The Syntax and Semantics of μCRL

Jan Friso Groote
Alban Ponse
Department of software technology, CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

A simple specification language based on CRL (*Common Representation Language*) and therefore called μCRL (*micro* CRL) is proposed. It has been developed to study processes with data. So the language contains only basic constructs with an easy semantics. To obtain executability, *effective* μCRL has been defined. In effective μCRL equivalence between closed *data-terms* is decidable and the operational behaviour is finitely branching and computable. This makes effective μCRL a good platform for tooling activities.

Key Words & Phrases: Specification Language, Abstract Data Types, Process Algebra, Operational Semantics.
1985 Mathematics Subject Classification: 68N99.
1987 CR Categories: D.2.1, D.3.1, D.3.3.
Note: The authors are supported by the European Communities under RACE project no. 1046, Specification and Programming Environment for Communication Software (SPECS). The first author is also supported by ESPRIT Basic Research Action 3006 (CONCUR). This document does not necessarily reflect the view of the SPECS project.

1 Introduction

In telecommunication applications the necessity of the use of formal methods has been observed several times. For that purpose several specification languages have been developed (SDL [6], LOTOS [15], PSF [18] and CRL [22]). These languages are designed to optimise usability. However, they turn out to be rather complicated, especially as far as their semantic basis is concerned. An enormous amount of manpower has already been invested into tooling these languages. But, although some major achievements have been made, this turns out to be hard and results often lag behind expectations.

In this paper we define a language called μCRL (*micro* CRL, where CRL stands for Common Representation Language [22]) as it consists of the essence of CRL. It has been developed under the assumption that an extensive study of the basic constructs of specification languages will yield fundamental insights that are hard to obtain via the languages mentioned above. These insights may assist further development of these languages. So our language is indeed very small although its definition still requires quite some pages. As μCRL only contains core constructs, it may not be so well suited as an actual specification language.

An advantage of our 'simple' approach is that when in the future several constructs that are not included in the language will be well understood and will have a concise and natural semantics, we can add them to the language without a time and manpower consuming redesign of existing but not optimally devised features.

The language μCRL consists of data and processes. The data part contains equational specifications: one can declare sorts and functions working upon these sorts, and describe the meaning of these functions by equational axioms. The process part contains processes described in the style of CCS [19], CSP [12] or ACP [2, 3], where the particular process syntax has been taken from ACP. It basically consists of a set of uninterpreted actions that may be parameterised by data. These actions can represent all kinds of real world activities, depending on the usage of the language. There are sequential, alternative and parallel composition operators. Furthermore, recursive processes are specified in a simple way.

An important feature is executability. To obtain this, we define *effective* μCRL. In effective μCRL it is required that the equations specifying data constitute a semi-complete term rewriting system. This implies that data equivalence is decidable. Moreover, the specification of recursive processes must be guarded and sums over data sorts must be finite. This guarantees that the operational behaviour of every effective μCRL specification is finitely branching and computable. We believe that effective μCRL is an excellent base for building tools.

Acknowledgements. The idea for μCRL comes from Jan Bergstra, who had also a pervasive influence on its current form, especially in keeping the language small. We further thank Jos Baeten, Michel Dauphin, Arie van Deursen, Willem Jan Fokkink, Bertrand Gruson, Jan Gustafsson, Georg Karner, Martin Kooij, Henri Korver, Sjouke Mauw, Emma van der Meulen, Jan Rekers and Gert Veltink for their valuable comments.

2 The syntax of μCRL

In this section we present the syntax of μCRL. It contains two major components, namely data specified by a many sorted term rewriting system and processes which are based on process algebra [3]. The syntax is defined in the BNF formalism. Syntactical categories are written in italics and we use a '.' to end each BNF clause. In reasoning about the syntax of μCRL we use the symbol $\equiv$ to denote syntactic equivalence.

