Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

The Syntax and Semantics of timed μCRL

J.F. Groote

Software Engineering (SEN)

SEN-R9709 June 30, 1997

Report SEN-R9709
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

The Syntax and Semantics of timed μCRL

Jan Friso Groote

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Department of Mathematics and Computing Science, Eindhoven University of Technology

P.O. Box 513, 5600 MB Eindhoven, The Netherlands

jfg@cwi.nl

Abstract

We define a specification language called '*timed μCRL*'. This language is designed to describe communicating processes employing data and time. Timed μCRL is the successor of μCRL [17]. It differs in two main aspects. It is possible to make explicit reference to time using a new '*at*' operator; $p \lhd t$ is the process p where the first action must take place at time t. Furthermore, a distinction has been made between constructors and functions in the datatypes. Care has been taken that every μCRL specification is also a correct timed μCRL specification with exactly the same meaning.

1991 Mathematics Subject Classification: 68M99, 68N99
1991 Computing Reviews Classification System: D.2.1, D.3.1, D.3.3
Keywords and Phrases: Specification Language, Abstract Data Types, Process Algebra, Operational Semantics, Real Time

1 Introduction

The language μCRL (*micro Common Representation Language*) has been defined to describe interacting processes that rely on data [17]. The major design objectives for μCRL were that

- μCRL had to be so expressive that 'real life systems', generally consisting of a set of interacting programs, could be described.
- μCRL had to be so simple and clear that it was suitable to form a basis for mathematical analysis.
- the definition of μCRL had to be sufficiently precise to allow for the independent construction of computer tools for μCRL capable in assisting in the actual development of systems.

We are confident to say that μCRL has indeed met all these intended purposes (see for example [5, 7, 8, 9, 15, 16, 18, 19, 21, 24, 28, 29, 33]). However, the need was felt to be able to deal with a more explicit notion of time. Moreover, the datatypes needed to be adapted to overcome some shortcomings, which are independent of the extension with time. Timed μCRL is the extension of μCRL in both respects. This extension is carried out in such a way that every μCRL specification is also a timed μCRL specification.

The real time extension

The language μCRL primarily describes the potential ordering of actions that processes can perform, which is called the behaviour of a process. It is easy to express that an action a must happen before an action b, and that action c happens in parallel with these. There is however no explicit reference to time. E.g. μCRL cannot express that action b must happen within five seconds after a, or if nothing is happening for 10 seconds, an *alarm* action is issued.

However, time is used explicitly in almost any computerized system. Computers rely on notions such as time-outs, scheduling and interrupts. Hardware and software clocks are common as are repeating time-based tasks. Real-time systems even add another dimension. These are required to perform certain tasks within predetermined time intervals. For such systems time is not only important for the internal behaviour, but time is also an important notion when it comes to the interaction with the environment.

Therefore, we have extended μCRL to handle time.

The major changes and major design considerations for the extension with time are given below.

- A specification in timed μCRL that makes no reference to time should look exactly the same as a 'classical' μCRL specification. Moreover, the intuitive meaning of such a specification should be equal in both languages. Formally, both semantics of this specification differ, as the semantics of timed μCRL explicitly refers to time, whereas the semantics of classical μCRL does not mention time at all.

- The extension with time should be natural and concise, fitting in the style of description of μCRL. We choose to extend μCRL with two operators, namely $x \triangleleft t$ (pronounce x at t), expressing that the first action of x must take place at time t, and $x \ll y$ (pronounce x before y) behaving as the process x that starts before y must have performed an action.

  We choose to let each term of sort time be an absolute reference to a moment in time, and not a reference relative to previous actions. We only selected one option, because descriptions with relative and absolute notions of time can easily be converted to each other. Terms denoting time are of sort **Time**, which must be declared in the datatypes whenever the 'at' operator is used. This sort has at least two functions, namely **0**, which is the starting point in time, and $\leq$ which is a total ordering on time. We leave it open whether **Time** is a discrete or dense domain, to be determined for each particular specification. Although $x \triangleleft t$ only refers to a single point in time, the sum operator $\Sigma_{t:\mathbf{Time}}$ in combination with the conditionals allows expression of any time interval.

- Timed μCRL should be as suitable for analysis and verification purposes as μCRL turned out to be. We did do a number of exercises with timed verifications. This has resulted in allowing simultaneous, subsequent actions. Concretely, it is allowed to write $a \triangleleft 1 \cdot b \triangleleft 1$, meaning that a takes place at time 1, followed by b taking place at time 1. Here we clearly sacrificed naturality of the language in favour of ease of use (as in [2, 23]). The most important reason to allow simultaneity is the elimination of the parallel operator in expressions of the form $a \triangleleft 1 \parallel b \triangleleft 1$. Using linear process operators [6], it is generally possible to eliminate the parallel operator without exponential blowup in this way. One of the alternatives that we investigated, is the use of multi-actions, e.g. $\langle a \cdot b \rangle \triangleleft 1$ [1], but this gives rise to an exponential number of multi-actions, when expanding n parallel processes, seriously hampering any verification effort.

Adaptation of the datatypes

In untimed μCRL, datatypes are defined by declaring a number of functions using the keyword **func** and by specifying via equations which terms must be considered equal. The declared functions are considered to be constructors of the datatype leading to the problems described below. We have added the possibility in timed μCRL to declare non-constructor functions using the keyword **map**.

- When specifying a data type with a small number of typical constructors, e.g. the natural numbers with zero and the successor function, and additionally some non constructor functions, such as addition, one is forced in μCRL to declare the non constructor functions as a constructor. This has as a consequence that whenever elementary properties about, say, the natural numbers are proven, induction on zero, the successor and addition has to be performed, or it has to be proven that addition can be eliminated from each term denoting a natural number. If addition is declared using **map** in timed μCRL it has the status of a non constructor function, and it is not necessary to incorporate it in the induction principle.
- It is not possible to declare an arbitrary constant in μCRL without specifying its value or introducing new elements. E.g. if a natural number is declared in μCRL by '**func** n', without specifying its value, it is taken as a new base natural number and $succ(n)$, $succ(succ(n))$, etc. are also new natural numbers. In timed μCRL this is easily solved by declaring '**map** n', which must be equal to some term denoted using the constructors 0 and $succ$.
- In μCRL it is impossible to declare a domain without determining the elements occurring in it. For instance declaring an arbitrary data domain D without any function with D as target domain would be the empty domain, which is explicitly forbidden in μCRL. In case a single element **func** $d_0 : \rightarrow D$ is declared this means that D_0 has exactly one single element d_0, and no other elements. However, often the intention of **func** $d_0 : \rightarrow D$ is that D is an arbitrary domain with at least a default element d_0. In timed μCRL this can correctly be expressed by '**map** $d_0 : \rightarrow D$'.

Changes in the operational semantics

The operational semantics of timed μCRL differs in two major aspects from the operational semantics of μCRL. In the first place every state has been extended with time. So, each state consists of a process and a moment in time. Secondly, transitions between states are labeled with actions with arguments of the form $a(t_1, \ldots, t_m)$ where t_i is a data term. As the introduction of **map** allows to have domains in the model of which not all elements can be denoted by terms, we introduce 'fresh' constants to the signature to denote these model elements.

Minor differences

The following minor alterations have been added in timed μCRL. Note that they have been defined such that every μCRL specification is still a correct timed μCRL specification.

In timed μCRL the possibility has been added to denote the initial state of a process by denoting **init** p where p is a closed process expression.

The properties of datatypes are described by equations. When defining μCRL it was assumed that these equations should be rewriting rules, as rewriting was the only conceived way of proving properties using the equations. This means that the left hand side of a rewriting rule was not allowed to be a single variable and the variables in the right hand side of an equations should also occur in the left hand side. In timed μCRL this is not required anymore.

It has been tried to make the definition of timed μCRL more accessible than the definition of μCRL. Therefore, the main text consists of an introduction to timed μCRL explaining how to specify in timed μCRL and how to carry out some elementary calculations. In Appendices A, B, C and D the syntax, static semantics, models of the datatype and the operational semantics of timed μCRL are defined, respectively.

Acknowledgements. I thank all persons who assisted in developing μCRL throughout the years. For timed μCRL I have benefited a lot from comments from Jos Baeten, Jan Bergstra, Bas Luttik, Alban Ponse, Michel Reniers, Jan Springintveld, Jan-Joris Vereijken and all people within COST 247, especially those developing Extended LOTOS [25], which also includes time.

variable	range
x, y, z	processes
X, Y	functions from data to processes
D	data sort
d, e	data terms of arbitrary sort
b	data terms of sort **Bool**
t, t', t_1, t_2, t_3	data terms of sort **Time**
c, c'	actions with one argument* ($a(d)$), τ or δ
a, a'	action names
*in the axioms we assume that each action has exactly one argument.	

Table 1: Conventions in the tables with axioms

2 The language

We provide an introduction into timed μCRL. Note that we use LaTeX type setting features at will. The only precise syntax for timed μCRL is given for plain text specifications (see Table 7 and Appendix A). This syntax is meant for specifications intended to be computer processed and for specifications of which their syntactically appearance must be unambiguously parsable. In expository texts we try to optimize readability.

2.1 Axioms

Before commencing to explain the language we spend a few words on the tables with axioms. For all operators in the language we provide characterizing axioms following the process algebraic tradition. Basically, they assist in interpreting the language correctly. The semantics of the language is given in terms of a data algebra and an operational semantics (see appendices C and D) and provides a rather operational understanding. The axioms provide a more syntactical perspective. Secondly, these laws are useful, as they explain why in some cases brackets may be omitted. For instance, it is allowed to write $a + b + c$, because, due to A2 in Table 2, it does not matter how brackets are put. The third reason for having these axioms is that they form the elementary basis for axiomatic reasoning about processes. We give an extensive set of examples of this in section 4. The conventions regarding the variables in the tables with axioms are summarized in Table 1.

2.2 Abstract data types

Processes in μCRL generally exchange data. In timed μCRL there is a simple, yet powerful mechanism to specify the data. We use (equational) abstract data types with an explicit distinction between constructor functions and 'normal' functions. The advantage of having such a simple language is that it can easily be explained and formally defined. Moreover, all properties of a datatype must be explicitly denoted, and henceforth it is clear which assumptions can be used when proving properties about data or processes. A disadvantage is of course that even the simplest datatypes must be specified each time, and that there are no high level constructs that allow compact specification of complex datatypes.

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

sort Bool

Because booleans are used in the if-then-else construct in the process language, the sort **Bool** must be declared in every timed μCRL specification.

Elements of a data type are declared by using the keywords **func** and **map**. Using **func** one can declare all elements in a datatype defining the structure of the datatype. E.g. by

sort **Bool**
func $\mathsf{t}, \mathsf{f}: \rightarrow$ **Bool**

one declares that t (true) and f (false) are the only elements of sort **Bool**. We say that t and f are the constructors of sort **Bool**. It is also obligatory, that t and f are declared in every specification. Moreover, it is assumed that t and f are the two (different) elements in **Bool**. This is expressed in axioms Bool1 and Bool2 in Table 2. In axiom Bool2 and elsewhere we use a variable b that can only be instantiated with data terms of sort **Bool**. If in a specification t and f can be proven equal, for instance if the specification contains an equation $\mathsf{t} = \mathsf{f}$, we say that the specification is inconsistent and it looses any reasonable meaning.

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

sort **Bool**, $\mathbb{N}$
func $\mathsf{t}, \mathsf{f}: \rightarrow$ **Bool**
$0: \rightarrow \mathbb{N}$
$S{:}\mathbb{N} \rightarrow \mathbb{N}$

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

If a sort D is declared without any constructor with target sort D, then it is assumed that D may be arbitrarily large. In particular D may contain elements that cannot be denoted by terms. This can be extremely useful, for instance when defining a data transfer protocol, that can transfer data elements from an arbitrary domain D. In such a case it suffices to declare in timed μCRL:

sort D

Note that in μCRL this was not allowed. According to its semantics this would yield an empty sort, as there were no constructors, and this was explicitly forbidden.

The keyword **map** is used to declare additional functions for a domain of which the structure is already given. For instance declaring a function $\wedge$ on the booleans, or declaring the $+$ on natural numbers can be done by adding the following lines to a specification in which $\mathbb{N}$ and **Bool** are already declared:

map $\wedge{:}$**Bool** $\times$ **Bool** $\rightarrow$ **Bool**
$+{:}\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

By adding plain equations, of the form $term = term$, assumptions about the functions can be added. Note that we use the keyword **rew** for equations confusingly suggesting that the equations must be rewrite rules. This is inherited from μCRL, where these equations were indeed intended to be rewrite rules. For the two functions declared above, we could add the equations:

var $x{:}$**Bool** $n, n'{:}\mathbb{N}$
rew $x \wedge \mathsf{t} = x$
$x \wedge \mathsf{f} = \mathsf{f}$
$n + 0 = n$
$n + S(n') = S(n + n')$

Note that before each group of the equations starting with the keyword **rew** we must declare the variables that are used.

Note also that although the machine readable syntax of timed μCRL in section A only uses prefix functions, we use these infix, if we believe that this increases readability. Moreover, we use common mathematical symbols such as $\wedge$ and $+$, which is also not allowed by the syntax of timed μCRL, for the same reason.

Ultimately, we remark that the equations for a datatype can be used in proving properties about data and process terms (see section 4.1).

If a specification refers to time, using the 'at' operator, it is obligatory to declare a domain **Time** as follows:

sort **Time**
map $\mathbf{0}: \rightarrow \mathbf{Time}$
$\leq : \mathbf{Time} \times \mathbf{Time} \rightarrow \mathbf{Bool}$

Instead of **map**, it is possible to use **func** for $\mathbf{0}$. The functions $\mathbf{0}$ and $\leq$ must satisfy the properties mentioned in Table 3, saying that $\leq$ is a total ordering and $\mathbf{0}$ is the smallest element. In order to formulate these properties, we have defined some auxiliary functions in this table.

Besides these requirement one is free to specify the kind of time one requires. This is done to accomodate those preferring discrete time, and others who prefer a notion of dense time. It is possible give **Time** any structure, as long as it satisfies the axioms in Table 3. One can for instance define that **Time** has only a finite number of elements, or one can define an ordinal like structure on it.

Discrete time can be specified as follows (omitting the specification of **Bool**):

sort **Time**, **Bool**
func $\mathbf{0}: \rightarrow \mathbf{Time}$
$next: \mathbf{Time} \rightarrow \mathbf{Time}$
$\mathsf{t}, \mathsf{f}: \rightarrow \mathbf{Bool}$
map $\leq : \mathbf{Time} \times \mathbf{Time} \rightarrow \mathbf{Bool}$
var $t, t': \mathbf{Time}$
rew $next(t) \leq \mathbf{0} = \mathsf{f}$
$next(t) \leq next(t') = t \leq t'$

Note that we do not explicitly state that $\mathbf{0} \leq t = \mathsf{t}$, because this is already in Table 3.

Functions may be overloaded, as long as every term has a unique sort. This means that the name of the function together with the sort of its arguments must be unique. E.g. it is possible to declare $max: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ and $max: \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}$, but is not allowed to declare a function $f: \mathbf{Bool} \rightarrow \mathbf{Bool}$ and $f: \mathbf{Bool} \rightarrow \mathbb{N}$. Actually, the overloading rule holds in general in timed μCRL such that every term is either an action, process name or a data term, and if it is a data term, it has a unique sort.

When declaring a sort D, it is required that it may not be empty. Therefore, the following declaration is invalid.

sort D
func $f: D \rightarrow D$

It declares that D is a domain in which all the terms have the form $f(f(f(\ldots)))$, i.e. an infinite number of applications of f. Such terms do not exist, and therefore D must be empty. This problem can also occur with more than one sort. E.g. sorts D and E with constructors from D to E and E to D. Fortunately, it is easy to detect such problems and therefore it is a static semantic constraint that such empty sorts may not occur (see Appendix B).

2.3 Actions

Actions are abstract representations of events in the real world that is described. For instance sending the number 3 can be described by $send(3)$ and boiling food can be described by $boil(food)$ where 3 and *food* are terms belonging to some sort. An action consists of a name possibly followed by one or more data terms within brackets. Actions are declared using the keyword **act** followed by a name and the sorts of the data that it can exchange. Below, we declare the action name *timeout* over which no data is transferred, and an action a that is used to transfer booleans, and pairs of natural numbers and data elements of sort D.

act *timeout*
a:**Bool**
a:$\mathbb{N} \times D$

In the tables with axioms we use the letter a for an action label, and we give each action a single argument, although in μCRL these actions may have zero or more than one argument. The letter c is used for actions with arguments, and for the constants τ and δ.

2.4 Occurrences of processes

Processes are expressions built upon actions indicating when a process may engage in certain events. In other words, a process represents the potential behaviour of a certain system.

In a timed μCRL specification processes appear at two places. First, there can be a single occurrence of an initial declaration, of the form

init p

where p is a process expression indicating the initial behaviour of the system that is described. The init section may be omitted, in which case the initial behaviour of the system is left unspecified.

The other place where process expressions may occur are in the right hand side of process declarations, which have the form:

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

Here X is the process name, the x_i are distinct variables, not clashing with the name of a constant function or a parameterless process or action name, and the s_i are sort names. In this rule, process $X(x_1, \ldots, x_n)$ is declared to have the same (potential) behaviour as the process expression p.

2.5 Basic processes

The two elementary operators to construct processes are the sequential composition operator, written as $p{\cdot}q$ and the alternative composition operator, written as $p + q$. The process $p{\cdot}q$ first performs the actions of p, until p terminates, and then continues with the actions in q. It is common to omit the sequential composition operator in process expressions. The process $p + q$, behaves like p or q, depending on which of the two performs the first action. Using the actions declared above, we can describe that action $a(3, d)$ must be performed, except if a time out occurs, in which case $a(\mathsf{t})$ must happen.

$$a(3, d) + timeout{\cdot}a(\mathsf{t})$$

Note that the sequential operator binds stronger than the alternative composition operator.

