\cdot}P(m, \neg b)\\ &+ \neg b \to s(n){\cdot}P(n, \neg b);\end{aligned}$$

init $P(0, true)$;

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

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

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

The following LPS that has been derived from the process definition above behaves in the same way:

$$\begin{aligned}
\textbf{proc}\ P_{13}(n_1{:}\mathbb{N}, b_1{:}\mathbb{B}, n_2{:}\mathbb{N}, b_2{:}\mathbb{B}) &= \textstyle\sum_{m:\mathbb{N}} b_1 \to r_1(m){\cdot}P_{13}(m, \neg b_1, n_2, b_2)\\
&\quad + (\neg b_1 \wedge b_2) \to c_2(n_1){\cdot}P_{13}(n_1, \neg b_1, n_1, \neg b_2)\\
&\quad + \neg b_2 \to s_3(n_2){\cdot}P_{13}(n_1, b_1, n_2, \neg b_2);\\
\textbf{init}\ P_{13}(0, \mathit{true}, 0, \mathit{true});&
\end{aligned}$$

5.2 Linearisation of mCRL2 process specifications

In general the linearisability of mCRL2 process specifications can be stated in the following way. Let M be an mCRL2 process specification with a set of process definitions for process references $\{P_1, \ldots, P_m\}$. Let a new sort S contain (representations of) mCRL2 process expressions that are constructed with $\{P_1, \ldots, P_m\}$, actions and data definitions from M. It is easy to see that such a data type can be defined in the data language of mCRL2 (Sect. 4). So, for any process expression p constructed with the definitions from M, let $\mathrm{S}(p)$ be its representation in S.

For the sort S we define the following predicates in the way that they reflect derivability with the SOS rules from mCRL2 (Sect. 3):

$$\begin{array}{lll}
[\![c_a(\mathrm{S}(p), \mathrm{S}(q), \mathrm{S}(\alpha), t)]\!]_\eta = [\![\mathit{true}]\!]_\eta & \text{iff} & [\![p]\!]_\eta \xrightarrow{[\![\alpha]\!]_\eta}_{[\![t]\!]_\eta} [\![q]\!]_\eta\\
[\![c_t(\mathrm{S}(p), \mathrm{S}(\alpha), t)]\!]_\eta = [\![\mathit{true}]\!]_\eta & \text{iff} & [\![p]\!]_\eta \xrightarrow{[\![\alpha]\!]_\eta}_{[\![t]\!]_\eta} \checkmark\\
[\![c_d(\mathrm{S}(p), t)]\!]_\eta = [\![\mathit{true}]\!]_\eta & \text{iff} & [\![p]\!]_\eta \rightsquigarrow_{[\![t]\!]_\eta}
\end{array}$$

where p and q are mCRL2 process expressions, α is an arbitrary multiaction and $t{:}\mathbb{R}^{\geq 0}$. We again argue that these predicates can be defined in the syntax of mCRL2 data type definitions (Sect. 4).

Note that it is possible to define processes in mCRL2 that can perform infinitely many different multiactions. For example, process P, defined as $P = a + (P\,|\,a)$, can perform multiactions a, $a\,|\,a$, $a\,|\,a\,|\,a$ and so on. An LPS cannot mimic this behaviour because it can only perform a finite number of different multiactions as it has only a finite number of summands.

For a fixed $n \geq 0$ we define the following LPS:

$$\begin{aligned}\textbf{proc}\ P(s{:}S) = &\textstyle\sum_{\mu\in A^n}\sum_{y_\mu:E_\mu}\sum_{t:\mathbb{R}^{\geq 0}}\sum_{s':S} c_a(s,s',\mathrm{S}(\mu(y_\mu)),t) \to \mu(y_\mu)\triangleleft t\cdot P(s')\\ &+\textstyle\sum_{\mu\in A^n}\sum_{y_\mu:E'_\mu}\sum_{t:\mathbb{R}^{\geq 0}} c_t(s,\mathrm{S}(\mu(y_\mu)),t) \to \mu(y_\mu)\triangleleft t\\ &+\textstyle\sum_{t:\mathbb{R}^{\geq 0}} c_d(s,t) \to \delta\triangleleft t;\\ \textbf{init}\quad &P(\mathrm{S}(p));\end{aligned}$$