2.1 Names

We assume the existence of a set $\mathcal{N}$ of *names* that are used to denote sorts, variables, functions, processes and labels of actions. The *names* in $\mathcal{N}$ are sequences over an alphabet not containing

$$\perp, +, \|, \underline{\|}\, , \mid, \triangleleft, \triangleright, \cdot, \delta, \tau, \partial, \rho, \Sigma, \surd, \times, \rightarrow, :, =, (,), \{, \}, \text{`,'}, \text{ a space and a newline.}$$

The space and the newline serve as separators between names and are used to lay out specifications. The symbol $\perp$ is used in the description of the semantics and the other symbols have special functions. Moreover, $\mathcal{N}$ does not contain the reserved keywords **sort**, **proc**, **var**, **act**, **func**, **comm**, **rew** and **from**.

2.2 Lists

In the sequel *X-list*, $\times$-*X-list*, and *space-X-list* for any syntactical category X are defined by the following BNF syntax:

$$\begin{array}{rcl} \textit{X-list} & ::= & X \mid \textit{X-list}, X. \\ \times\textit{-X-list} & ::= & X \mid \times\textit{-X-list} \times X. \\ \textit{space-X-list} & ::= & X \mid \textit{space-X-list}\ X. \end{array}$$

Lists are often described by the (informal) use of dots, e.g. $b_1 \times ... \times b_m$ with $m \geq 1$ is a $\times$-*X-list* where $b_1, ..., b_m$ are expressions in the syntactical category X. Note that lists cannot be empty.

2.3 Sort specifications

A *sort-specification* consists of a list of *names* representing sorts, preceded by the keyword **sort**.

$$\textit{sort-specification} \quad ::= \quad \textbf{sort}\ \textit{space-name-list}.$$

2.4 Function specifications

A *function-specification* consists of a list of function declarations. A *function-declaration* consists of a *name-list* (the names play the role of constant and function names), the sorts of their parameters and their target sort:

$$\begin{array}{rcl} \textit{function-specification} & ::= & \textbf{func}\ \textit{space-function-declaration-list}. \\ \textit{function-declaration} & ::= & \textit{name-list} : \to \textit{name} \\ & \mid & \textit{name-list} : \times\textit{-name-list} \to \textit{name}. \end{array}$$

2.5 Rewrite specifications

A *rewrite-specification* is given by a many sorted term rewriting system. Its syntax is given by the following BNF grammar:

$$\begin{array}{rcl} \textit{rewrite-specification} & ::= & \textit{variable-declaration-section} \\ & & \textit{rewrite-rules-section}. \end{array}$$

In a *variable-declaration-section* all variables that are used in a *rewrite-rules-section* must be declared. In such a declaration, it is also stated what the sort of a variable is. A variable declaration section may be empty.

$$\begin{array}{rcl} \textit{variable-declaration-section} & ::= & \textbf{var}\ \textit{space-variable-declaration-list} \\ & \mid & . \end{array}$$

In a *variable-declaration*, the *name-list* contains the declared variables and the *name* denotes their sort:

$$\textit{variable-declaration} \quad ::= \quad \textit{name-list} : \textit{name}.$$

Data-terms are defined in the standard way. The *name* without brackets in the syntax represents a variable or a constant.

$$\begin{array}{rcl} \textit{data-term} & ::= & \textit{name} \\ & | & \textit{name}(\textit{data-term-list}). \end{array}$$

The equations in a *rewrite-rules-section* define the meaning of functions operating on data. The syntax of a *rewrite-rules-section* is given by:

$$\begin{array}{rcl} \textit{rewrite-rules-section} & ::= & \textbf{rew}\ \textit{space-rewrite-rule-list}. \\ \textit{rewrite-rule} & ::= & \textit{name} = \textit{data-term} \\ & | & \textit{name}(\textit{data-term-list}) = \textit{data-term}. \end{array}$$

2.6 Process expressions and process specifications

In this section we first define what *process-expressions* look like. Then we define how these expressions can be used to construct *process-specifications*.