The meaning of all process operators is given in terms of operational semantics in Appendix D. We will address a few issues about operational semantics in the next section, and skip further treatment here.

A1	$x+y=y+x$	SUM1	$\sum_{d:D} x = x$
A2	$x+(y+z)=(x+y)+z$	SUM3	$\sum X = \sum X + Xd$
A3	$x+x=x$	SUM4	$\sum_{d:D}(Xd+Yd) = \sum X + \sum Y$
A4	$(x+y)\cdot z = x\cdot z + y\cdot z$	SUM5	$(\sum X)\cdot x = \sum_{d:D}(Xd\cdot x)$
A5	$(x\cdot y)\cdot z = x\cdot(y\cdot z)$	SUM11	$(\forall d{\in}D\ Xd = Yd) \rightarrow \sum X = \sum Y$
AT6	$x+\delta\lhd\mathbf{0} = x$		
A7	$\delta\cdot x = \delta$		
Bool1	$\neg(\mathsf{t}=\mathsf{f})$	C1	$x\lhd\mathsf{t}\rhd y = x$
Bool2	$\neg(b=\mathsf{t})\rightarrow b=\mathsf{f}$	C2	$x\lhd\mathsf{f}\rhd y = y$

Table 2: Basic axioms for timed mCRL

In Table 2 axioms A1-A5 are listed describing the elementary properties of the sequential and alternative composition operators. For instance, the axioms A1, A2 and A3 express that $+$ is commutative, associative and idempotent. In these and other axioms we use variables x, y and z that can be instantiated by process terms. Note that for process terms we use the letters p, q and r.

2.6 Deadlock and encapsulation

Timed μCRL contains a constant δ, expressing that from now on, no action can be performed. It models the situation where no action can be performed, for instance in case a number of computers are waiting for each other, and henceforth not performing any action anymore. Henceforth, this constant is called *deadlock*. A typical property for δ is $p+\delta=p$, provided p does not refer to time; the choice in $p+q$ is determined by the first action performed by either p or q, and therefore one can never choose for δ. However, if p refers to time, it can be that p must perform its first action before certain time. If that action does not occur before that time, p cannot be chosen anymore, and $p+\delta$ acts like δ. In Table 2 the axiom AT6 is the variant of $x+\delta=x$, where time is incorporated.

Another property of δ is $\delta\cdot p=\delta$. It says that whatever is after δ cannot be reached. It is formulated as A7 in Table 2.

Sometimes, we want to express that certain actions cannot happen, and must be blocked. Generally, this is only done when we want to force this action into a communication. The encapsulation operator ∂_H is specially designed for this task. In $\partial_H(p)$ it prevents all actions of which the action name is mentioned in H from happening. Typically,

$$\partial_{\{b\}}(a\cdot b(3)\cdot c) = a\cdot\delta$$

where a, b and c are actions. The properties of ∂_H are mentioned in Table 5.

2.7 Timed processes

A novel feature of timed μCRL is that it can be expressed at which time certain actions must take place. This is done using the *at* operator. The process $p@t$ (we use the letter t for terms of sort **Time**), behaves like the process p, with the restriction that the first action of p must take place at time t. So, assuming the natural numbers denote moments in time, the following process denotes that actions a, b and c must take place at times 1, 2 and 3, respectively.

$$a@1\cdot b@2\cdot c@3$$

Time1	$(t_1 \leq t_2 \vee t_2 \leq t_3 = \mathsf{t}) \rightarrow t_1 \leq t_3 = \mathsf{t}$		
Time2	$\mathbf{0} \leq t = \mathsf{t}$		
Time3	$t_1 \leq t_2 \vee t_2 \leq t_1 = \mathsf{t}$		
Time4	$(t_1 \leq t_2 \wedge t_2 \leq t_1 = \mathsf{t}) \leftrightarrow t_1 = t_2$		
Time5	$eq(t_1, t_2) = t_1 \leq t_2 \wedge t_2 \leq t_1$	Bool3	$\neg\mathsf{t} = \mathsf{f}$
Time6	$min(t_1, t_2) = if(t_1 \leq t_2, t_1, t_2)$	Bool4	$\neg\mathsf{f} = \mathsf{t}$
Time7	$if(\mathsf{t}, t_1, t_2) = t_1$	Bool5	$b \wedge \mathsf{t} = b$
Time8	$if(\mathsf{f}, t_1, t_2) = t_2$	Bool6	$b \wedge \mathsf{f} = \mathsf{f}$
		Bool7	$b \vee \mathsf{t} = \mathsf{t}$
		Bool8	$b \vee \mathsf{f} = b$

Table 3: Properties for **Time** and auxiliary functions for **Bool** and **Time**

ATA1	$x = \sum_{t:\mathbf{Time}} x \lhd t$
ATA2	$\delta \lhd t \lhd t \leq t' \rhd \delta \lhd \mathbf{0} + c \lhd t' = c \lhd t'$
ATA3	$c \lhd t \cdot x = c \lhd t \cdot (\sum_{t':\mathbf{Time}} x \lhd t' \lhd t \leq t' \rhd \delta \lhd \mathbf{0} + \delta \lhd t)$
ATB1	$\delta \lhd t \lhd t' = \delta \lhd min(t, t')$
ATB2	$c \lhd t \lhd t' = c \lhd t \lhd eq(t, t') \rhd \delta \lhd min(t, t')$
ATB3	$(x + y) \lhd t = x \lhd t + y \lhd t$
ATB4	$(x \cdot y) \lhd t = x \lhd t \cdot y$
ATB5	$(\sum_{d:D} Xd) \lhd t = \sum_{d:D} Xd \lhd t$
ATB6	$(x \between y) \lhd t = x \lhd t \between y$
ATB7	$(x \mid y) \lhd t = x \lhd t \mid y$
ATB8	$(x \mid y) \lhd t = x \mid y \lhd t$
ATB9	$\partial_H(x \lhd t) = \partial_H(x) \lhd t$
ATB10	$\tau_I(x \lhd t) = \tau_I(x) \lhd t$
ATB11	$\rho_R(x \lhd t) = \rho_R(x) \lhd t$
$\ll$1	$x \ll c = x$
$\ll$2	$x \ll (y + z) = x \ll y + x \ll z$
$\ll$3	$x \ll y \cdot z = x \ll y$
$\ll$4	$x \ll \sum X = \sum_{d:D} x \ll Xd$
$\ll$5	$x \ll y \lhd t = \sum_{t':\mathbf{Time}} (x \ll y) \lhd t' \lhd t' \leq t \rhd \delta \lhd \mathbf{0}$

Table 4: Time related axioms for timed μCRL

If an action happens at time t, then a subsequent action can take place at a time t or higher. This means that in the first and second example below the b action can happen, and in the third example b is blocked.

$a \cdot 2 \cdot b \cdot 3$
$a \cdot 2 \cdot b \cdot 2$
$a \cdot 2 \cdot b \cdot 1$

Actually, the last example above is equivalent to $a \cdot 2 \cdot \delta \cdot 2$, saying that after action a takes place, we have a time deadlock. In order to let b take place as prescribed, we have to reverse time. As this is clearly in conflict with reality, we choose to stop time at time 2 ($\delta \cdot 2$). This is our general approach. Whenever, a specification prescribes timing behaviour that cannot be realised, it will exhibit time deadlocks. So, before implementing a timed μCRL specification, it is a good habit to prove it time deadlock free.

Every process starts at time $\mathbf{0}$.

In order to be able to provide an axiomatisation for the parallel operator, the bounded initialization operator $\ll$ is introduced. The process $p \ll q$ behaves like p, where the first action of p must happen before last moment on which the first action of q can happen. If this cannot be the case, $p \ll q$ is a deadlock at the latest time the first action of q could have happened. Although the operator differs slightly from its similarily named operator in [1], we have decided to give it the same name.

Both the bounded initialisation and the 'at' operator are axiomatized using the axioms in Table 4. For these axioms we have used three auxiliary functions, namely $min(t, t')$, giving the earliest of t and t', $eq(t, t')$ determining whether t and t' are the same moments in time, and $if(b, t, t')$ yielding t if b equals true, and t' otherwise. These auxiliary functions are defined in Table 3.

2.8 Sums and conditionals

The process expression $p \lhd b \rhd q$ where p and q are processes, and b is a data term of sort **Bool**, behaves like p if b is equal to t (true) and if b is equal to f (false), behaves like q. This operator is called the conditional operator, and operates precisely as a *then_if_else* construct. Using the conditional operator data influences process behaviour. For instance a counter, that counts the number of a actions that occur, issuing a b action and resetting the internal counter after 10 a's, can be described by $Counter(0)$ where $Counter$ is declared by (we omit declaring the datatypes):

proc $Counter(n{:}\mathbb{N}) = a \cdot Counter(n+1) \lhd n < 10 \rhd b \cdot Counter(0)$

The conditional operator is characterised by axioms C1 and C2 in Table 2. There are no more axioms needed, because all properties about the conditionals appear to be provable using axioms Bool1 and Bool2.

The sum operator $\sum_{d:D} P(d)$ behaves like $P(d_1) + P(d_2) + \cdots$, i.e. as the choice between $P(d_i)$ for any data term d_i taken from D. This is generally used to describe a process that is reading some input. E.g. in the following example we describe a single place buffer, repeatedly reading a value d using action name r, and then delivering that value via action name s.

proc $Buffer = \sum_{d:D} r(d) \cdot s(d) \cdot Buffer$

In Tables 2, 4 and 5 axioms for the sum operator are listed. The sum operator is a difficult operator, because it acts as a binder just like the lambda in the lambda calculus (see e.g. [4]). This introduces a range of problems with substitutions. In order to avoid having to deal with these explicitly, we allow the use of explicit lambda operators and variables representing functions from data to processes.

In the Tables the variables x, y and z may be instantiated with processes and the capital variables X and Y can be instantiated with functions from some data sort to processes. The sum operator $\sum$ expects a function from a datatype to a process, whereas $\sum_{d:D}$ expects a process. When we substitute

a process p for a variable x or a function $\lambda d{:}D.p$ for a variable X in the scope of a sum operator, no variable in p may become bound. If this appears to happen, we must first rename the variable d into, say e, provided e does not occur in p. This renaming is called α-conversion. We consider processes modulo α-conversion, so the expressions $\sum_{d:D} p(d)$ and $\sum_{e:D} p(e)$ are equal. Consequently, we may only substitute the action $a(d)$ for x in the left hand side of SUM1 in Table 2, after renaming d into e. So, SUM1 is a concise way of saying that if d does not appear in p, then we may omit the sum operator in $\sum_{d:D} p$.

As another example, consider axiom SUM4. It says that we may distribute the sum operator over a plus, even if the sum binds a variable. This can be seen by substituting for X and Y $\lambda d{:}D.a(d)$, and $\lambda d{:}D.b(d)$, where no variable becomes bound. After β-reduction, the left hand side of SUM4 becomes $\sum_{d:D}(a(d)+b(d))$ and the right hand side is $\sum_{d:D} a(d) + \sum_{d:D} b(d)$.

Although, the sum and conditional operator seem rather straightforward operators on first sight, they have considerably strengthened the applicability and power of process algebra.

2.9 Internal actions and hiding

Abstraction is an important means to analyse communicating systems. It means that certain actions are made invisible, such that the relationship between the remaining actions becomes more clear. A specification can be proven equal to an implementation, consisting of a number of parallel processes, after hiding all communications between these components.

The hidden action or internal action is denoted by τ. It represents an action that can take place in a system, but that cannot be observed directly. The internal action is meant for analysis purposes, and hardly ever used in specifications, as it is very uncommon to specify that something unobservable must happen. We consider the treatment of internal actions, including formulating axioms that allow to manipulate them, beyond our current scope.

In order to make actions hidden, the hiding operator τ_I is introduced, where I is a set of action names. The process $\tau_I(p)$ behaves as the process p, except that all actions with action names in I are renamed to τ. This is clearly characterized by the axioms in Table 6 and axiom ATB10 in Table 4.

2.10 Parallel processes

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

The parallel operator allows to describe intricate processes. For instance a bag reading natural numbers using action name r and delivering those via action name s can be described by:

act $r, s{:}\mathbb{N}$
proc $Bag = \sum_{n:\mathbb{N}} r(n){\cdot}(s(n) \parallel Bag)$

Note that the elementary property of bags, namely that at most as many numbers can be delivered as have been received in the past, is satisfied by this description.

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

comm $a \mid b = c$

This means that if actions $a(d_1, \ldots, d_n)$ and $b(d_1, \ldots, d_n)$ can happen in parallel, they may synchronize and this synchronization is denoted by $c(d_1, \ldots, d_n)$. If two actions synchronize, their arguments must be the same. In a communication declaration it is required that action names a, b and c are declared with exactly the same data sorts. It is not necessary that these sorts are unique. It is for example perfectly right to declare the three actions both with a sort $\mathbb{N}$ and with a pair of sorts $D \times \mathbf{Bool}$.

SUM6	$(\sum X) \between x = \sum_{d:D}(Xd \between x)$	CF	$a(d)\|a'(e) = \begin{cases} \gamma(a,a')(d) \lhd eq(d,e) \rhd \delta & \text{if sorts of } d \text{ and } e \text{ are equal,} \\ & \text{and } \gamma(a,a') \text{ defined} \\ \delta & \text{otherwise} \end{cases}$
SUM7	$(\sum X)\|x = \sum_{d:D}(Xd\|x)$		
SUM7′	$x\|(\sum X) = \sum_{d:D}(x\|Xd)$		
SUM8	$\partial_H(\sum X) = \sum_{d:D} \partial_H(Xd)$		
SUM9	$\tau_I(\sum X) = \sum_{d:D} \tau_I(Xd)$		
SUM10	$\rho_R(\sum X) = \sum_{d:D} \rho_R(Xd)$	CD1	$\delta\|c = \delta$
		CD2	$c\|\delta = \delta$
CM1	$x \parallel y = x \between y + y \between x + x\|y$	CT1	$\tau\|c = \delta$
CM2	$c \between x = (c \ll x)\cdot x$	CT2	$c\|\tau = \delta$
CM3	$c\cdot x \between y = (c \ll y)\cdot(x \parallel y)$		
CM4	$(x+y) \between z = x \between z + y \between z$	DD	$\partial_H(\delta) = \delta$
CM5	$c\cdot x\|c' = (c\|c')\cdot x$	DT	$\partial_H(\tau) = \tau$
CM6	$c\|c'\cdot x = (c\|c')\cdot x$	D1	$\partial_H(a(d)) = a(d)$ if $a \notin H$
CM7	$c\cdot x\|c'\cdot y = (c\|c')\cdot(x \parallel y)$	D2	$\partial_H(a(d)) = \delta$ if $a \in H$
CM8	$(x+y)\|z = x\|z + y\|z$	D3	$\partial_H(x+y) = \partial_H(x) + \partial_H(y)$
CM9	$x\|(y+z) = x\|y + x\|z$	D4	$\partial_H(x\cdot y) = \partial_H(x)\cdot\partial_H(y)$

Table 5: Axioms for parallellism in timed μCRL

If a communication is declared as above, synchronisation is another possibility for parallel processes. For example the process $a \parallel b$ is now equivalent to $a\cdot b + b\cdot a + c$. Generally, this is not quite what is desired, as the intention generally is that a and b do not happen on their own. Therefore, the encapsulation operator can be used. The process $\partial_{\{a,b\}}(a \parallel b)$ is equal to c.

Data transfer between parallel components occurs very often. Here we describe as an example a simplified instance of data transfer. One process sends a natural number n via action name s, and another process reads it, via action name r and then announces it via action name a. Using encapsulation and hiding operators we enforce the processes to communicate, and make the communication internal. We omit declaring the datatypes.

map n
act $r, s, c, a{:}\mathbb{N}$
comm $r \mid s = c$
proc $p = \tau_{\{c\}}\partial_{\{r,s\}}(s(n) \parallel \sum_{m:\mathbb{N}} r(m)\cdot a(m))$

The process p behaves the same as $\tau\cdot a(n)$, as is to be expected.

Processes that are put in parallel can have time constraints on their actions that are used for communication. It may happen that these constraints are incompatible. For instance a process can be able to perform a read action r before or at time 10, whereas the corresponding send action occurs at time 11. This is expressed by $\partial_{\{r,s\}}(\sum_{t:\mathbf{Time}}(r(t) \lhd t \leq 10 \rhd \delta \lhd \mathbf{0} \parallel s \lhd 11))$. This process is equivalent to $\delta \lhd 10$, expressing that up till time 10, the process can execute without violating any timing constraint, but to reach any moment in time larger than 10, a timing constraint of an individual process must be ignored. In timed μCRL the timing constraints of individual processes are always respected.

Axioms that describe the parallel operator are in Tables 4 and 5. In order to formulate the axioms two auxiliary parallel operators have been defined. The left merge $\between$ is a binary operator that behaves exactly as the parallel operator, except that its first action must come from the left hand side. The communication merge $\mid$ is also a binary operator behaving as the parallel operator, except that the first action must be a synchronisation between its left and right operand. The core law for the parallel operator is CM1 in Table 5. It says that in $x \parallel y$ either x performs the first step, represented by

TID	$\tau_I(\delta) = \delta$	RD	$\rho_R(\delta) = \delta$
TIT	$\tau_I(\tau) = \tau$	RT	$\rho_R(\tau) = \tau$
TI1	$\tau_I(a(d)) = a(d)$ if $a \notin I$	R1	$\rho_R(a(d)) = R(a)(d)$
TI2	$\tau_I(a(d)) = \tau$ if $a \in I$		
TI3	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$	R3	$\rho_R(x + y) = \rho_R(x) + \rho_R(y)$
TI4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$	R4	$\rho_R(x \cdot y) = \rho_R(x) \cdot \rho_R(y)$

Table 6: Axioms for hiding and renaming in timed μCRL

the summand $x \between y$, or y can do the first step, represented by $y \between x$, or the first step of $x \parallel y$ is a communication between x and y, represented by $x \mid y$. All other axioms in Tables 4 and 5 are designed to eliminate the parallel operators in favour of the alternative and the sequential operator.