where

- p is the initial process expression of M,
- A^n is the set of bags of action names defined in M of size at most n.
- E_μ is a sort **struct** $e_\mu(\pi^1{:}E^1,\ldots,\pi^k{:}E^k)$ where $\mu = a^1|\cdots|a^k$, $0 \leq k \leq n$, and E^i (for $1 \leq i \leq k$) are the sorts of arguments of actions a^i.
- $\mu(y_\mu)$ is the multiaction $a^1(\pi^1(y_\mu))|\cdots|a^k(\pi^k(y_\mu))$, where $\mu = a^1|\cdots|a^k$, $0 \leq k \leq n$, and π^i (for $1 \leq i \leq k$) are the projection functions of E_μ.

This LPS defining process P is related to the process specification M in the following way. Suppose the process defined by M cannot perform multiactions of size greater than n. Then this process is bisimilar to the process defined by the LPS for P. The proof of this fact follows by considering the SOS rules applied to the process definition P.

This approach to the linearisation problem has two drawbacks. One is a theoretical one, because such a transformation is only valid in one model of mCRL2 axioms, namely the one defined in Sect. 3. The other one is of a practical nature; it turns out that in many practical cases the state space of P has far too many states.

The solution to both problems is a linearisation method that is based on the syntactic transformation of process specifications. This means using the axioms of mCRL2 and rules to transform process definitions. This method of linearisation is implemented in both the μCRL and mCRL2 toolsets. In [37] it is described how this can be done for a large class of process specifications in timed μCRL. 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. The techniques have been used in almost the same form in mCRL2 to obtain LPSs for mCRL2 processes.

6 Concluding remarks

In this paper, we have motivated and presented the process and data language of mCRL2, the successor of μCRL. Our main results are (relative) completeness of the process language and linearisability of most specifications. We conclude by mentioning some future work.

Formal semantics of the data language. Usually the operations of algebraic specifications are of first order: functions cannot occur as parameters nor as results of other functions. In higher-order specifications, this restriction is lifted: functions are treated as first-class citizens. Higher-order algebra [35, 36, 38] is

roughly ordinary universal algebra [39, 40] extended with function types. For the data language a formal semantics will be given in terms of higher-order algebra. The concrete data types should then be expressed in this basic formalism in an algebraic manner, i.e. by means of signatures and axioms.

Timed branching bisimulation equivalence. In this article we have presented axioms for a notion of equivalence called strong bisimilarity. This notion does not allow one to abstract from the presence of internal actions.

It would be desirable to consider a notion of equivalence that does abstract from internal actions without abstracting from the observable effect of their presence. In the setting of μCRL, the notion of (rooted) branching bisimulation equivalence [29, 30] is used for this purpose.

For timed process algebras, timed versions of branching bisimulation equivalence have been proposed in the literature (see for example [41]). To date, we are not yet confident that the right notion of timed branching bisimulation equivalence has already been proposed. Therefore, we aim to look further into this topic.

Proof techniques. The proof techniques that have been developed for the setting of μCRL, such as the linearisation algorithms for μCRL and timed μCRL [37], the cones and foci proof method for μCRL [42, 3] and timed μCRL [43], the confluence reduction techniques for μCRL [44], and the simple LPS transformations from the μCRL toolset [9, 10] have to be adapted to the setting of mCRL2. We expect that this only involves small changes to the proof techniques, their proofs of correctness, and their implementations in the mCRL2 toolset.

Relation with Petri nets. One of the motivations for changing the process language of mCRL2 was the desire to model Petri nets in a compositional way. We believe we have succeeded in this goal, but this claim needs further proof. Also, we still want to establish which of the techniques that are applied to Petri nets can be carried over to mCRL2 in a meaningful way, and the other way around, which of the techniques from the world of μCRL and mCRL2 are useable for the analysis of Petri nets.