Process-expressions are defined via the following syntax explicitly taking care of the precedence among operators:

$$\begin{array}{rcl} \textit{process-expression} & ::= & \textit{parallel-expression} \\ & | & \textit{parallel-expression} + \textit{process-expression}. \end{array}$$

$$\begin{array}{rcl} \textit{parallel-expression} & ::= & \textit{merge-parallel-expression} \\ & | & \textit{comm-parallel-expression} \\ & | & \textit{cond-expression} \\ & | & \textit{cond-expression} \mathbin{|\!\!\lfloor} \textit{cond-expression}. \end{array}$$

$$\begin{array}{rcl} \textit{merge-parallel-expression} & ::= & \textit{cond-expression} \parallel \textit{merge-parallel-expression} \\ & | & \textit{cond-expression} \parallel \textit{cond-expression}. \end{array}$$

$$\begin{array}{rcl} \textit{comm-parallel-expression} & ::= & \textit{cond-expression} \mid \textit{comm-parallel-expression} \\ & | & \textit{cond-expression} \mid \textit{cond-expression}. \end{array}$$

$$\begin{array}{rcl} \textit{cond-expression} & ::= & \textit{dot-expression} \\ & | & \textit{dot-expression} \triangleleft \textit{data-term} \triangleright \textit{dot-expression}. \end{array}$$

$$\begin{array}{rcl} \textit{dot-expression} & ::= & \textit{basic-expression} \\ & | & \textit{basic-expression} \cdot \textit{dot-expression}. \end{array}$$

$$\begin{array}{rcl} \textit{basic-expression} & ::= & \delta \\ & | & \tau \\ & | & \partial(\{\textit{name-list}\}, \textit{process-expression}) \\ & | & \tau(\{\textit{name-list}\}, \textit{process-expression}) \\ & | & \rho(\{\textit{renaming-declaration-list}\}, \textit{process-expression}) \\ & | & \Sigma(\textit{single-variable-declaration}, \textit{process-expression}) \\ & | & \textit{name} \\ & | & \textit{name}(\textit{data-term-list}) \\ & | & (\textit{process-expression}). \end{array}$$

The $+$ is the alternative composition. A *process-expression* $p + q$ behaves exactly as the argument that performs the first step.

The merge or parallel composition operator ($\|$) interleaves the behaviour of both arguments except that some actions in the arguments may communicate, which means that they happen at exactly the same moment and result in a communication action. In a *communication-specification* it can be declared which actions may communicate. The left merge ($\lfloor\!\lfloor$) behaves exactly as the parallel operator, except that its first step must originate from its left argument only. The communication merge ($|$) also behaves as the parallel operator, but now the first action must be a communication between both components. The left merge and the communication merge are added to allow proof theoretic reasoning. It is not expected that they will be used in specifications. In the sequel the syntactical category *parallel-expression* also refers to *merge-parallel-expression* and *comm-parallel-expression*.

The *conditional* construct *dot-expression* $\lhd$ *data-term* $\rhd$ *dot-expression* is an alternative way to write an **if - then - else**-expression and is introduced by HOARE cs. [13] (see also [1]). The *data-term* is supposed to be of the standard sort of the Booleans (**Bool**). The $\lhd$-part is executed if the *data-term* evaluates to true (T) and the $\rhd$-part is executed if the *data-term* evaluates to false (F).

The sequential composition operator '$\cdot$' says that first its left hand side can perform actions, and if it terminates then the second argument continues.

The constant δ describes the process that cannot do anything, especially, it cannot terminate. For instance, the process $\delta \cdot p$ can never perform an action of p. We also expect that δ is not used in specifications, but in reasoning δ is very handy to indicate that at a certain place a deadlock occurs.

The constant τ represents some internal activity that cannot be observed by the environment. It is therefore called the internal action.

The encapsulation operator ∂ is used to prevent actions of which the *name* is mentioned in its first argument from happening. This enables one to force actions into a communication.

The hiding operator, also denoted by a τ, is used to rename actions of which the *name* is mentioned into an internal action.

The renaming operator ρ is more general. It renames the *names* of actions according to the scheme in its first argument. A *renaming-declaration* is given by the following syntax:

$$\textit{renaming-declaration} \quad ::= \quad \textit{name} \rightarrow \textit{name}.$$

The first mentioned *name* is renamed to the second one.

The sum operator is used to declare a variable of a specific sort for use in a *process-expression*. A *single-variable-declaration* is defined by:

$$\textit{single-variable-declaration} ::= \textit{name} : \textit{name}.$$

The scope of the variable is exactly the *process-expression* mentioned in the sum operator. The behaviour of this construct is a choice between the behaviours of *process-expression* in which each value of the sort of the variable has been substituted for the variable.