2.11 Renaming

In some cases it is efficient to reuse a given specification with different action names. This for instance allows the definition of generic components that can be used in different configurations. The renaming operator ρ_R is suited for this purpose. The subscript R is a sequence of renamings of the form $a \to b$, meaning that action name a must be renamed to b. As R is supposed to be a function every action name at the left hand side of an arrow must be unique in R. The process $\rho_R(p)$ behaves like the process p with its action names renamed according to R. An equational characterisation of the renaming operator can be found in Tables 4, 5 and 6

2.12 Notational conventions

As stated elsewhere, we generally do not denote the sequential composition operator.

Furthermore, instead of writing $\sum_{t:\mathbf{Time}} p(t) \lhd c(t) \rhd \delta \lhd \mathbf{0}$ indicating the process p being executed at times t for which $c(t)$ holds, we sometimes write the more readable $\sum_{t:c(t)} p(t)$. The t at the left hand side of the colon indicates that it is the variable that is bound in the sum operator.

2.13 Fischer's mutual exclusion protocol

As an example we describe Fischer's mutual exclusion protocol [30, 34]. This protocol guarantees that only one single party can be in a critical section at any given time. The idea is that an application process can ask the protocol, using a *req*uest action, to organise exclusive access for some critical region. The protocol then executes a simple program, and will after some time respond with an *enter* action, indicating that the application process has now exclusive access. When the application process is ready, it issues a *leave* instruction, indicating to the protocol that it does not require exclusive access anymore.

We describe this protocol for only two parties, although it can be used for any arbitrary number. The main idea behind the protocol is that each party after being asked to access a critical region, checks whether the common variable x equals 0, indicating that no process has claimed access. It then quickly sets the variable to its own sequence number to claim the critical section, and checks whether the variable has still the same value after a sufficiently long delay, guaranteeing that no other process will get access to the section. When leaving the critical region, the protocol sets the variable x back to 0.

Before specifying the process behaviour we describe standard datatypes, where eq on natural numbers represents the equality function, and d is a positive time duration. As we assume that the sorts

and functions are self evident, we do not explain these further.

The processes asking for access use a common variable x, which is modelled as a process X. As can be seen, the variable x can be set to a new value, and can be tested for the value it contains.

There are two processes that control gaining access, namely $FP(1)$ and $FP(2)$. If $FP(n)$ $(n = 1, 2)$ is asked to arrange access to the critical region, it carries out a protocol described in $FP_0(n)$. First it checks whether variable x is equal to 0, and records the time at which this action succesfully takes place in variable t. Then, within d time, it sets x to value n, and subsequently after at least d time reads the value of x. If this value is equal to n, $FP_0(n)$ terminates, allowing the *enter* action to happen. Otherwise, it starts the protocol described in $FP_0(n)$ over again.

Note that it is assumed in this specification that the shared variable can be accessed at any time. This may not be realistic, and if so, it must explicitly be modelled that shared variables have certain response times and cannot be accessed too often in a certain time interval. We do not address this aspect here.

sort $\mathbf{Time}\ \mathbb{N}\ \mathbf{Bool}$

func $\mathsf{t}, \mathsf{f}: \to \mathbf{Bool}$
$0: \to \mathbb{N}$
$S: \mathbb{N} \to \mathbb{N}$

map $\mathbf{0}: \to \mathbf{Time}$
$\leq, > : \mathbf{Time} \times \mathbf{Time} \to \mathbf{Bool}$
$+: \mathbf{Time} \times \mathbf{Time} \to \mathbf{Time}$
$1, 2: \to \mathbb{N}$
$eq: \mathbb{N} \times \mathbb{N} \to \mathbf{Bool}$
$d: \to \mathbf{Time}$
$\neg: \mathbf{Bool} \to \mathbf{Bool}$

var $t_1, t_2, u: \mathbf{Time}$
$x, y: \mathbb{N}$

rew $1 = S(0)$
$2 = S(1)$
$eq(0, 0) = \mathsf{t}$
$eq(0, S(x)) = \mathsf{f}$
$eq(S(x), 0) = \mathsf{f}$
$eq(S(x), S(y)) = eq(x, y)$
$t_1 + u \leq t_2 + u = t_1 \leq t_2$
$t_1 \leq t_1 + u = \mathsf{t}$
$t_1 > t_2 = \neg(t_1 \leq t_2)$
$\neg\mathsf{f} = \mathsf{t}$
$\neg\mathsf{t} = \mathsf{f}$

act $req, enter, leave, set_x, \overline{set}_x, set^\star_x, test_x, \overline{test}_x, test^\star_x: \mathbb{N}$

comm $set_x | \overline{set}_x = set^\star_x$
$test_x | \overline{test}_x = test^\star_x$

proc $X(x{:}\mathbb{N}) = \sum_{y:\mathbb{N}} \overline{set}_x(y)\, X(y) + \overline{test}_x(x)\, X(x)$

$FP(n{:}\mathbb{N}) = req(n)\, FP_0(n)\, enter(n)\, leave(n)\, set_x(0)\, FP(n)$
$FP_0(n{:}\mathbb{N}) = \sum_{t:\mathbf{Time}} test_x(0)\langle t \sum_{t':t'\leq t+d} set_x(n)\langle t'$
$\sum_{t'':t''>t'+d} \sum_{m:\mathbb{N}} (\overline{test}_x(m)\langle t'' \lhd eq(m,n) \rhd test_x(m)\langle t''\, FP_0(n))$

init $\tau_{\{set^\star_x, test^\star_x\}} \partial_{\{set_x, test_x, \overline{set}_x, \overline{test}_x\}} (FP(1) \parallel X(0) \parallel FP(2))$

3 Algebras, operational semantics and strong bisimulation

In order to establish a common intuition about the meaning of each data term a model in the form of a data algebra is provided and to establish an intuition for process terms, a transition system is associated to it.

Data algebras are defined in section C. The basic idea is that for every declared sort there is a set of elements in the data algebra. Every data term is interpreted as such an element. A data algebra is called a model if all equations of a datatype hold in the algebra.

For the semantics of a process term, we use a timed transition system, which is defined in Definitions D.2 and D.3. A timed transition system essentially tells under which actions one can traverse from one state to another. States are pairs of a process term and a moment in time. There are two sorts of transitions, namely action transitions and timing transitions.

Action transitions are labelled by an action name, followed by zero or more elements from an algebraic domain. We follow the urgent model of [2, 27] and choose that the occurrence of an action does not take time. I.e. if

$$\langle p, t\rangle \xrightarrow{a(\vec{u})} \langle p', t'\rangle,$$

then $t = t'$. The rules in Tables 9, 10 and 11 define when a transition can do a step. These rules are provided in the SOS style [32, 20].

Time progresses through time transitions which are labelled with an idle step $\imath$, e.g.

$$\langle p, t\rangle \xrightarrow{\imath} \langle p, t'\rangle$$

expresses that process p idles from t to t'. The rule in Table 8 together with Definition D.1 of the ultimate delay define when idle steps can be performed.

Given the transition relation we have a good understanding of the steps that can be performed by a process. An obvious question is when two processes are actually equivalent. Such a notion is required to justify the axioms. There is no clear answer given in the literature (see [12, 13] for an overview), but there is a general consensus that strong bisimulation is the finest equivalence of all potential candidates, and henceforth, that two processes are certainly equivalent if they are strongly bisimilar. Therefore, we provide below timed strong bisimulation.

Definition 3.1. Let E be a specification and let $\boldsymbol{A}$ be a model for the datatypes in E. Let $\mathbb{P}$ be the set of all process terms, and let $\surd$ be a special symbol indicating termination. Moreover, let $D_{\mathbf{Time}}$ be the domain in which elements of sort **Time** are interpreted (see Definition C.6. A collection of symmetric relations $\langle R_t^{\boldsymbol{A}} \subseteq (\mathbb{P} \cup \{\surd\}) \times (\mathbb{P} \cup \{\surd\}) | t \in D_{\mathbf{Time}}\rangle$ is called a *timed (strong) bisimulation* iff for all $p, q \in \mathbb{P} \cup \{\surd\}$ and $t \in D_{\mathbf{Time}}$ such that $pR_t^{\boldsymbol{A}}q$, it holds that:

1. if $p \equiv \surd$, then $q \equiv \surd$,
2. if $\langle p, t\rangle \xrightarrow{l} \langle p', t\rangle$, then there is a q' such that $\langle q, t\rangle \xrightarrow{l} \langle q', t\rangle$, and $p' R_t^{\boldsymbol{A}} q'$.
3. if $\langle p, t\rangle \xrightarrow{\imath} \langle p, t'\rangle$ then $\langle q, t\rangle \xrightarrow{\imath} \langle q, t'\rangle$ and $pR_{t'}^{\boldsymbol{A}}q$.

We write $p \leftrightarrows^t q$ iff for model $\boldsymbol{A}$ there is a timed (strong) bisimulation $\langle R_t^{\boldsymbol{A}} | t \in D_{\mathbf{Time}}\rangle$ such that $pR_t q$, and we write $p \leftrightarrows q$ if for all $t \in D_{\mathbf{Time}}$ $p \leftrightarrows^t q$. In this case we say that p and q are *timed (strongly) bisimilar*.

Timed bisimulation is a congruence for the process operators introduced in this text and all axioms are sound with respect to it. There has not been any attempt to provide a complete set of axioms. For a treatment of completeness issues, as well as weak timed bisimulations, the reader is for instance referred to [26, 27, 10].

4 Examples of simple verifications

An important usage of the axioms of timed μCRL is to verify the correctness of distributed systems. Manipulating terms using the axioms is not always easy, but generally boils down to a typical number of ever occuring steps. In order to allow a deeper understanding of the language and its axioms we provide a number of typical verification examples. We restrict ourselves to essentially non recursive processes and do not employ the hidden nature of τ. For these we need rules such as the recursive specification principle (RSP) and tau laws for timed process algebra, which we have not provided.

4.1 Data

4.1.1 Proving terms unequal

In μCRL it is possible to establish when two data terms are not equal. This is for instance required in order to establish that two processes cannot communicate. There is a characteristic way of proving that terms are not equal, namely by assuming that they are equal, and showing that this implies $\mathsf{t} = \mathsf{f}$, contradicting axiom Bool1.

We give an example showing that the natural numbers zero (0) and one ($S(0)$) are not equal. We assume that the natural numbers with a 0 and successor function S are appropriately declared. In order to show zero and one different, we need a function that relates $\mathbb{N}$ to **Bool**. Note that if there is no such function, there are models of the datatype $\mathbb{N}$ where zero and one are equal, and then we can of course not prove them different. For our function we choose 'less or equal than' on the natural numbers, defined as follows:

map $\leq : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
var $n, m : \mathbb{N}$
rew $0 \leq n = \mathsf{t}$
$S(n) \leq 0 = \mathsf{f}$
$S(n) \leq S(m) = n \leq m$

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

4.1.2 Using induction on Bool

An easy and very convenient axiom is Bool2. It says that if a boolean term b is not equal to true, it must be equal to f or in other words that there are at most two boolean values. Applying this axiom boils down to a case distinction, proving a statement for the values true and false, and concluding that the property must then universally hold. We refer to this style of proof by the phrase 'induction on booleans'.

A typical example is the proof of $b \wedge b = b$. Using induction on **Bool**, it suffices to prove that this equality hold for $b = \mathsf{t}$ and $b = \mathsf{f}$. In other words, we must show that $\mathsf{t} \wedge \mathsf{t} = \mathsf{t}$ and $\mathsf{f} \wedge \mathsf{f} = \mathsf{f}$. These are trivial instances of axioms Bool5 and Bool6 in Table 3.

Note that among many others, the equalities $b \vee b = b$ and $\neg\neg b = b$ can be shown in the same way.

4.1.3 Identities about time

We list a number of identities about time which we use later on. Although the identities appear very straightforward, their proofs sometimes are slightly tricky. In these cases some twist is needed to derive these identities from the axioms. We provide proofs of the first two equations.

Lemma 4.1.

1. $eq(t_0, t_1) = \mathsf{t}$ iff $t_0 = t_1$;

2. $t_1 \leq t_0 = \mathsf{t} \rightarrow min(t_0, t_1) = t_1$;
3. $min(t_0, t_1) = min(t_1, t_0)$;
4. $t_0 \leq t_1 \wedge t_0 \leq t_2 = t_0 \leq min(t_1, t_2)$;
5. $t_0 \leq min(t_0, t_1) = t_0 \leq t_1$.

Proof.

1. Using axioms Time3 and Time4 from Table 3 this is trivial.
2. We prove this fact by a case distinction on $t_0 \leq t_1$.
   - $t_0 \leq t_1 = \mathsf{f}$) It follows that
     $$min(t_0, t_1) \overset{\text{Time6}}{=} \mathit{if}(t_0 \leq t_1, t_0, t_1) \overset{\text{assumption}}{=} \mathit{if}(\mathsf{f}, t_0, t_1) \overset{\text{Time8}}{=} t_1.$$
   - $t_0 \leq t_1 = \mathsf{t}$) This side is somewhat more involved. First we show that t_0 and t_1 are equal:
     $$eq(t_0, t_1) \overset{\text{Time4}}{=} t_0 \leq t_1 \wedge t_1 \leq t_0 \overset{\text{assumptions}}{=} \mathsf{t} \wedge \mathsf{t} \overset{\text{Bool5}}{=} \mathsf{t}.$$
     So, using Lemma 4.1, it follows that $t_0 = t_1$. Now the proof is trivial:
     $$min(t_0, t_1) \overset{\text{Time6}}{=} \mathit{if}(t_0 \leq t_1, t_0, t_1) \overset{\text{assumption}}{=} \mathit{if}(\mathsf{t}, t_0, t_1) \overset{\text{Time7}}{=} t_0 = t_1.$$

□

4.1.4 Induction on $\mathbb{N}$

The division between constructors and mappings gives us induction principles. Suppose we have declared the natural numbers with constructors zero and successor. If we extend this with a mapping *add* as follows:

map $add{:}\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
var $n, m{:}\mathbb{N}$
rew $add(n, 0) = n$
$add(n, S(m)) = S(add(n, m))$

we can for instance in a fully standard fashion derive that $add(0, n) = n$. We apply induction on the constructors for $\mathbb{N}$.

First, we must show that $add(0, 0) = 0$. This is a trivial instance of the first axiom about *add*. Secondly, assuming $add(0, n) = n$, we must show $add(0, S(n)) = S(n)$. This follows by:

$$add(0, S(n)) = S(add(0, n)) = S(n).$$

In a similar way, induction can be used on any datatype declared with constructors.

4.2 Processes

4.2.1 Conditionals

We list some simple identities between processes containing conditionals. As the proofs of these identities straightforward, only the first one is proven. Note that there are many more of these identities; we have not at all attempted to give an exhaustive list.

Lemma 4.2.

1. $x \lhd b \rhd y = x \lhd b \rhd \delta \lhd \mathbf{0} + y \lhd \neg b \rhd \delta \lhd \mathbf{0}$;

2. $x \lhd b_1 \vee b_2 \rhd \delta \lhd \mathbf{0} = x \lhd b_1 \rhd \delta \lhd \mathbf{0} + x \lhd b_2 \rhd \delta \lhd \mathbf{0}$;
3. $(b = \mathsf{t} \to x = y) \to x \lhd b \rhd z = y \lhd b \rhd z$;
4. $x \lhd b \rhd y \between z = x \between z \lhd b \rhd y \between z$.

Proof. All these identities are proven with induction on booleans. The first identity for $b = \mathsf{t}$ states $x = x + \delta \lhd \mathbf{0}$ which is exactly axiom AT6. For $b = \mathsf{f}$ it says $y = \delta \lhd \mathbf{0} + y$, which using axiom A1 and AT6 is also trivial to prove. □

4.2.2 Summand inclusion

For processes we use the shorthand $x \subseteq y$ for $x + y = y$ and we write $x \supseteq y$ for $y \subseteq x$. This notation is called *summand inclusion*. It is useful to divide the proof of an equality into proving two inclusions, as the following lemma shows.

Lemma 4.3 (*Summand inclusion*)**.**

$$x \subseteq y,\ y \subseteq x \ \to\ x = y$$

Proof. By assumption we know that $x + y = y$ and $y + x = x$. Hence, $x = y + x \overset{\text{A1}}{=} x + y = y$. □

4.2.3 Expansion of the parallel operator

Primary to many verifications is the *expansion* of the parallel operator. This means that the parallel operator is removed in favour of the alternative and sequential operator. Given that we want to have a finite set of axioms we require the auxiliary left and communication merge operator, and the bounded initialisation operator. Below we give a typical example of such an expansion. Note that due to the presence of time these expansions differ slightly from those in [3].