References

[1] Groote, J.F., Mathijssen, A., van Weerdenburg, M., Usenko, Y.S.: From μCRL to mCRL2: motivation and outline. In: Proc. Workshop Essays on Algebraic Process Calculi (APC 25). Volume 162 of Electronic Notes in Theoretical Computer Science. (2006) 191–196

[2] Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. In Ponse, A., Verhoef, C., Vlijmen, S.F.M.v., eds.: Algebra of Communicating Processes, Utrecht 1994. Workshops in Computing, Springer-Verlag (1995) 26–62

[3] Groote, J.F., Reniers, M.A.: Algebraic process verification. In Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier Science Publishers B.V., Amsterdam (2001) 1151–1208
[4] Groote, J.F.: The syntax and semantics of timed μCRL. Technical Report SEN-R9709, CWI, Amsterdam (1997)
[5] Reniers, M.A., Groote, J.F., van der Zwaag, M.B., van Wamel, J.: Completeness of timed μCRL. Fundamenta Informaticae **50**(3-4) (2002) 361–402
[6] van Weerdenburg, M.: GenSpect process algebra. Master's thesis, Technische Universiteit Eindhoven (TU/e) (2004)
[7] van Weerdenburg, M.: Process algebra with local communication. Technical Report 05/05, Technische Universiteit Eindhoven (TU/e) (2005)
[8] Baeten, J.C.M., Weijland, W.P.: Process Algebra. Volume 18 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (1990)
[9] Blom, S., Fokkink, W.J., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.C.: CRL: A toolset for analysing algebraic specifications. In Berry, G., Comon, H., Finkel, A., eds.: Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings. Volume 2102 of Lecture Notes in Computer Science. (2001) 250–254
[10] Blom, S., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.C.: New developments around the μCRL tool set. In: Proc. 8th Int'l Workshop on Formal Methods for Industrial Critical Systems (FMICS'03). Volume 80 of Electronic Notes in Theoretical Computer Science. (2003) 1–5
[11] Fokkink, W.J., Groote, J.F., Reniers, M.A.: Process algebra needs proof methodology (columns: Concurrency). Bulletin of the EATCS **82** (2004) 109–125
[12] Fokkink, W.J., Groote, J.F., Pang, J., Badban, B., van de Pol, J.C.: Verifying a sliding window protocol in μCRL. In Rattray, C., Maharaj, S., Shankland, C., eds.: Proc. 10th International Conferance on Algebraic Methodology and Software Technology, AMAST'04, Stirling. Volume 3116 of Lecture Notes in Computer Science., Springer-Verlag (2004) 148–163
[13] Groote, J.F., Pang, J., Wouters, A.G.: Analysis of a distributed system for lifting trucks. Journal of Logic and Algebraic Programming **55**(1-2) (2003) 21–56
[14] Pang, J., Fokkink, W.J., Hofman, R., Veldema, R.: Model checking a cache coherence protocol for a java DSM implementation. In: Proc. 2003 International Parallel and Distributed Processing Symposium (IPDPS'03), Nice, IEEE Computer Society Press (2003)
[15] Groote, J.F., Ponse, A.: Process algebra with guards. Combining Hoare logic and process algebra. Formal Aspects of Computing **6**(2) (1994) 115–164
[16] Bezem, M.A., Groote, J.F.: Invariants in process algebra with data. In Jonsson, B., Parrow, J., eds.: CONCUR'94: Concurrency Theory. Volume 836 of Lecture Notes in Computer Science., Springer-Verlag (1994) 401–416
[17] Jensen, K.: Coloured Petri Nets. EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1992)
[18] Baeten, J.C.M., Bergstra, J.A.: Non interleaving process algebra. In Best, E., ed.: CONCUR'93, International Conference on Concurrency Theory. Volume 715 of Lecture Notes in Computer Science., Springer-Verlag (1993) 308–323
[19] Baeten, J.C.M., Basten, A.A.: Partial-order process algebra (and its relation to Petri nets). In Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier Science Publishers B.V., Amsterdam (2001) 769–872
[20] Mauw, S., Reniers, M.A.: A process algebra for interworkings. In Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier Science Publishers B.V., Amsterdam (2001)