Assume a sort D is declared with at least an element d_0, together with three actions c, r, s of sort D such that r and s communicate to c (i.e. $\gamma(r, s) = c$).

$$\begin{array}{l}
\sum_{d:D} r(d) \parallel s(d_0) \overset{\text{CM1}}{=} \\
\sum_{d:D} r(d) \between s(d_0) + s(d_0) \between \sum_{d:D} r(d) + \sum_{d:D} r(d) \mid s(d_0) \overset{\text{SUM6,SUM7}}{=} \\
\sum_{d:D} (r(d) \between s(d_0)) + s(d_0) \between \sum_{d:D} r(d) + \sum_{d:D} (r(d) \mid s(d_0)) \overset{\text{CM2,CF}}{=} \\
\sum_{d:D} (r(d) \ll s(d_0)) \cdot s(d_0) + (s(d_0) \ll \sum_{d:D} r(d)) \cdot \sum_{d:D} r(d) + \sum_{d:D} c(d) \lhd eq(d, d_0) \rhd \delta \overset{\ll 1, \ll 4, \text{SUM1}}{=} \\
\sum_{d:D} r(d) \cdot s(d_0) + s(d_0) \cdot \sum_{d:D} r(d) + \sum_{d:D} c(d) \lhd eq(d, d_0) \rhd \delta
\end{array}$$

Below it is illustrated how the result of the communication can be further simplified, and how communication between r and s is enforced.

4.2.4 Elimination of a finite sum

We show how the following identity can be proven:

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

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

Lemma 4.4. For all $m{:}\mathbb{N}$ we find (S is the successor function):

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

Proof. Using Lemma 4.3 we can split the proof into two summand inclusions.

$\subseteq$) We first prove the following statement with induction on n:

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

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

$$\begin{array}{l} X0 + \sum_{m:\mathbb{N}} XS(m) \stackrel{\text{SUM3}}{=} \\ X(0) + \sum_{m:\mathbb{N}} XS(m) + XS(n') \supseteq \\ Xn. \end{array}$$

So (2) has been proven without assumption on n (i.e. for all n). Hence, application of SUM11, SUM4 and SUM1 yields:

$$\sum_{n:\mathbb{N}} Xn \subseteq X0 + \sum_{m:\mathbb{N}} XS(m),$$

as had to be shown.

$\supseteq$) Using SUM3 it immediately follows that for all m

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

So, SUM11, SUM4 and SUM1 yield:

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

□

Equation (1) can now easily be proven by:

$$\begin{array}{l} \sum_{n:\mathbb{N}} r(n) \lhd n \leq 2 \rhd \delta \stackrel{\text{Lemma 4.4}}{=} \\ r(0) \lhd 0 \leq 2 \rhd \delta + \sum_{n':\mathbb{N}} r(S(n')) \lhd S(n') \leq 2 \rhd \delta \stackrel{\text{Lemma 4.4}}{=} \\ r(0) + r(S(0)) \lhd S(0) \leq 2 \rhd \delta + \sum_{n'':\mathbb{N}} r(S(S(n''))) \lhd S(S(n'')) \leq 2 \rhd \delta \stackrel{\text{Lemma 4.4}}{=} \\ r(0) + r(1) + r(S(S(0)) \lhd S(S(0)) \leq 2 \rhd \delta + \sum_{n''':\mathbb{N}} r(S(S(S(n'''))) \lhd S(S(S(n'''))) \leq 2 \rhd \delta = \\ r(0) + r(1) + r(2) + \delta \end{array}$$

Note that we use that we can prove that $0 \leq 2 = \mathsf{t}$, $S(0) \leq 2 = \mathsf{t}$, $S(S(0)) \leq 2 = \mathsf{t}$ and $S(S(S(n'''))) \leq 2 = \mathsf{f}$. According to Lemma 4.6.1 below we may omit the last δ.

4.2.5 Abuse of SUM11

One of the most tricky rules of μCRL is SUM11. The universal quantifier expresses that the rule may only be applied when there are no assumptions made on d. Here we show what can go wrong.

First, assume that using the elimination of a finite sum, we have proven $\sum_{b:\mathbf{Bool}} x \lhd b \rhd y = x + y$. This fact is valid, although the proof is slightly tricky. An easy corollary of this fact is that $\sum_{b:\mathbf{Bool}} x \lhd b \rhd y = \sum_{b:\mathbf{Bool}} y \lhd b \rhd x$. We prove below the invalid equation $x = \sum_{b:\mathbf{Bool}} x \lhd b \rhd y$. Note that this equation easily allows us to conclude that $x = \sum_{b:\mathbf{Bool}} x \lhd b \rhd y = \sum_{b:\mathbf{Bool}} y \lhd b \rhd x = y$, collapsing the whole domain of processes.

The erroneous proof of $\sum_{b:\mathbf{Bool}} x \lhd b \rhd y = x$ goes with induction on b. First take $b = \mathsf{t}$. Clearly, $x \lhd b \rhd y = x$, using axiom C1. Hence, using SUM11 and SUM1, $\sum_{b:\mathbf{Bool}} x \lhd b \rhd y = x$. Second, consider the case where $b = \mathsf{f}$. Using C2 we find $y \lhd b \rhd x = x$. Hence, using SUM11 and SUM1 we find $\sum_{b:\mathbf{Bool}} y \lhd b \rhd x = x$. According to the observation above, we may exchange the position of x and y in the left hand side. So, we find the desired $\sum_{b:\mathbf{Bool}} x \lhd b \rhd y = x$. As we have proven the equation for $b = \mathsf{t}$ and $b = \mathsf{f}$, we may conclude that it universally holds.

The problem of course in this example is that when applying SUM11, the premise is only valid for certain b, instead of all b, namely $b = \mathsf{t}$ and $b = \mathsf{f}$ respectively.

Despite this pittfall, we quite often use sequences $\sum \ldots = \sum \ldots = \sum \ldots$, where we transform the process terms at $\ldots$ using, axioms and lemmas. See for instance the proof of Lemma 4.6.2. In such cases we carefully check that the transformation is valid for every variable bound by the sum operator, and we silently apply SUM11.

4.2.6 Sum elimination

An important law is *sum elimination*. It states that the sum over a datatype from which only one element can be selected can be removed. This lemma occurred for the first time in [14]. Note that we assume that we have a function eq available, reflecting equality between terms. We provide two variants. A general one, for use in timed setting, and a specific one for specifications in which time does not play a role. We only provide the proof of the first one.

Lemma 4.5 (*Sum elimination*)**.** Let D be a sort and $eq{:}D \times D \to \mathbf{Bool}$ a function such that for all $d, e{:}D$ it holds that $eq(d, e) = \mathsf{t}$ iff $d = e$. Then

- $\sum_{d:D} Xd \lhd eq(d, e) \rhd \delta \cdot \mathbf{0} = Xe$.
- $\sum_{d:D} c(d) \cdot Xd \lhd eq(d, e) \rhd \delta = c(e) \cdot Xe$.

Proof. According to Lemma 4.3 it suffices to prove summand inclusion in both directions.

$\subseteq$) Using Lemma 4.2.1 above we find:

$$\forall d{:}D.Xe = Xe \lhd eq(d, e) \rhd \delta \cdot \mathbf{0} + Xe \lhd \neg eq(d, e) \rhd \delta \cdot \mathbf{0}.$$

We may put the $\forall d{:}D$ in front of the formula, because we did not make any assumption about d. Using SUM11, SUM4, Lemma 4.2.3 and the assumption that $eq(d, e) = \mathsf{t} \to d = e$, we find:

$$\sum_{d:D} Xe = \sum_{d:D} Xd \lhd eq(d, e) \rhd \delta \cdot \mathbf{0} + \sum_{d:D} Xe \lhd \neg eq(d, e) \rhd \delta \cdot \mathbf{0}.$$

Using SUM1 and the summand inclusion notation we obtain:

$$\sum_{d:D} Xd \lhd eq(d, e) \rhd \delta \cdot \mathbf{0} \subseteq Xe.$$

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

$$\sum_{d:D} Xd \lhd eq(d, e) \rhd \delta \cdot \mathbf{0} \supseteq Xe \lhd eq(e, e) \rhd \delta \cdot \mathbf{0} = Xe.$$

□

4.2.7 Data transfer

We show how we can verify that data sent by one process indeed arrives at the recipient, who can then use it further. Assume we are given a data domain D with at least an element d_0 and an equality function eq. We define the following sender process S (the variable x denotes some arbitrary continuation of the process):

proc $S = s(d_0)\cdot x$

A receiving process can be defined as follows (Yd is a continuating process in which the received value d is used):

proc $R = \sum_{d:D} r(d)\cdot Yd$

We let the send s and receive r actions synchronize by declaring:

comm $s|r = c$

And we should be able to show that:

$$\partial_{\{r,s\}}(S \parallel R) = c(d_0)\cdot\partial_{\{r,s\}}(x \parallel Yd_0)$$

where the encapsulation operator $\partial_{\{r,s\}}$ enforces the r and s action to synchronize.

The verification uses the following steps, where the expansion step is similar to the one given in section 4.2.3:

$$\begin{array}{l}\partial_{\{r,s\}}(S \parallel R) \overset{\text{Expansion}}{=} \\ \partial_{\{r,s\}}(s(d_0)\cdot(x \parallel R) + \sum_{d:D} r(d)\cdot(S \parallel Yd) + \sum_{d:D} c(d)\cdot(x \parallel Yd) \lhd eq(d,d_0) \rhd \delta) = \\ \sum_{d:D} c(d)\partial_{\{r,s\}}((x \parallel Yd) \lhd eq(d,d_0) \rhd \delta) \overset{\text{Sum elimination}}{=} \\ c(d_0)\cdot\partial_{\{r,s\}}(x \parallel Yd_0)\end{array}$$

4.2.8 Process identities involving time

The first identity below is an instance of the law $x + \delta = x$ in process algebra that is invalid in timed μCRL. It illustrates however that $x + \delta = x$ is still valid, if x does not refer to time. The second identity is used in the next example.

Lemma 4.6.

1. $c + \delta = c$;
2. $\sum_{t:\mathbf{Time}} \delta \vartriangleleft min(t,t_0) \lhd t \leq t_1 \rhd \delta \vartriangleleft \mathbf{0} = \delta \vartriangleleft min(t_0,t_1)$.

Proof.

1. $c + \delta \overset{\text{ATA1,ATB3}}{=} \sum_{t:\mathbf{Time}}(c \vartriangleleft t + \delta \vartriangleleft t) \overset{\text{ATA2}}{=} \sum_{t:\mathbf{Time}} c \vartriangleleft t \overset{\text{ATA1}}{=} c$.
2. The proof of the second item is somewhat more involved:

$$\begin{array}{l}\sum_{t:\mathbf{Time}} \delta \vartriangleleft min(t,t_0) \lhd t \leq t_1 \rhd \delta \vartriangleleft \mathbf{0} \overset{\text{Time3,4.2.2}}{=} \\ \sum_{t:\mathbf{Time}} \delta \vartriangleleft min(t,t_0) \lhd t \leq t_1 \wedge t \leq t_0 \rhd \delta \vartriangleleft \mathbf{0} + \\ \sum_{t:\mathbf{Time}} \delta \vartriangleleft min(t,t_0) \lhd t \leq t_1 \wedge t_0 \leq t \rhd \delta \vartriangleleft \mathbf{0} \overset{\text{4.1.2,4.2.3,Time6}}{=} \\ \sum_{t:\mathbf{Time}} \delta \vartriangleleft t \lhd t \leq min(t_0,t_1) \rhd \delta \vartriangleleft \mathbf{0} + \\ \sum_{t:\mathbf{Time}} \delta \vartriangleleft t_0 \lhd t \leq t_1 \wedge t_0 \leq t \rhd \delta \vartriangleleft \mathbf{0} \overset{*}{=} \\ \delta \vartriangleleft min(t_0,t_1) + \delta \vartriangleleft t_0 \lhd t_0 \leq t_1 \rhd \delta \vartriangleleft \mathbf{0} \overset{\text{4.1.2,4.2.3}}{=} \\ \delta \vartriangleleft min(t_0,t_1) + \delta \vartriangleleft min(t_0,t_1) \lhd t_0 \leq t_1 \rhd \delta \vartriangleleft \mathbf{0} \overset{\text{A3}}{=} \\ \delta \vartriangleleft min(t_0,t_1)\end{array}$$

At * we use the identities $\sum_{t:\mathbf{Time}} \delta \cdot t_0 \lhd t \leq t_1 \wedge t_0 \leq t \rhd \delta \cdot \mathbf{0} = \delta \cdot t_0 \lhd t_0 \leq t_1 \rhd \delta \cdot \mathbf{0}$ and $\sum_{t:\mathbf{Time}} \delta \cdot t \lhd \leq t' \rhd \delta \cdot \mathbf{0} = \delta \cdot \mathbf{0}$ which are straightforwardly provable using among others summand inclusion, ATA2, SUM3, SUM4 and SUM11.

□

4.2.9 Processes with time constraints

In this section we show what happens if processes with time constraints must communicate. In order to show what can happen we describe a process that sends via action name s at time t_0 to a process that receives via action name r before time t_1. We prove the combination of these processes equal to $c \cdot t_0 \lhd t_0 \leq t_1 \rhd \delta \cdot t_1$. This says that if t_0 comes before t_1 the processes communicate at time t_0. If however t_0 comes after t_1, then if time would pass t_1, the constraints of the receiving party would be violated. Therefore, we find a deadlock at time t_1.

We describe the communicating processes as follows:

$$\partial_{\{r,s\}}(s \cdot t_0 \parallel \sum_{t:\mathbf{Time}} r \cdot t \lhd t \leq t_1 \rhd \delta \cdot \mathbf{0}).$$

By applying the law CM1 we reduce this term to

$$\begin{array}{l} \partial_{\{r,s\}}(\sum_{t:\mathbf{Time}} r \cdot t \lhd t \leq t_1 \rhd \delta \cdot \mathbf{0} \between s \cdot t_0) + \\ \partial_{\{r,s\}}(s \cdot t_0 \between \sum_{t:\mathbf{Time}} r \cdot t \lhd t \leq t_1 \rhd \delta \cdot \mathbf{0}) + \\ \partial_{\{r,s\}}(s \cdot t_0 \mid \sum_{t:\mathbf{Time}} r \cdot t \lhd t \leq t_1 \rhd \delta \cdot \mathbf{0}) \end{array} \tag{3}$$

Now we prove the first and last summand of (3) in turn. The proof of the second summand goes in the same way as the proof of the first one and is therefore not given. The first summand of (3) can be shown equal to $\delta \cdot min(t_0, t_1)$ as follows:

$$\begin{array}{ll} \partial_{\{r,s\}}(\sum_{t:\mathbf{Time}} r \cdot t \lhd t \leq t_1 \rhd \delta \cdot \mathbf{0} \between s \cdot t_0) & \overset{\text{SUM6,SUM8,4.2.4}}{=} \\ \sum_{t:\mathbf{Time}}(\partial_{\{r,s\}}(r \cdot t \between s \cdot t_0) \lhd t \leq t_1 \rhd \partial_{\{r,s\}}(\delta \cdot \mathbf{0} \between s \cdot t_0)) & \overset{\text{CM2}}{=} \\ \sum_{t:\mathbf{Time}}(\partial_{\{r,s\}}((r \cdot t \ll s \cdot t_0) \cdot s \cdot t_0) \lhd t \leq t_1 \rhd \partial_{\{r,s\}}((\delta \cdot \mathbf{0} \ll s \cdot t_0) \cdot s \cdot t_0)) & \overset{\text{D1,D4,}\ll 5}{=} \\ \sum_{t:\mathbf{Time}}(\partial_{\{r,s\}}(\sum_{t':\mathbf{Time}}(r \cdot t \ll s) \cdot t' \lhd t' \leq t_0 \rhd \delta \cdot \mathbf{0}) \cdot s \cdot t_0 \lhd t \leq t_1 \rhd & \\ \qquad \partial_{\{r,s\}}(\sum_{t':\mathbf{Time}}(\delta \cdot \mathbf{0} \ll s) \cdot t' \lhd t' \leq t_0 \rhd \delta \cdot \mathbf{0}) \cdot s \cdot t_0) & \overset{\ll 1,\text{ATB1}}{=} \\ \sum_{t:\mathbf{Time}}(\partial_{\{r,s\}}(\sum_{t':\mathbf{Time}}(r \cdot t \cdot t' \lhd t' \leq t_0 \rhd \delta \cdot \mathbf{0}) \cdot s \cdot t_0 \lhd t \leq t_1 \rhd & \\ \qquad \partial_{\{r,s\}}(\sum_{t':\mathbf{Time}}(\delta \cdot min(\mathbf{0}, t')) \lhd t' \leq t_0 \rhd \delta \cdot \mathbf{0}) \cdot s \cdot t_0) = & \\ \sum_{t:\mathbf{Time}}((\sum_{t':\mathbf{Time}} \delta \cdot min(t, t') \lhd t' \leq t_0 \rhd \delta \cdot \mathbf{0}) \cdot \delta \cdot t_0 \lhd t \leq t_1 \rhd \delta \cdot \mathbf{0}) & \overset{4.6.2}{=} \\ \sum_{t:\mathbf{Time}} \delta \cdot min(t, t_0) \lhd t \leq t_1 \rhd \delta \cdot \mathbf{0} & \overset{4.6.2}{=} \\ \delta \cdot min(t_0, t_1). & \end{array}$$

Similarily, it can be shown that the second summand of (3) equals $\delta \cdot min(t_0, t_1)$.

We now prove the third summand of (3) equal to $c\langle t_0 \rangle \lhd t_0 \leq t_1 \rhd \delta\langle t_1 \rangle$.

$$
\begin{aligned}
&\partial_{\{r,s\}}(s\langle t_0 | \textstyle\sum_{t:\mathbf{Time}} r\langle t \rangle \lhd t \leq t_1 \rhd \delta\langle \mathbf{0} \rangle) \overset{\text{SUM7}',\text{SUM8}}{=} \\
&\textstyle\sum_{t:\mathbf{Time}} (\partial_{\{r,s\}}(s\langle t_0 | r\langle t \rangle) \lhd t \leq t_1 \rhd \partial_{\{r,s\}}(s\langle t_0 | \delta\langle \mathbf{0} \rangle)) \overset{\text{ATB}*,\text{CD2},\text{CF},\text{D1}}{=} \\
&\textstyle\sum_{t:\mathbf{Time}} ((c\langle t_0 \rangle \lhd eq(t_0,t) \rhd \delta\langle min(t_0,t) \rangle) \lhd t \leq t_1 \rhd \delta\langle min(t_0,\mathbf{0}) \rangle) = \\
&\textstyle\sum_{t:\mathbf{Time}} c\langle t_0 \rangle \lhd eq(t_0,t) \wedge t \leq t_1 \rhd \delta\langle \mathbf{0} + \\
&\textstyle\sum_{t:\mathbf{Time}} \delta\langle min(t_0,t) \rangle \lhd \neg eq(t_0,t) \wedge t \leq t_1 \rhd \delta\langle \mathbf{0} \overset{\text{Sum elimination},4.2.1,\text{ATA2}}{=} \\
&(c\langle t_0 \rangle + \delta\langle min(t_0,t_1) \rangle) \lhd t_0 \leq t_1 \rhd \delta\langle \mathbf{0} + \\
&\textstyle\sum_{t:\mathbf{Time}} \delta\langle min(t_0,t) \rangle \lhd \neg eq(t_0,t) \wedge t \leq t_1 \wedge t_0 \leq t_1 \rhd \delta\langle \mathbf{0} + \\
&\textstyle\sum_{t:\mathbf{Time}} \delta\langle min(t_0,t) \rangle \lhd \neg eq(t_0,t) \wedge t \leq t_1 \wedge t_1 < t_0 \rhd \delta\langle \mathbf{0} \overset{4.6.2}{=} \\
&c\langle t_0 \rangle \lhd t_0 \leq t_1 \rhd \delta\langle \mathbf{0} + \\
&\textstyle\sum_{t:\mathbf{Time}} \delta\langle min(t_0,t) \rangle \lhd \neg eq(t_0,t) \wedge t \leq t_1 \wedge t_0 \leq t_1 \rhd \delta\langle \mathbf{0} + \\
&\textstyle\sum_{t:\mathbf{Time}} \delta\langle min(t_0,t) \rangle \lhd t \leq t_1 \wedge t_0 \leq t_1 \rhd \delta\langle \mathbf{0} + \\
&\textstyle\sum_{t:\mathbf{Time}} \delta\langle min(t_0,t) \rangle \lhd t \leq t_1 \wedge t_1 < t_0 \rhd \delta\langle \mathbf{0} = \\
&c\langle t_0 \rangle \lhd t_0 \leq t_1 \rhd \delta\langle \mathbf{0} + \\
&\textstyle\sum_{t:\mathbf{Time}} \delta\langle min(t_0,t) \rangle \lhd t \leq t_1 \wedge t_0 \leq t_1 \rhd \delta\langle \mathbf{0} + \\
&\textstyle\sum_{t:\mathbf{Time}} \delta\langle min(t_0,t) \rangle \lhd t \leq t_1 \wedge t_1 < t_0 \rhd \delta\langle \mathbf{0} \overset{4.2.1}{=} \\
&c\langle t_0 \rangle \lhd t_0 \leq t_1 \rhd \delta\langle \mathbf{0} + \\
&\textstyle\sum_{t:\mathbf{Time}} \delta\langle min(t_0,t) \rangle \lhd t \leq t_1 \rhd \delta\langle \mathbf{0} = \\
&c\langle t_0 \rangle \lhd t_0 \leq t_1 \rhd \delta\langle \mathbf{0} + \delta\langle min(t_0,t_1) \rangle = \\
&c\langle t_0 \rangle \lhd t_0 \leq t_1 \rhd \delta\langle t_1 \rangle.
\end{aligned}
$$

Using the results above we can prove (3) equal to

$$
\begin{aligned}
&\delta\langle min(t_0,t_1) \rangle + \delta\langle min(t_0,t_1) \rangle + c\langle t_0 \rangle \lhd t_0 \leq t_1 \rhd \delta\langle t_1 \rangle = \\
&(c\langle t_0 \rangle + \delta\langle min(t_0,t_1) \rangle) \lhd t_0 \leq t_1 \rhd (\delta\langle t_1 \rangle + \delta\langle min(t_0,t_1) \rangle) = \\
&c\langle t_0 \rangle \lhd t_0 \leq t_1 \rhd \delta\langle t_1
\end{aligned}
$$

A The formal syntax of timed μCRL

We define a syntax of timed μCRL specifications. Contrary to the syntax used in the main text, this syntax is completely fixed; the syntax is meant for computer processing and intended for specifications of real systems, where ambiguities, including ambiguities in the syntax cannot be tolerated. The syntax uses standard (roman) symbols only, extended with some common punctuation symbols.

The syntax is defined in the Syntax Definition Formalism (SDF) [22]. According to the convention in SDF we write syntactical categories with a capital, and keywords with small letters. The first LAYOUT rule says that spaces (' '), tabs (`\t`) and newlines (`\n`) may be used to generate some attractive layout and are not part of the μCRL specification itself. The second LAYOUT rule says that lines starting with a `%`-sign followed by zero or more non-newline characters (`~[\n]*`) followed by a newline (`\n`) must be taken as comments and are therefore also not a part of the μCRL syntax.

A `Name` is an arbitrary string over `a-z`, `A-Z`, `0-9` and the special characters `^_'-`. By a default SDF convention keywords cannot be a `Name` at the same time. In the context free syntax most items are self-explanatory. The symbol `+` stands for one or more and `*` for zero or more occurrences. For instance `{ Name ","}+` is a list of one or more `Name` separated by commas, without a trailing comma.

The phrase `{right}` means that an operator is right-associative and `{assoc}` means that an operator is associative. The phrase `{bracket}` says that the defined construct is not an operator, but just a way to disambiguate the construction of a syntax tree.

The priorities say that the operator '`@`' has highest and `+` has lowest priority when parsing process terms with ambiguous bracketing.

```
exports
 sorts Name
       Name-list
       Domain
       Sort-specification
       Function-specification
       Function-declaration
       Equation-specification
       Variable-declaration
       Variables
       Data-term
       Equation-section
       Single-equation
       Process-term
       Renaming
       Variable
       Process-specification
       Process-declaration
       Action-specification
       Action-declaration
       Communication-specification
       Communication-declaration
       Initial-declaration
       Specification
 lexical syntax
       [ \t\n]                                        -> LAYOUT
       "%" ~[\n]* "\n"                                -> LAYOUT
       [a-zA-Z0-9^_'\-]+                              -> Name
 context-free syntax
       { Name ","}+                                   -> Name-list
       { Name "#"}+                                   -> Domain
       "sort" Name+                                   -> Sort-specification
       "func" Function-declaration+                   -> Function-specification
       "map" Function-declaration+                    -> Function-specification
       Name-list ":" Domain "->" Name                 -> Function-declaration
       Name-list ":" "->" Name                        -> Function-declaration

       Variable-declaration Equation-section          -> Equation-specification
       "var" Variables+                               -> Variable-declaration
                                                      -> Variable-declaration
       Name-list ":" Name                             -> Variables
       Name                                           -> Data-term
       Name "(" { Data-term "," }+ ")"                -> Data-term
       "rew" Single-equation+                         -> Equation-section
       Data-term "=" Data-term                        -> Single-equation

       Process-term "+" Process-term                  -> Process-term {right}
       Process-term "||" Process-term                 -> Process-term {right}
       Process-term "||_" Process-term                -> Process-term
       Process-term "|"   Process-term                -> Process-term {right}
       Process-term "<|" Data-term "|>" Process-term -> Process-term
       Process-term "." Process-term                  -> Process-term {right}
       Process-term "@" Data-term                     -> Process-term
       Process-term "<<" Process-term                 -> Process-term {left}
       "delta"                                        -> Process-term
```

```
      "tau"                                          -> Process-term
      "encap" "(" "{" Name-list "}" ","
                      Process-term ")"               -> Process-term
      "hide" "(" "{" Name-list "}" ","
                      Process-term ")"               -> Process-term
      "rename" "(" "{" { Renaming "," }+ "}" ","
                           Process-term ")"          -> Process-term
      "sum" "(" Variable "," Process-term ")"        -> Process-term
      Name "(" { Data-term "," }+ ")"                -> Process-term
      Name                                           -> Process-term
      "(" Process-term ")"                           -> Process-term {bracket}
      Name "->" Name                                 -> Renaming
      Name ":" Name                                  -> Variable
      "proc" Process-declaration+                    -> Process-specification
      Name "(" { Variable "," }+ ")" "="
                           Process-term              -> Process-declaration
      Name "=" Process-term                          -> Process-declaration
      "act" Action-declaration+                      -> Action-specification
      Name-list ":" Domain                           -> Action-declaration
      Name                                           -> Action-declaration
      "comm" Communication-declaration+              -> Communication-specification
      Name "|" Name "=" Name                         -> Communication-declaration
      "init" Process-term                            -> Initial-declaration

      Sort-specification                             -> Specification
      Function-specification                         -> Specification
      Equation-specification                         -> Specification
      Action-specification                           -> Specification
      Communication-specification                    -> Specification
      Process-specification                          -> Specification
      Initial-declaration                            -> Specification
      Specification Specification                    -> Specification  {assoc}

priorities
      Process-term "@" Data-term -> Process-term >
      Process-term "." Process-term -> Process-term >
      Process-term "<<" Process-term -> Process-term >
      { Process-term "||" Process-term -> Process-term,
        Process-term "|" Process-term -> Process-term,
        Process-term "||_" Process-term -> Process-term } >
      Process-term "<|" Data-term "|>" Process-term -> Process-term >
      Process-term "+" Process-term -> Process-term
```

In order to relate the symbols that we use in the text, with their plain text representation in the syntax above, we provide a translation table (see Table 7). We followed some conventions regarding naming that have been set by PSF [31]. Above the double line in this table keywords and operators that occur explicitly in the syntax above. Below the double line some predefined names occur.

B Static semantics and wellformedness

In this section it is defined when a specification is correctly defined. We use the syntactical categories from the previous section (in teletype font) to refer to items in a specification. If we denote a concrete part of a specification, we prefer using the latex symbols, to increase readability. The definitions below

LaTeX	plain text	
sort	`sort`	
func	`func`	
map	`map`	
var	`var`	
rew	`rew`	
act	`act`	
comm	`comm`	
init	`init`	
$+$	`+`	
$\parallel$	`||`	
$\between$	`||_`	
$\mid$	`|`	
$\triangleleft$	`<|`	
$\triangleright$	`|>`	
$\cdot$	`.`	$\cdot$ is sometimes omitted
`	`@`	
$\ll$	`<<`	
δ	`delta`	
τ	`tau`	in case τ is a constant
$\partial_{\{\ldots\}}\cdots$	`encap({...},...)`	
$\tau_{\{\ldots\}}\cdots$	`hide({...},...)`	
$\rho_{\{\ldots\}}\cdots$	`rename({...},...)`	
$\sum_{d:D}\cdots$	`sum(d:D,...)`	

Bool	`Bool`	
t	`T`	
f	`F`	
Time	`Time`	
0	`time0`	This is the symbol for time zero
$\leq$	`le`	This is a binary function from **Time** $\times$ **Time** to **Bool**

Table 7: Translation table

are an adapted copy from those in [17].

In essence the static semantics says that functions and terms are well typed, and some sorts and functions are present in the specification. The validity of all static semantic requirements can efficiently be decided for any specification.

A specification is well formed, if it satisfies the static semantic requirements, the symmetric closure of the communication function is associative, there are no empty sorts and the sorts **Bool** and **Time** are appropriately defined. We only give an operational semantics to well formed specifications.

B.1 The signature of a specification

Definition B.1. The signature $Sig(E)$ of a `Specification` E consists of a seven-tuple

$$(Sort, Fun, Map, Act, Comm, Proc, Init)$$

where each component is a set containing all elements of a main syntactical category declared in E. The signature $Sig(E)$ of E is inductively defined as follows:

- If $E \equiv \textbf{sort}\ n_1 \cdots n_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\{n_1, \ldots, n_m\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.
- If $E \equiv \textbf{func}\ fd_1 \cdots fd_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, Fun, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$, where

$$\begin{array}{lll} Fun & \stackrel{\text{def}}{=} & \{n_{ij}\colon \to S_i \mid fd_i \equiv n_{i1}, \ldots, n_{il_i}\colon \to S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\} \\ & \cup & \{n_{ij}\colon S_{i1} \times \cdots \times S_{ik_i} \to S_i \mid \\ & & \quad fd_i \equiv n_{i1}, \ldots, n_{il_i}\colon S_{i1} \times \cdots \times S_{ik_i} \to S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\}. \end{array}$$

- If $E \equiv \textbf{map}\ md_1 \cdots md_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, Map, \emptyset, \emptyset, \emptyset, \emptyset)$, where

$$\begin{array}{lll} Map & \stackrel{\text{def}}{=} & \{n_{ij}\colon \to S_i \mid md_i \equiv n_{i1}, \ldots, n_{il_i}\colon \to S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\} \\ & \cup & \{n_{ij}\colon S_{i1} \times \cdots \times S_{ik_i} \to S_i \mid \\ & & \quad md_i \equiv n_{i1}, \ldots, n_{il_i}\colon S_{i1} \times \cdots \times S_{ik_i} \to S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\}. \end{array}$$

- If E is a `Equation-specification`, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.
- If $E \equiv \textbf{act}\ ad_1 \cdots ad_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, Act, \emptyset, \emptyset, \emptyset)$, where

$$\begin{array}{lll} Act & \stackrel{\text{def}}{=} & \{n_i \mid ad_i \equiv n_i, 1 \leq i \leq m\} \\ & \cup & \{n_{ij}\colon S_{i1} \times \cdots \times S_{ik_i} \mid \\ & & \quad ad_i \equiv n_{i1}, \ldots, n_{il_i}\colon S_{i1} \times \cdots \times S_{ik_i}, 1 \leq i \leq m, 1 \leq j \leq l_i\}. \end{array}$$

- If $E \equiv \textbf{comm}\ cd_1 \cdots cd_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \{cd_i \mid 1 \leq i \leq m\}, \emptyset, \emptyset)$.
- If $E \equiv \textbf{proc}\ pd_1 \cdots pd_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{pd_i \mid 1 \leq i \leq m\}, \emptyset)$.
- If $E \equiv \textbf{init}\ pe$ then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{pe\})$.
- If $E \equiv E_1\ E_2$ with $Sig(E_i) = (Sort_i, Fun_i, Map_i, Act_i, Comm_i, Proc_i, Init_i)$ for $i = 1, 2$, then

$$\begin{array}{l} Sig(E) \stackrel{\text{def}}{=} (Sort_1 \cup Sort_2, Fun_1 \cup Fun_2, Map_1 \cup Map_2, \\ \qquad\qquad Act_1 \cup Act_2, Comm_1 \cup Comm_2, Proc_1 \cup Proc_2, Init_1 \cup Init_2). \end{array}$$

Definition B.2. Let $Sig = (Sort, Fun, Map, Act, Comm, Proc, Init)$ be a signature. We write

$Sig.Sort$ for $Sort$, $Sig.Fun$ for Fun, $Sig.Map$ for Map, $Sig.Act$ for Act,
$Sig.Comm$ for $Comm$, $Sig.Proc$ for $Proc$, $Sig.Init$ for $Init$.

B.2 Variables

Variables play an important role in specifications. The next definition says given a specification E, which elements from `Name` can play the role of a variable without confusion with defined constants. Moreover, variables must have an unambiguous and declared sort.

Definition B.3. Let Sig be a signature. A set $\mathcal{V}$ containing pairs $\langle x{:}S\rangle$ with x and S from `Name`, is called a *set of variables* over Sig iff for each $\langle x{:}S\rangle \in \mathcal{V}$:

- for each `Name` S' and `Process-term` p it holds that $x{:} \to S' \notin Sig.Fun \cup Sig.Map$, $x \notin Sig.Act$ and $x = p \notin Sig.Proc$,
- $S \in Sig.Sort$,
- for each `Name` S' such that $S' \not\equiv S$ it holds that $\langle x{:}S'\rangle \notin \mathcal{V}$.

Definition B.4. Let vd be a `Variable-declaration`. The function $Vars$ is defined by:

$$Vars(vd) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } vd \text{ is empty,} \\ \{\langle x_{ij}{:}S_i\rangle \mid 1 \le i \le m, \\ \qquad 1 \le j \le l_i\} & \text{if } vd \equiv \textbf{var } x_{11},\ldots,x_{1l_1}{:}S_1 \ \ldots\ x_{m1},\ldots,x_{ml_m}{:}S_m. \end{cases}$$

In the following definitions we give functions yielding the sort of and the variables in a `Data-term` t.

Definition B.5. Let t be a `data-term` and Sig a signature. Let $\mathcal{V}$ be a set of variables over Sig. We define:

$$sort_{Sig,\mathcal{V}}(t) \stackrel{\text{def}}{=} \begin{cases} S & \text{if } t \equiv x \text{ and } \langle x{:}S\rangle \in \mathcal{V}, \\ S & \text{if } t \equiv n,\ n{:} \to S \in Sig.Fun \cup Sig.Map \\ & \quad \text{and for no } S' \not\equiv S\ n{:} \to S' \in Sig.Fun \cup Sig.Map, \\ S & \text{if } t \equiv n(t_1,\ldots,t_m), \\ & \quad n{:}sort_{Sig,\mathcal{V}}(t_1) \times \cdots \times sort_{Sig,\mathcal{V}}(t_m) \to S \in Sig.Fun \cup Sig.Map \\ & \quad \text{and for no } S' \not\equiv S\ n{:}sort_{Sig,\mathcal{V}}(t_1) \times \cdots \times sort_{Sig,\mathcal{V}}(t_m) \to \\ & \quad S' \in Sig.Fun \cup Sig.Map, \\ \bot & \text{otherwise.} \end{cases}$$

If a variable or a function is not or inappropriately declared no answer can be obtained. In this case $\bot$ results.