[21] 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)
[22] Best, E., Devillers, R., Hall, J.: The Petri Box Calculus: A new causal algebra with multilabel communication. Volume 609 of Lecture Notes in Computer Science., Springer-Verlag (1992) 21–69
[23] Best, E., Devillers, R., Koutny, M.: Petri Net Algebra. EATCS Monographs on Theoretical Computer Science. Springer-Verlag (2001)
[24] Best, E., Devillers, R., Koutny, M.: A unified model for nets and process algebra. In Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier Science Publishers B.V., Amsterdam (2001) 873–944
[25] Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer-Verlag (1980)
[26] Willemse, T.A.C.: Semantics and Verification in Process Algebras with Data and Timing. PhD thesis, Technische Universiteit Eindhoven (TU/e) (2003)
[27] Plotkin, G.D.: A structural approach to operational semantics. Technical Report DIAMI FN-19, Computer Science Department, Aarhus University (1981)
[28] Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60-61** (2004) 17–139
[29] van Glabbeek, R.J.: The linear time - branching time spectrum II. In Best, E., ed.: CONCUR'93, International Conference on Concurrency Theory. Volume 715 of Lecture Notes in Computer Science., Springer-Verlag (1993) 66–81
[30] van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM **43**(3) (1996) 555–600
[31] Luttik, B.: Choice Quantification in Process Algebra. PhD thesis, University of Amsterdam (2002)
[32] van Glabbeek, R.: The linear time - branching time spectrum. In Baeten, J., Klop, J., eds.: CONCUR'90 - Theories of Concurrency: Unification and Extension. Volume 458 of Lecture Notes in Computer Science., Amsterdam, Springer-Verlag (1990) 278–297
[33] Baeten, J.C.M., Verhoef, C.: A congruence theorem for structured operational semantics with predicates. In Best, E., ed.: CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory. Volume 715., Springer-Verlag (1993) 477–492
[34] Meinke, K.: Universal algebra in higher types. Theoretical Computer Science **100**(2) (1992) 385–417
[35] Meinke, K.: 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., Springer-Verlag (1996) 124–143
[36] Möller, B., Tarlecki, A., Wirsing, M.: Algebraic specification of reachable higher-order algebras. In: Recent Trends in Data Type Specification. Volume 332 of Lecture Notes in Computer Science., Springer-Verlag (1988) 154–169
[37] Usenko, Y.S.: Linearization in μCRL. PhD thesis, Technische Universiteit Eindhoven (TU/e) (2002)
[38] Möller, B.: Algebraic specifications with higher-order operators. In: The IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Amsterdam, The Netherlands, North-Holland Publishing Co. (1987) 367–398
[39] Burris, S., Sankappanavar, H.: A Course in Universal Algebra. Springer-Verlag (1981)

[40] Loeckx, J., Ehrich, H.D., Wolf, M.: Specification of Abstract Data Types. Wiley (1996)
[41] Klusener, A.S.: Models and Axioms for a Fragment of Real Time Process Algebra. PhD thesis, Eindhoven University of Technology (1993)
[42] Groote, J.F., Springintveld, J.: Focus points and convergent process operators. A proof strategy for protocol verification. Technical Report 142, University Utrecht, Department of Philosophy (1995)
[43] van der Zwaag, M.: The cones and foci proof technique for timed transition systems. Information Processing Letters **80**(1) (2001) 33–40
[44] Groote, J.F., Sellink, M.P.A.: Confluence for process verification. Theoretical Computer Science **170**(1-2) (1996) 47–81