Definition B.6. Let Sig be a signature, $\mathcal{V}$ a set of variables over Sig and let t be a `Data-term`.

$$Var_{Sig,\mathcal{V}}(t) \stackrel{\text{def}}{=} \begin{cases} \{\langle x{:}S\rangle\} & \text{if } t \equiv x \text{ and } \langle x{:}S\rangle \in \mathcal{V}, \\ \emptyset & \text{if } t \equiv n \text{ and } n{:} \to S \in Sig.Fun \cup Sig.Map, \\ \bigcup_{1\le i\le m} Var_{Sig,\mathcal{V}}(t_i) & \text{if } t \equiv n(t_1,\ldots,t_m), \\ \{\bot\} & \text{otherwise.} \end{cases}$$

We call a `Data-term` t *closed* wrt. a signature Sig and a set of variables $\mathcal{V}$ iff $Var_{Sig,\mathcal{V}}(t) = \emptyset$. Note that $Var_{Sig,\mathcal{V}}(t) \subseteq \mathcal{V} \cup \{\bot\}$ for any `data-term` t. If $\bot \in Var_{Sig,\mathcal{V}}(t)$, then due to some missing or inappropriate declaration it can not be determined what the variables of t are on basis of Sig and $\mathcal{V}$.

B.3 Static semantics

A `Specification` must be internally consistent. This means that all objects that are used must be declared exactly once and are used such that the sorts are correct. It also means that action, process, constant and variable names cannot be confused. Furthermore, it means that communications are specified in a functional way and that it is guaranteed that the terms used in an equation are well-typed. Because all these properties can be statically decided, a specification that is internally consistent is called SSC (*Statically Semantically Correct*). All next definitions culminate in Definition B.13.

Definition B.7 (`Data-term`). Let *Sig* be a signature, and let $\mathcal{V}$ be a set of variables over *Sig*. A `Data-term` t is called SSC wrt. *Sig* and $\mathcal{V}$ iff one of the following holds

- $t \equiv n$ with n a `Name` and $\langle n{:}S\rangle \in \mathcal{V}$ for some S, or $n{:} \to sort_{Sig,\mathcal{V}}(n) \in Sig.Fun \cup Sig.Map$.
- $t \equiv n(t_1, \ldots, t_m)$ $(m \geq 1)$ and $n{:}sort_{Sig,\mathcal{V}}(t_1) \times \cdots \times sort_{Sig,\mathcal{V}}(t_m) \to sort_{Sig,\mathcal{V}}(n(t_1, \ldots, t_m)) \in Sig.Fun \cup Sig.Map$ and all t_i $(1 \leq i \leq m)$ are SSC wrt. *Sig* and $\mathcal{V}$.

Definition B.8 (`Equation-section`). Let *Sig* be a signature and $\mathcal{V}$ be a set of variables over *Sig*. An `Equation-section` **rew** $rw_1 \cdots rw_m$ with $m \geq 1$ is SSC wrt. *Sig* and $\mathcal{V}$ iff for all $1 \leq i \leq m$ if $rw_i \equiv t_1 = t_2$, both t_1 and t_2 are SSC wrt. *Sig* and $\mathcal{V}$ and $sort_{Sig,\mathcal{V}}(t_1) = sort_{Sig,\mathcal{V}}(t_2)$.

Definition B.9 (`Variable-declaration`). A `Variable-declaration` *vd* is SSC wrt. a signature *Sig* iff one of the following holds.

- *vd* is empty.
- $vd \equiv$ **var** $n_{11}, \ldots, n_{1k_1}{:}S_1$
  $\vdots$
  $n_{m1}, \ldots, n_{mk_m}{:}S_m$

  with $m \geq 1$, $k_i \geq 1$ for $1 \leq i \leq m$ and
  - $n_{ij} \not\equiv n_{i'j'}$ whenever $i \neq i'$ or $j \neq j'$ for $1 \leq i \leq m$, $1 \leq i' \leq m$, $1 \leq j \leq k_i$ and $1 \leq j' \leq k_{i'}$,
  - the set $Vars($**var** $n_{11}, \ldots, n_{1k_1} : S_1 \cdots n_{m1}, \ldots, n_{mk_m} : S_m)$ is a set of variables over *Sig*.

Definition B.10 (`Process-term`). Let *Sig* be a signature and $\mathcal{V}$ be a set of variables over *Sig*. We say that a `Process-term` p is *SSC wrt. to Sig and E* iff one of the following hold:

- $p \equiv p_1 + p_2$, $p \equiv p_1 \parallel p_2$, $p \equiv p_1 \between p_2$, $p \equiv p_1 \mid p_2$, $p \equiv p_1 \cdot p_2$ or $p \equiv p_1 \ll p_2$ and
  - p_1 is SSC wrt. *Sig* and $\mathcal{V}$,
  - p_2 is SSC wrt. *Sig* and $\mathcal{V}$.
- $p \equiv p_1 \lhd t \rhd p_2$ and
  - p_1 is SSC wrt. *Sig* and $\mathcal{V}$,
  - p_2 is SSC wrt. *Sig* and $\mathcal{V}$,
  - t is SSC wrt. *Sig* and $\mathcal{V}$ and $sort_{Sig,\mathcal{V}}(t) =$ **Bool**.
- $p \equiv p_1 \text{‘} t$ and
  - p_1 is SSC wrt. *Sig* and $\mathcal{V}$
  - t is SSC wrt. *Sig* and $\mathcal{V}$ and $sort_{Sig,\mathcal{V}}(t) =$ **Time**.
- $p \equiv \delta$ or $p \equiv \tau$.
- $p \equiv \partial_{\{n_1,\ldots,n_m\}} p_1$ or $p \equiv \tau_{\{n_1,\ldots,n_m\}} p_1$ with $m \geq 1$ and
  - for all $1 \leq i < j \leq m$ $n_i \not\equiv n_j$,
  - for $1 \leq i \leq m$ either $n_i \in Sig.Act$ or $n_i{:}S_1 \times \cdots \times S_k \in Sig.Act$ for some $k \geq 1$ and `Names` $S_1, \ldots, S_k$,
  - p_1 is SSC wrt. *Sig* and $\mathcal{V}$.
- $p \equiv \rho_{\{n_1 \to n'_1, \ldots, n_m \to n'_m\}} p_1$ and

  - for $1 \leq i \leq m$ either $n_i \in \mathit{Sig.Act}$ or $n_i{:}S_1 \times \cdots \times S_k \in \mathit{Sig.Act}$ for some $k \geq 1$ and `Names` $S_1, \ldots, S_k$,
  - for each $1 \leq i < j \leq m$ it holds that $n_i \not\equiv n_j$,
  - for $1 \leq i \leq m$, $k \geq 1$ and `Names` $S_1, .., S_k$ it holds that if $n_i{:}S_1 \times \cdots \times S_k \in \mathit{Sig.Act}$, then also $n_i'{:}S_1 \times \cdots \times S_k \in \mathit{Sig.Act}$,
  - for $1 \leq i \leq m$ it holds that if $n_i \in \mathit{Sig.Act}$, then also $n_i' \in \mathit{Sig.Act}$,
  - p_1 is SSC wrt. Sig and $\mathcal{V}$.

- $p \equiv \Sigma_{x:S} p_1$ and iff
  - $(\mathcal{V} \backslash \{\langle x{:}S' \rangle \mid S' \text{ is a } \texttt{Name}\}) \cup \{\langle x{:}S \rangle\}$ is a set of variables over Sig,
  - p_1 is SSC wrt. Sig and $(\mathcal{V} \backslash \{\langle x{:}S' \rangle \mid S' \text{ is a } \texttt{Name}\}) \cup \{\langle x{:}S \rangle\}$.
- $p \equiv n$ and $n = p' \in \mathit{Sig.Proc}$ for some `Process-term` p', or $n \in \mathit{Sig.Act}$.
- $p \equiv n(t_1, \ldots, t_m)$ with $m \geq 1$ and
  - $n(x_1{:}\mathit{sort}_{\mathit{Sig},\mathcal{V}}(t_1), \ldots, x_m{:}\mathit{sort}_{\mathit{Sig},\mathcal{V}}(t_m)) = p' \in \mathit{Sig.Proc}$ for `Names` $x_1, \ldots, x_m$ and `Process-term` p', or $n{:}\mathit{sort}_{\mathit{Sig},\mathcal{V}}(t_1) \times \cdots \times \mathit{sort}_{\mathit{Sig},\mathcal{V}}(t_m) \in \mathit{Sig.Act}$,
  - for $1 \leq i \leq m$ the `Data-term` t_i is SSC wrt. Sig and $\mathcal{V}$.

Definition B.11 (`Action-declaration`). Let Sig be a signature. An `Action-declaration` ad is SSC wrt. Sig iff one of the following hold:

- $\mathit{ad} \equiv n$ and for each `Name` S' it holds that $n{:} \rightarrow S' \notin \mathit{Sig.Fun} \cup \mathit{Sig.Map}$.
- An `Action-declaration` $n_1, \ldots, n_m{:}S_1 \times \cdots \times S_k$ with $k, m \geq 1$ is SSC wrt. Sig iff
  - for all $1 \leq i < j \leq m$ it holds that $n_i \not\equiv n_j$,
  - for all $1 \leq i \leq k$ it holds that $S_i \in \mathit{Sig.Sort}$,
  - for all $1 \leq i \leq m$ and for each `Name` S' it holds that $n_i{:}S_1 \times \cdots \times S_k \rightarrow S' \notin \mathit{Sig.Fun} \cup \mathit{Sig.Map}$.

Definition B.12 (`Specification`). Let Sig be a signature and $\mathcal{V}$ be a set of variables over Sig. We define the predicate 'is SSC wrt. Sig' inductively over the syntax of a `Specification`.

- A `Specification` **sort** $n_1 \cdots n_m$ with $m \geq 1$ is SSC wrt. Sig iff all $n_1, \ldots, n_m$ are pairwise different.
- A `Specification` **func** $n_{11}, \ldots, n_{1l_1}{:}S_{11} \times \cdots \times S_{1k_1} \rightarrow S_1$
  $\vdots$
  $n_{m1}, \ldots, n_{ml_m}{:}S_{m1} \times \cdots \times S_{mk_m} \rightarrow S_m$

  with $m \geq 1$, $l_i \geq 1$, $k_i \geq 0$ for $1 \leq i \leq m$ is SSC wrt. Sig iff
  - for all $1 \leq i \leq m$ $n_{i1}, \ldots, n_{il_i}$ are pairwise different,
  - for all $1 \leq i < j \leq m$ it holds that if $n_{ik} \equiv n_{jk'}$ for some $1 \leq k \leq l_i$ and $1 \leq k' \leq l_j$, then either $k_i \neq k_j$, or $S_{il} \not\equiv S_{jl}$ for some $1 \leq l \leq k_i$,
  - for all $1 \leq i \leq m$ and $1 \leq j \leq k_i$ it holds that $S_{ij} \in \mathit{Sig.Sort}$ and $S_i \in \mathit{Sig.Sort}$.

- A `Specification` **map** $n_{11},\ldots,n_{1l_1}{:}S_{11}\times\cdots\times S_{1k_1}\to S_1$
  $\vdots$
  $n_{m1},\ldots,n_{ml_m}{:}S_{m1}\times\cdots\times S_{mk_m}\to S_m$

  with $m\geq 1$, $l_i\geq 1$, $k_i\geq 0$ for $1\leq i\leq m$ is SSC wrt. Sig iff
  - for all $1\leq i\leq m$ $n_{i1},\ldots,n_{il_i}$ are pairwise different,
  - for all $1\leq i<j\leq m$ it holds that if $n_{ik}\equiv n_{jk'}$ for some $1\leq k\leq l_i$ and $1\leq k'\leq l_j$, then either $k_i\neq k_j$, or $S_{il}\not\equiv S_{jl}$ for some $1\leq l\leq k_i$,
  - for all $1\leq i\leq m$ and $1\leq j\leq k_i$ it holds that $S_{ij}\in Sig.Sort$ and $S_i\in Sig.Sort$.

- A `Specification` of the form: *var-dec*
  rew-rul

  where *var-dec* is a `Variable-declaration` and *rew-rul* is a *Equation-section* is SSC wrt. Sig iff
  - *var-dec* is SSC wrt. Sig,
  - *rew-rul* is SSC wrt. Sig and $Vars$(*var-dec*).

- A `Specification` **act** $ad_1\cdots ad_m$ with $m\geq 1$ is SSC wrt. Sig iff
  - for all $1\leq i\leq m$ the `Action-declaration` ad_i is SSC wrt. Sig,
  - for all $1\leq i<j\leq m$ it holds that $Sig(\textbf{act}\ ad_i).Act\cap Sig(\textbf{act}\ ad_j).Act=\emptyset$.

- A `Specification` **comm** $n_{11}|n_{12}=n_{13}\cdots n_{m1}|n_{m2}=n_{m3}$ with $m\geq 1$ is SSC wrt. Sig iff
  - for each $1\leq i<j\leq m$ it is not the case that $n_{i1}\equiv n_{j1}$ and $n_{i2}\equiv n_{j2}$, or $n_{i1}\equiv n_{j2}$ and $n_{i2}\equiv n_{j1}$,
  - for each $1\leq i\leq m$ either $n_{i1}\in Sig.Act$ or there is a $k\geq 1$ such that $n_{i1}{:}S_1\times\cdots\times S_k\in Sig.Act$,
  - for each $1\leq i\leq m$, $k\geq 1$ and `Names` $S_1,\ldots,S_k$ it holds that if $n_{i1}{:}S_1\times\cdots\times S_k\in Sig.Act$ then $n_{i2}{:}S_1\times\cdots\times S_k\in Sig.Act$ and $n_{i3}{:}S_1\times\cdots\times S_k\in Sig.Act$,
  - for each $1\leq i\leq m$, $k\geq 1$ and `Names` $S_1,\ldots,S_k$ it holds that if $n_{i2}{:}S_1\times\cdots\times S_k\in Sig.Act$ then $n_{i1}{:}S_1\times\cdots\times S_k\in Sig.Act$ and $n_{i3}{:}S_1\times\cdots\times S_k\in Sig.Act$,
  - for each $1\leq i\leq m$ it holds that if $n_{i1}\in Sig.Act$ then $n_{i2}\in Sig.Act$ and $n_{i3}\in Sig.Act$,
  - for each $1\leq i\leq m$ it holds that if $n_{i2}\in Sig.Act$ then $n_{i1}\in Sig.Act$ and $n_{i3}\in Sig.Act$.

- A `specification` **proc** $pd_1\cdots pd_m$ with $m\geq 1$ is SSC wrt. Sig iff
  - for each $1\leq i<j\leq m$:
    - if $pd_i\equiv n=p$ and $pd_j\equiv n'=p'$ then $n\not\equiv n'$,
    - if for some $k\geq 1$ it holds that $pd_i\equiv n(x_1{:}S_1,\ldots,x_k{:}S_k)=p$ and $pd_j\equiv n'(x'_1{:}S_1,\ldots,x'_k{:}S_k)=p'$ then $n\not\equiv n'$,
  - if $pd_i\equiv n=p$ $(1\leq i\leq m)$, then
    - $n\notin Sig.Act$ and p is SSC wrt. Sig and $\emptyset$, and
    - for each `Name` S' it holds that $n{:}\to S'\notin Sig.Fun\cup Sig.Map$,
  - if $pd_i\equiv n(x_1{:}S_1,\ldots,x_k{:}S_k)=p$ $(1\leq i\leq m)$, then
    - $n{:}S_1\times\cdots\times S_k\notin Sig.Act$,
    - for each `Name` S' it holds that $n{:}S_1\times\cdots\times S_k\to S'\notin Sig.Fun\cup Sig.Map$,

  - the `Names` $x_1, \ldots, x_k$ are pairwise different and $\{\langle x_j{:}S_j\rangle \mid 1 \leq j \leq k\}$ is a set of variables over Sig,
  - p is SSC wrt. Sig and $\{\langle x_j{:}S_j\rangle \mid 1 \leq j \leq k\}$.

- A `Specification` of the form **init** p is SSC wrt. Sig iff p SSC wrt. to Sig and $\emptyset$.
- A `specification` E_1 E_2 is SSC wrt. Sig iff
  - E_1 and E_2 are SSC wrt. Sig,
  - $Sig(E_1).Sort \cap Sig(E_2).Sort = \emptyset$,
  - if $n{:}S_1 \times \cdots \times S_m \to S \in Sig(E_1).Fun \cup Sig(E_1).Map$ for some $m \geq 0$ then $n{:}S_1 \times \cdots \times S_m \to S' \notin Sig(E_2).Fun \cup Sig(E_2).Map$ for any `Name` S',
  - $Sig(E_1).Act \cap Sig(E_2).Act = \emptyset$,
  - if $n_1|n_2 = n_3 \in Sig(E_1).Comm$ then for any `Names` n_3' and n_3'' $n_1|n_2 = n_3' \notin Sig(E_2).Comm$ and $n_2|n_1 = n_3'' \notin Sig(E_2).Comm$,
  - if $pd_1 \in Sig(E_1).Proc$ and $pd_2 \in Sig(E_2).Proc$, then
    - if $pd_1 \equiv n_1 = p_1$ and $pd_2 \equiv n_2 = p_2$, then $n_1 \not\equiv n_2$,
    - if for some $m \geq 1$ $pd_1 \equiv n_1(x_1{:}S_1, \ldots, x_m{:}S_m) = p_1$ and $pd_2 \equiv n_2(x_1'{:}S_1, \ldots, x_m'{:}S_m) = p_2$, then $n_1 \not\equiv n_2$,
  - $Sig(E_1).Init = \emptyset$ or $Sig(E_2).Init = \emptyset$.

Definition B.13. Let E be a `Specification`. We say that E is SSC iff E is SSC wrt. $Sig(E)$.

B.4 The communication function

The following definition helps us in guaranteeing that the communication function is commutative and associative. This implies that the merge is also commutative and associative which allows us to write parallel processes without brackets.

Definition B.14. Let Sig be a signature. The set $Sig.Comm^*$ is defined by:

$$Sig.Comm^* \stackrel{\text{def}}{=} \{n_1|n_2 = n_3,\ n_2|n_1 = n_3 \mid n_1|n_2 = n_3 \in Sig.Comm\}.$$

So, in $Sig.Comm^*$ communication is always commutative. A *specification* E is *communication-associative* iff

$$n_1|n_2 = n,\ n|n_3 = n' \in Sig(E).Comm^* \Rightarrow \exists n''{:}\ n_2|n_3 = n'', n_1|n'' = n' \in Sig(E).Comm^*.$$

With the condition that E is SSC this exactly implies that communication is associative.

B.5 Well-formed μCRL specifications

We define what well-formed specifications are. We only provide well-formed `Specification`s with a semantics. Well-formedness is a decidable property.

Definition B.15. Let Sig be a signature. We call a `Name` S a *constructor sort* iff $S \in Sig.Sort$ and there exists `Names` $S_1, \ldots, S_k, f$ $(k \geq 0)$ such that $f{:}S_1 \times \cdots \times S_k \to S \in Sig.Fun$.

Definition B.16. Let E be a `Specification` that is SSC. We inductively define which sorts are *non empty constructor sorts* in E. A constructor sort S is called *non empty* iff there is a function $f{:}S_1 \times \cdots \times S_k \to S \in Sig.Fun$ $(k \geq 0)$ such that for all $1 \leq i \leq k$ if S_i is a constructor sort, it is non empty. We say that E has *no empty constructor sorts* iff each constructor sort is non empty.

Definition B.17. Let E be a `Specification`. E is called *well-formed* iff

- E is SSC,
- E is communication-associative,
- E has no empty constructor sorts,
- $\mathbf{Bool} \in Sig(E).Sort$, $T\colon \to \mathbf{Bool} \in Sig(E).Fun$ and $F\colon \to \mathbf{Bool} \in Sig(E).Fun$,
- If $\mathbf{Time} \in Sig(E).Sort$, then $\mathbf{0}\colon \to \mathbf{Time} \in Sig(E).Fun \cup Sig(E).Map$ and $\leq : \mathbf{Time} \times \mathbf{Time} \to \mathbf{Bool} \in Sig(E).Map$.

C Semantics of the datatypes

In this section we define the models of the data part of well-formed specifications. Given a signature Sig we introduce the class of Sig-algebras. For a well-formed `Specification` E with $Sig(E) = Sig$, we define the subclass of Sig-algebras that form a model for the data part of E. As we want to denote all elements in an algebra, we extend the signature with constants. But first we introduce substitutions.

C.1 Substitutions

We define substitutions on `Data-terms` and `Process-terms`.

Definition C.1. Let Sig be a signature and $\mathcal{V}, \mathcal{W}$ sets of variables over Sig. A *substitution* σ over Sig, $\mathcal{V}$ and $\mathcal{W}$ is a mapping from $\mathcal{V}$ to `Data-terms` that are SSC wrt. Sig and $\mathcal{W}$ such that for each $\langle x{:}S\rangle \in \mathcal{V}$ it holds that $sort_{Sig,\mathcal{W}}(\sigma(\langle x{:}S\rangle)) = S$. Substitutions are extended to `Data-terms` that are SSC wrt. Sig and $\mathcal{V}$ by:

$$\begin{aligned}
&\sigma(x) \stackrel{\text{def}}{=} \sigma(\langle x{:}S\rangle) \quad \text{if } \langle x{:}S\rangle \in \mathcal{V} \text{ for some } \texttt{Name}\ S,\\
&\sigma(n) \stackrel{\text{def}}{=} n \quad \text{if } n\colon \to S \in Sig.Fun \cup Sig.Map,\\
&\sigma(n(t_1, \ldots, t_m)) \stackrel{\text{def}}{=} n(\sigma(t_1), \ldots, \sigma(t_m)).
\end{aligned}$$

Note that $\sigma(t)$ is SSC wrt. Sig and $\mathcal{W}$.

Definition C.2. Let Sig be a signature and $\mathcal{V}, \mathcal{W}$ sets of variables over Sig. Let σ be a substitution over Sig, $\mathcal{V}$ and $\mathcal{W}$. We inductively extend σ to a `Process-terms` that are SSC wrt. Sig and $\mathcal{V}$ as follows:

- $\sigma(p_1 \Box p_2) \stackrel{\text{def}}{=} \sigma(p_1) \Box \sigma(p_2)$ if $\Box \in \{+, \|, \between, |, \cdot, \ll\}$,
- $\sigma(p_1 \lhd t \rhd p_2) \stackrel{\text{def}}{=} \sigma(p_1) \lhd \sigma(t) \rhd \sigma(p_2)$,
- $\sigma(p \text{`} t) \stackrel{\text{def}}{=} \sigma(p) \text{`} \sigma(t)$,
- $\sigma(\delta) \stackrel{\text{def}}{=} \delta$
- $\sigma(\tau) \stackrel{\text{def}}{=} \tau$,
- $\sigma(\Box_H(p)) \stackrel{\text{def}}{=} \Box_H(\sigma(p))$ if $\Box \in \{\partial, \tau, \rho\}$,

- $\sigma(\Sigma_{d:D} p) \stackrel{\text{def}}{=} \Sigma_{e:D} \sigma'(p)$ where σ' is a substitution over Sig, $\mathcal{V} \cup \{\langle e{:}D\rangle\}$ and $\mathcal{W}$ defined by

$$\sigma'(\langle x{:}S\rangle) = \begin{cases} \sigma(\langle x{:}S\rangle) & \text{for all } \langle x{:}S\rangle \in \mathcal{V}, \\ e & \text{if } x = d \text{ and } S = D. \end{cases}$$

  Here e is a fresh variable, i.e. for any `Name` S $\langle e{:}S\rangle \notin \mathcal{V} \cup \mathcal{W}$ and $\mathcal{V} \cup \{\langle e{:}D\rangle\}$ is a set of variables over Sig.

- $\sigma(n(t_1, \ldots, t_m)) \stackrel{\text{def}}{=} n(\sigma(t_1), \ldots, \sigma(t_m))$, and
- $\sigma(n) \stackrel{\text{def}}{=} n$.

C.2 Equations

We define the function *eqns* that extracts the equations together with declared variables from a `Specification`.

Definition C.3. We define the function *eqns* on a `Specification` E inductively as follows:

- If E is a `Sort-specification`, `Function-specification`, `Action-specification`, `Communication-specification`, `Process-specification` or an `Initial-declaration` then

$$eqns(E) \stackrel{\text{def}}{=} \emptyset.$$

- If $E \equiv V\ R$ with V a `Variable-declaration` and $R \equiv$ **rew** $rd_1 \cdots rd_m$ an `Equation-section` for some $m \geq 1$, then

$$eqns(E) \stackrel{\text{def}}{=} \{\langle rd_i, Vars(V)\rangle \mid 1 \leq i \leq m\}.$$

- If $E \equiv E_1\ E_2$ where E_1 and E_2 are `Specifications`, then

$$eqns(E) \stackrel{\text{def}}{=} eqns(E_1) \cup eqns(E_2).$$

C.3 Algebras

Definition C.4. Let E be a well-formed `Specification` and let $Sorts \supseteq Sig(E).Sort$ be a set of `Names`. A *$Sig(E)$-algebra* is a tuple $\mathbb{A} = (\{D_S | S \in Sorts\}, I)$ where

- $\{D_S | S \in Sorts\}$ is a collection of non empty sets,
- I is a function from $Sig(E).Fun \cup Sig(E).Map$ to functions over $\bigcup_{S \in Sorts} D_S$ such that for every $f{:}S_1 \times \cdots \times S_n \to S \in Sig(E).Fun \cup Sig(E).Map$, the function $I(f{:}S_1 \times \cdots \times S_n \to S)$ runs from $D_{S_1} \times \cdots \times D_{S_n}$ to D_S. $I(f{:}S_1 \times \cdots \times S_n \to S)$ is called the *interpretation* of the function f.

We define the interpretation $[\![\cdot]\!]_{\mathbb{A}}$ from `Data-terms` that are SSC wrt. $Sig(E)$ and $\emptyset$ to the domains of $\mathbb{A}$ as follows:

- if $t \equiv n$, then $[\![t]\!]_{\mathbb{A}} \stackrel{\text{def}}{=} I(n{:} \to sort_{Sig(E),\emptyset}(n))$,
- if $t \equiv n(t_1, \ldots, t_m)$ for some $m \geq 1$, then $[\![t]\!]_{\mathbb{A}} \stackrel{\text{def}}{=} I(n{:}sort_{Sig(E),\emptyset}(t_1) \times \cdots \times sort_{Sig(E),\emptyset}(t_m) \to sort_{Sig(E),\emptyset}(t))([\![t_1]\!]_{\mathbb{A}}, \ldots, [\![t_m]\!]_{\mathbb{A}})$.

For `Data-terms` t and u that are SSC wrt. E and $\emptyset$ we write $\mathbb{A} \models t = u$ iff $[\![t]\!]_{\mathbb{A}} = [\![u]\!]_{\mathbb{A}}$.

C.4 Extensions of signatures and algebras

Definition C.5. Let E be a well formed `Specification`, *Sorts* a set of `Names` and $\mathbb{A} = (\{D_S | S \in Sorts\}, I)$ a model of E. The *extended signature* $Sig(E, \mathbb{A})$ is defined as follows:

- $Sig(E, \mathbb{A}).Sort = Sig(E).Sort$,
- $Sig(E, \mathbb{A}).Fun = Sig(E).Fun \cup \{c_u{:} \to S | u \in D_s, S \in Sorts\}$ where c_u are fresh names, i.e. $c_u{:} \to S \notin Sig(E).Fun \cup Sig(E).Map$ for any `Name` S and $c_u \notin Sig(E).Act$ and for all `Process-terms` p $c_u = p \notin Sig(E).Proc$,
- $Sig(E, \mathbb{A}).Map = Sig(E).Map$,
- $Sig(E, \mathbb{A}).Act = Sig(E).Act$,
- $Sig(E, \mathbb{A}).Comm = Sig(E).Comm$,
- $Sig(E, \mathbb{A}).Proc = Sig(E).Proc$,
- $Sig(E, \mathbb{A}).Init = Sig(E).Init$.

The *extension of an algebra* $\mathbb{A}$ is the algebra $\mathbb{A}_{ext} = (\{D_S | S \in Sorts\}, I')$ where I' is defined by

$$I'(f{:}S_1 \times \cdots \times S_n \to S) = \begin{cases} u & \text{if } f = c_u \text{ and } n = 0 \\ I(f{:}S_1 \times \cdots \times S_n \to S) & \text{otherwise.} \end{cases}$$

C.5 Model

Definition C.6. Let E be a well-formed `Specification` and $Sorts \supseteq Sig(E).Sort$ a set of `Names`. A $Sig(E)$-algebra $\mathbb{A} = (\{D_S | S \in Sorts\}, I)$ is a *model* of E, notation $\mathbb{A} \models E$, iff

1. whenever $\langle t = t', \mathcal{V} \rangle \in eqns(E)$, then for any substitution σ over $Sig(E, \mathbb{A})$, $\mathcal{V}$ and $\emptyset$ it holds that $\mathbb{A}_{ext} \models \sigma(t) = \sigma(t')$.
2. if $S \in Sig(E).Sort$ is a constructor sort, then for every element $u \in D_S$ there is a function $f{:}S_1 \times \cdots \times S_m \to S \in Sig(E).Fun$ such that $u = I(f{:}S_1 \times \cdots \times S_m \to S)(a_1, \ldots, a_m)$ with $a_i \in D_{S_i}$.
3. The set $D_{\mathbf{Bool}}$ contains exactly two elements, and $I(\mathsf{t}{:} \to \mathbf{Bool})$ and $I(\mathsf{f}{:} \to \mathbf{Bool})$ are different.
4. The `Name` **Time** $\in Sorts$ and there exists functions $\mathbf{0}_{\mathbb{A}}{:} \to D_{\mathbf{Time}}$ and $\leq_{\mathbb{A}} : D_{\mathbf{Time}} \times D_{\mathbf{Time}} \to D_{\mathbf{Bool}}$ such that
   - if $t_1 \leq_{\mathbb{A}} t_2 = \mathsf{t}_{\mathbb{A}}$ and $t_2 \leq_{\mathbb{A}} t_3 = \mathsf{t}_{\mathbb{A}}$ then $t_1 \leq_{\mathbb{A}} t_3 = \mathsf{t}_{\mathbb{A}}$ for all $t_1, t_2, t_3 \in D_{\mathbf{Time}}$,
   - $\mathbf{0}_{\mathbb{A}} \leq_{\mathbb{A}} t = \mathsf{t}_{\mathbb{A}}$ for all $t \in D_{\mathbf{Time}}$,
   - Either $t_1 \leq_{\mathbb{A}} t_2$ or $t_2 \leq_{\mathbb{A}} t_1$ for all $t_1, t_2 \in D_{\mathbf{Time}}$,
   - for all $t_1, t_2 \in D_{\mathbf{Time}}$ $t_1 \leq_{\mathbb{A}} t_2 = \mathsf{t}_{\mathbb{A}}$ and $t_2 \leq_{\mathbb{A}} t_1 = \mathsf{t}_{\mathbb{A}}$ iff $t_1 = t_2$.

   Here $\mathsf{t}_{\mathbb{A}} = I(\mathsf{t}{:} \to \mathbf{Bool})$. Moreover, if $\mathbf{Time} \in Sig(E).Sort$, then $\mathbf{0}_{\mathbb{A}} = I(\mathbf{0}{:} \to \mathbf{Time})$ and $\leq_{\mathbb{A}} = I(\leq : \mathbf{Time} \times \mathbf{Time} \to \mathbf{Time})$.

D Operational semantics

In this section we assume that a well-formed specification E with some timed μCRL model $\mathbb{A}$ are given. We define for each process-term p that is SSC wrt. $Sig(E)$ and $\emptyset$, a transition system, explaining the operational behaviour of such a term.

D.1 The delay predicate

The *delay predicate* $U(t,p)$ expresses that p can at least idle until time t. It has the same purpose as the ultimate delay in [1, 2, 27].

Definition D.1. Let E be a well-formed `Specification` that is SSC and $Sorts \supseteq Sig(E).Sort$ a set of `Name`s. Let $\mathbb{A} = (\{D_S | S \in Sorts\}, I)$ be a model of E. Let p be a `Process-term` that is SSC wrt. $Sig(E, \mathbb{A})$ and $\emptyset$, and $t \in D_{\mathbf{Time}}$. The ultimate delay $U(t,p)$ is the least predicate satisfying:

$U(t,a)$, $U(t,a(u_1,\ldots,u_n))$, $U(t,\tau)$ and $U(t,\delta)$ hold;
$U(t,p+q)$ holds iff $U(t,p)$ or $U(t,q)$;
$U(t,p{\cdot}q)$ holds iff $U(t,p)$;
$U(t,p \lhd c \rhd q)$ holds iff $U(t,p)$ and $\mathbb{A}_{ext} \models c = \mathsf{t}$, or $U(t,q)$ and $\mathbb{A}_{ext} \models c = \mathsf{f}$;
$U(t,\sum_{d:S} p)$ holds iff for some $u \in D_S$ $U(t,\sigma(p))$, where σ is a substitution over $Sig(E,\mathbb{A})$, $\{\langle d{:}S\rangle\}$ and $\emptyset$ such that $\sigma(d) = u$;
$U(t,\rho_R(p))$, $U(t,\tau_I(p))$ and $U(t,\partial_H(p))$ hold iff $U(t,p)$;
$U(t,p \parallel q)$, $U(t,p \mid q)$, $U(t,p \between q)$ and $U(t,p \ll q)$ hold iff $U(t,p)$ and $U(t,q)$;
$U(t,p \lhd t')$ holds iff $t \leq_{\mathbb{A}} [\![t']\!]_{\mathbb{A}_{ext}}$ and $U(t,p)$;
$U(t,X)$ holds iff $U(t,p)$ and $X = p \in Sig(E).proc$;
$U(t,X(u_1,\ldots,u_n))$ holds iff $U(t,\sigma(p))$ where σ is a substitution over $Sig(E,\mathbb{A})$, $\{\langle x_i{:}S_i\rangle | 1 \leq i \leq n\}$ and $\emptyset$ such that $\sigma(\langle x_i{:}S_i\rangle) = u_i$, and $X(x_1{:}S_1,\ldots,x_n{:}S_n) = p \in Sig(E).proc$ and $sort_{Sig,\emptyset}(u_i) = S_i$.

D.2 Transition system (general)

Definition D.2. A (timed) transition system $\mathcal{A}$ is a quadruple $(S, L, \longrightarrow, s)$ where

- S is a set of *states*;
- L is a set of *labels*;
- $\longrightarrow \subseteq (S \times L \times S) \cup (S \times S)$ is a *transition relation*;
- $s \in S$ is the *initial state*.

Elements $(s', l, s'') \in \longrightarrow$ are generally written as $s' \xrightarrow{l} s''$. Elements $(s', s'') \in \longrightarrow$ are called idle steps and are generally written as $s' \xrightarrow{\imath} s''$.

D.3 Transition system (specific)

Definition D.3. Let E be a well-formed `Specification` and $Sorts \supseteq Sig(E).Sort$ a set of `Name`s. Let $\mathbb{A} = (\{D_S | S \in Sorts\}, I)$ be a model of E. Let p be a `Process-term` that is SSC wrt. $Sig(E, \mathbb{A})$ and $\emptyset$. The *meaning* of p in the context of E and $\mathbb{A}$ is the timed transition system $\mathcal{A}(p, E, \mathbb{A})$ defined by

$$(S, L, \longrightarrow, s)$$

where

- $S \stackrel{\text{def}}{=} (Term \cup \{\surd\}) \times D_{\mathbf{Time}}$ where $Term$ are the `Process-term`s that are SSC wrt. $Sig(E, \mathbb{A})$ and $\emptyset$ and $\surd$ is a special termination symbol;
- $L \stackrel{\text{def}}{=} \{\tau\} \cup \{n(c_{u_1},\ldots,c_{u_m}) \mid m \geq 0, n \in Sig(E).Act$ and $u_i \in D_S$ for some $S \in Sorts\}$;

$\langle p, t\rangle \xrightarrow{t} \langle p, t'\rangle$ iff $U(t', p)$ and not $t' \leq_{\mathbf{A}} t$

Table 8: Rule for time passsing

- $s \stackrel{\text{def}}{=} \langle p, \mathbf{0}_{\mathbf{A}}\rangle$,
- $\longrightarrow$ is the transition relation that contains exactly all transitions provable using the rules in Tables 8, 9, 10 and 11. In these tables p, q, p' and q' range over `Process-terms` that are SSC wrt. $Sig(E)$ and $\emptyset$; $\overline{p}, \overline{q}$ may be either equal to $\surd$, or can be a `Process-term` which is SSC wrt. to $Sig(E)$ and $\emptyset$. P is a `Process-term` which is SSC wrt. $Sig(E)$ and some set of variables $\mathcal{V}$. The variables $t, t' \in D_{\mathbf{Time}}$, l and l' range over the set L of labels, n, n_i are `Names`, $m \geq 0$ unless stated otherwise, $k \geq 1$ and $u, u_1, \ldots, u_m$ are `Data-terms` with $S \in Sorts$. In Table 11, the conditions below a pair of rules apply to both rules.

Definition D.4. Let E be a well-formed `Specification` containing an `Initial-Declaration` **init** p and let $Sorts \supseteq Sig(E).Sorts$ be a set of `Nams`. Let $\mathbf{A} = (\{D_S | S \in Sorts\}, I)$ be a model of E. The *meaning* of E in the context of $\mathbf{A}$ is the timed transition system $\mathcal{A}(p, E, \mathbf{A})$.

References

[1] J.C.M. Baeten and J.A. Bergstra. Real time process algebra, *Formal Aspects of Computing* 3(2):142-188, 1991.

[2] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra, *Formal Aspects of Computing* 8(2):188-208, 1996.

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

[4] H.P. Barendregt. *The lambda calculus*. Studies in logic and the foundations of mathematics, Volume 103, North-Holand, 1981.

[5] M.A. Bezem, R.N. Bol and J.F. Groote. Formalizing Process Algebraic Verifications in the Calculus of Constructions. *Formal Aspects of Computing*, 9:1-48, 1997.

[6] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proceedings Concur'94*, Uppsala, Sweden, Lecture Notes in Computer Science no. 836, pages 401-416, Springer Verlag, 1994.

[7] M.A. Bezem and J.F. Groote. A correctness proof of a one bit sliding window protocol in μCRL. The Computer Journal, 37(4): 289-307, 1994.

[8] D. Bosscher and A. Ponse. Translating a process algebra with symbolic data values to linear format. In U.H. Engberg, K.G. Larsen, and A. Skou, editors, Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Aarhus 1995, BRICS Notes Series, pages 119–130. University of Aarhus, 1995.

[9] D. Dams and J.F. Groote. Specification and Implementation of Components of a μCRL Toolbox. Technical Report 152, Logic Group Preprint Series, Department of Philosophy, Utrecht University, 1995.

$\langle \tau, t\rangle \stackrel{\tau}{\longrightarrow} \langle \surd, t\rangle$
$\langle n, t\rangle \stackrel{n()}{\longrightarrow} \langle \surd, t\rangle \qquad n \in Sig(E).Act$ $\langle n(u_1, \ldots, u_m), t\rangle \stackrel{n(c_{[\![u_1]\!]_{\mathbb{A}_{ext}}}, \ldots, c_{[\![u_m]\!]_{\mathbb{A}_{ext}}})}{\longrightarrow} \langle \surd, t\rangle$ $n{:}sort_{Sig(E,\mathbb{A}),\emptyset}(u_1) \times \cdots \times sort_{Sig(E,\mathbb{A}),\emptyset}(u_m) \in Sig(E, \mathbb{A}).Act$
$\dfrac{\langle p, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle}{\langle n, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle} \qquad n = p \in Sig(E, \mathbb{A}).Proc$ $\dfrac{\langle \sigma(P), t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle}{\langle n(u_1, \ldots, u_m), t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle}$ $n(x_1{:}S_1, \ldots, x_m{:}S_m) = P \in Sig(E, \mathbb{A}).Proc$, σ is a substitution over $Sig(E, \mathbb{A})$ $\{\langle x_1{:}S_1\rangle, \ldots, \langle x_m{:}S_m\rangle\}$ and $\emptyset$ such that $\sigma(\langle x_i{:}S_i\rangle) \equiv u_i$ and $S_i = sort_{Sig(E,\mathbb{A}),\emptyset}(u_i)$ for $1 \leq i \leq m$
$\dfrac{\langle p, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle}{\langle p + q, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle} \qquad \dfrac{\langle q, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{q}, t\rangle}{\langle p + q, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{q}, t\rangle}$
$\dfrac{\langle p, t\rangle \stackrel{l}{\longrightarrow} \langle p', t\rangle}{\langle p{\cdot}q, t\rangle \stackrel{l}{\longrightarrow} \langle p'{\cdot}q, t\rangle} \qquad \dfrac{\langle p, t\rangle \stackrel{l}{\longrightarrow} \langle \surd, t\rangle}{\langle p{\cdot}q, t\rangle \stackrel{l}{\longrightarrow} \langle q, t\rangle}$
$\dfrac{\langle p, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle}{\langle p \lhd u \rhd q, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle} \quad \mathbb{A}_{ext} \models u = \mathsf{t} \qquad \dfrac{\langle q, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{q}, t\rangle}{\langle p \lhd u \rhd q, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{q}, t\rangle} \quad \mathbb{A}_{ext} \models u = \mathsf{f}$
$\dfrac{\langle \sigma(P), t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle}{\langle \sum_{x:S} P, t\rangle \stackrel{l}{\longrightarrow} \langle \overline{p}, t\rangle}$ σ is a substitution over $Sig(E, \mathbb{A})$, $\{\langle x{:}S\rangle\}$ and $\emptyset$

Table 9: Rules for the basic operators for timed μCRL ($m \geq 1$)

$$\frac{\langle p,t\rangle \xrightarrow{l} \langle p',t\rangle}{\langle p \parallel q,t\rangle \xrightarrow{l} \langle p' \parallel q,t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{l} \langle \surd,t\rangle}{\langle p \parallel q,t\rangle \xrightarrow{l} \langle q,t\rangle}$$

$$\frac{\langle q,t\rangle \xrightarrow{l} \langle q',t\rangle}{\langle p \parallel q,t\rangle \xrightarrow{l} \langle p \parallel q',t\rangle} \qquad \frac{\langle q,t\rangle \xrightarrow{l} \langle \surd,t\rangle}{\langle p \parallel q,t\rangle \xrightarrow{l} \langle p,t\rangle}$$

$$\frac{\langle p,t\rangle \xrightarrow{n_1(u_1,\ldots,u_m)} \langle p',t\rangle \quad \langle q,t\rangle \xrightarrow{n_2(u_1,\ldots,u_m)} \langle q',t\rangle}{\langle p \parallel q,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle p' \parallel q',t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{n_1(u_1,\ldots,u_m)} \langle \surd,t\rangle \quad \langle q,t\rangle \xrightarrow{n_2(u_1,\ldots,u_m)} \langle q',t\rangle}{\langle p \parallel q,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle q',t\rangle}$$

$$\frac{\langle p,t\rangle \xrightarrow{n_1(u_1,\ldots,u_m)} \langle p',t\rangle \quad \langle q,t\rangle \xrightarrow{n_2(u_1,\ldots,u_m)} \langle \surd,t\rangle}{\langle p \parallel q,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle p',t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{n_1(u_1,\ldots,u_m)} \langle \surd,t\rangle \quad \langle q,t\rangle \xrightarrow{n_2(u_1,\ldots,u_m)} \langle \surd,t\rangle}{\langle p \parallel q,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle \surd,t\rangle}$$

$$\frac{\langle p,t\rangle \xrightarrow{n_1(u_1,\ldots,u_m)} \langle p',t\rangle \quad \langle q,t\rangle \xrightarrow{n_2(u_1,\ldots,u_m)} \langle q',t\rangle}{\langle p|q,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle p' \parallel q',t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{n_1(u_1,\ldots,u_m)} \langle \surd,t\rangle \quad \langle q,t\rangle \xrightarrow{n_2(u_1,\ldots,u_m)} \langle q',t\rangle}{\langle p|q,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle q',t\rangle}$$

$$\frac{\langle p,t\rangle \xrightarrow{n_1(u_1,\ldots,u_m)} \langle p',t\rangle \quad \langle q,t\rangle \xrightarrow{n_2(u_1,\ldots,u_m)} \langle \surd,t\rangle}{\langle p|q,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle p',t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{n_1(u_1,\ldots,u_m)} \langle \surd,t\rangle \quad \langle q,t\rangle \xrightarrow{n_2(u_1,\ldots,u_m)} \langle \surd,t\rangle}{\langle p|q,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle \surd,t\rangle}$$

$$\frac{\langle p,t\rangle \xrightarrow{l} \langle p',t\rangle}{\langle p \between q,t\rangle \xrightarrow{l} \langle p' \parallel q,t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{l} \langle \surd,t\rangle}{\langle p \between q,t\rangle \xrightarrow{l} \langle q,t\rangle}$$

Table 10: Rules for the parallel operators of timed μCRL ($n_1 \mid n_2 = n \in Sig(E).Comm^*$)

$$\frac{\langle p,t\rangle \xrightarrow{l} \langle p',t\rangle}{\langle \tau_{\{n_1,\ldots,n_k\}}p,t\rangle \xrightarrow{l} \langle \tau_{\{n_1,\ldots,n_k\}}p',t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{l} \langle \surd,t\rangle}{\langle \tau_{\{n_1,\ldots,n_k\}}p,t\rangle \xrightarrow{l} \langle \surd,t\rangle}$$

if $l \equiv n(u_1,\ldots,u_m)$ then $n \not\equiv n_i$ for all $1 \leq i \leq k$

$$\frac{\langle p,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle p',t\rangle}{\langle \tau_{\{n_1,\ldots,n_k\}}p,t\rangle \xrightarrow{\tau} \langle \tau_{\{n_1,\ldots,n_k\}}p',t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{n(u_1,\ldots,u_m)} \langle \surd,t\rangle}{\langle \tau_{\{n_1,\ldots,n_k\}}p,t\rangle \xrightarrow{\tau} \langle \surd,t\rangle}$$

$n \equiv n_i$ for some $1 \leq i \leq k$,

$$\frac{\langle p,t\rangle \xrightarrow{l} \langle p',t\rangle}{\langle \rho_{\{n_1\to n'_1,\ldots,n_k\to n'_k\}}p,t\rangle \xrightarrow{l'} \langle \rho_{\{n_1\to n'_1,\ldots,n_k\to n'_k\}}p',t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{l} \langle \surd,t\rangle}{\langle \rho_{\{n_1\to n'_1,\ldots,n_k\to n'_k\}}p,t\rangle \xrightarrow{l'} \langle \surd,t\rangle}$$

if $l \equiv n_i(u_1,\ldots,u_m)$ then $l' \equiv n'_i(u_1,\ldots,u_m)$. Otherwise $l \equiv l'$

$$\frac{\langle p,t\rangle \xrightarrow{l} \langle p',t\rangle}{\langle \partial_{\{n_1,\ldots,n_k\}}p,t\rangle \xrightarrow{l} \langle \partial_{\{n_1,\ldots,n_k\}}p',t\rangle} \qquad \frac{\langle p,t\rangle \xrightarrow{l} \langle p',t\rangle}{\langle \partial_{\{n_1,\ldots,n_k\}}p,t\rangle \xrightarrow{l} \langle \partial_{\{n_1,\ldots,n_k\}}p',t\rangle}$$

$l \not\equiv n_i(u_1,\ldots,u_m)$ for all $1 \leq i \leq k$

$$\frac{\langle p,t\rangle \xrightarrow{l} \langle \overline{p},t\rangle}{\langle p \lhd t,t\rangle \xrightarrow{l} \langle \overline{p},t\rangle}$$

$$\frac{\langle p,t\rangle \xrightarrow{l} \langle \overline{p},t\rangle}{\langle p \ll q,t\rangle \xrightarrow{l} \langle \overline{p},t\rangle} \quad U(t,q)$$

Table 11: Rules for hiding, encapsulation, renaming and time for timed μCRL

[10] W.J. Fokkink and A.S. Klusener. An effective axiomatization for real time ACP. Information and computation 122(2):286-299, 1995.

[11] L.-Å. Fredlund, J.F. Groote and H. Korver. Formal Verification of a Leader Election Protocol in Process Algebra. *Theoretical Computer Science*, 177:459-486, 1997.

[12] R.J. van Glabbeek. The linear time – branching time spectrum. Extended abstract in J.C.M. Baeten & J.W. Klop, editors: Proceedings *CONCUR '90, Theories of Concurrency: Unification and Extension*, Amsterdam, August 1990, LNCS 458, Springer-Verlag, pp. 278–297, 1990.

[13] R.J. van Glabbeek. The Linear Time – Branching Time Spectrum II; The semantics of sequential systems with silent moves (extended abstract). In E. Best, editor, *Proceedings CONCUR'93,* 4^{th} *International Conference on Concurrency Theory*, Lecture Notes in Computer Science 715, pp. 66-81, Springer-Verlag, 1993.

[14] J.F. Groote and H. Korver. Correctness proof of the bakery protocol in μCRL. In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds, *Algebra of Communicating Processes*, Workshops in Computing, pp. 63-86, 1994.

[15] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. A case study in computer checked verification. In M. Wirsing and M. Nivat, Editors, *Proceedings of AMAST'96*, Munich, Lecture Notes in Computer Science 1101, Springer Verlag, pages 536-550, 1996.

[16] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In Andrews et al. *Proceedings of the International Workshop on Semantics of Specification Languages.* Workshops in Computing, pages 231-250. Springer Verlag, 1994.

[17] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds, *Algebra of Communicating Processes*, Workshops in Computing, pp. 26-62, 1994.

[18] J.F. Groote and M.P.A. Sellink. Confluence for Process Verification. In *Theoretical Computer Science B (Logic, semantics and theory of programming)* Volume 170(1-2):47–81, 1996.

[19] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. Technical Report 142, Logic Group Preprint Series, Utrecht University, 1995. This report also appeared as Technical Report CS-R9566, Centrum voor Wiskunde en Informatica, 1995

[20] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202-260, October 1992.

[21] J.F. Groote and S.F.M. van Vlijmen. A modal logic for μCRL. Modal Logic and Process Algebra, a Bisimulation Perspective A. Ponse, M. de Rijke and Y. Venema, eds. CSLI Lecture Notes No. 53, pages 131-150, Stanford, 1995.

[22] J. Heering, P.R.H. Hendriks, P. Klint and J. Rekers. The syntax definition formalism SDF – reference manual –. *ACM SIGPLAN Notices*, 24(11):43–75. 1989.

[23] T.A. Henzinger, X. Nicollin, J. Sifakis and S. Yovine. Symbolic Model Checking for Real-Time Systems. Information and Computation 111:193-244, 1994.

[24] J.A. Hillebrand and H. Korver. A well-formedness checker for μCRL. Technical Report P9501, University of Amsterdam, Programming Research Group, 1995.

[25] ISO/IEC/JTC1/SC21/WG7 Enhancements to LOTOS. Working Draft on Enhancements to LOTOS, ftp://ftp.dit.upm.es/pub/lotos/elotos/Working.Docs, January 1997.

[26] A.S. Klusener. Abstraction in real time process algebra. In: Real time, theory in practice, Proc. REX91 (J.W. de Bakker, C. Huizing, W.P. de Roever and G. Rozenberg, eds.), Mook 1991, Springer LNCS 600, pp. 325-352, 1990.

[27] A.S. Klusener. Models and axioms for a fragment of real time process algebra. Ph.D. Thesis. Technical University Eindhoven, 1993.

[28] H. Korver. Building a simulator in the μCRL toolbox. A case study in modern software engineering. Report CS-R9632, Centrum voor Wiskunde en Informatica, Amsterdam, 1996.

[29] H. Korver and J. Springintveld. A computer-checked verification of Milner's Scheduler. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'94)*, Sendai, Japan. Lecture Notes in Computer Science 789, Springer-Verlag, 1994.

[30] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1-11, 1987.

[31] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae* XIII:85–139, 1990.

[32] G.D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[33] A. Ponse. Computable processes and bisimulation equivalence. Formal Aspects of Computing, 8:648–678, 1996.

[34] J.J. Vereijken. Fischer's protocol in timed process algebra. In A. Ponse, C. Verhoef and S. F. M. van Vlijmen, editors, Proceedings of ACP95, Report CSR 95/14, Eindhoven University of Technology, pp. 245-283, 1995.