The constructs *name* and *name*(*data-term-list*) are either process instantiations or actions: *name* refers to a declared process (or to an action) and *data-term-list* contains the arguments of the process identifier (or the action).

The syntax of *process-expressions* says that $\cdot$ binds strongest, the conditional construct binds stronger than the parallel operators which in turn bind stronger than $+$.

A *process-specification* is a list of (parameterised) names, which are used as process identifiers, that are declared together with their bodies.

$$\begin{array}{rcl} \textit{process-specification} & ::= & \textbf{proc}\ \textit{space-process-declaration-list}. \\ \textit{process-declaration} & ::= & \textit{name} = \textit{process-expression} \\ & | & \textit{name}(\textit{single-variable-declaration-list}) = \textit{process-expression}. \end{array}$$

2.7 Action specification

In an *action-specification* all actions that are used are declared. Actions may be parameterised by data, and in that case we must declare on which sorts an action depends. An *action-specification* has the following form:

$$\begin{array}{rcl} \textit{action-specification} & ::= & \textbf{act}\ \textit{space-action-declaration-list}. \\ \textit{action-declaration} & ::= & \textit{name} \\ & | & \textit{name-list} : \times\textit{-name-list}. \end{array}$$

2.8 Communication specification

A *communication-specification* prescribes how actions may communicate. It only describes communication on the level of *names* of actions, e.g. if it is specified that $in \mid out = com$ then each action $in(t_1, ..., t_k)$ can communicate with $out(t'_1, ..., t'_m)$ to $com(t_1, ..., t_k)$ provided $k = m$ and t_i and t'_i denote the same data-element for $i = 1, ..., k$.

$$\begin{array}{rcl} \textit{communication-specification} & ::= & \textbf{comm}\ \textit{space-communication-declaration-list}. \\ \textit{communication-declaration} & ::= & \textit{name} \mid \textit{name} = \textit{name}. \end{array}$$

In the last rule the $\mid$ is a language symbol and should not be confused with the | used in sets and the BNF-syntax.

2.9 Specifications

Specifications are entities in which data, processes, actions etc. can be declared. The syntax of a *specification* is:

$$\begin{array}{rcl} \mathit{specification} & ::= & \mathit{sort\text{-}specification} \\ & | & \mathit{function\text{-}specification} \\ & | & \mathit{rewrite\text{-}specification} \\ & | & \mathit{action\text{-}specification} \\ & | & \mathit{communication\text{-}specification} \\ & | & \mathit{process\text{-}specification} \\ & | & \mathit{specification}\ \ \mathit{specification}. \end{array}$$

2.10 The standard sort Bool

In every *specification* the following function and sort declarations must be included. The reason for this special treatment of the sort **Bool** is that we want to guarantee that true and false as booleans are different. This can only be achieved if the names for true, false and the sort of booleans are predetermined.

$$\begin{array}{ll} \textbf{sort} & \textbf{Bool} \\ \textbf{func} & T :\rightarrow \textbf{Bool} \\ & F :\rightarrow \textbf{Bool} \end{array}$$

2.11 An example

As an example we give a *specification* of a data transfer process. *Data-elements* of sort D are transferred from *in* to *out*.

$$\begin{array}{ll} \textbf{sort} & \textbf{Bool} \\ \textbf{func} & T, F :\rightarrow \textbf{Bool} \\ \textbf{sort} & D \\ \textbf{func} & d1, d2, d3 :\rightarrow D \\ \textbf{act} & in, out : D \\ \textbf{proc} & TR = \sum(x : D, in(x) \cdot out(x) \cdot TR) \end{array}$$

2.12 The from construct

For a *process-expression* or a *data-term* t, we write t **from** E for a *specification* E where we mean the *process-expression* or *data-term* t as defined in E. Often, it is clear from the context to which *specification* E the item t belongs. In this case we generally write t without explicit reference to E.

3 Static semantics

Not every *specification* is necessarily correctly defined. It may be that objects are not declared, that they are declared at several places or are not used in a proper way. In this section we define under which circumstances a *specification* does not have these problems and hence has a correct *static semantics*. Furthermore, we define some functions that will be used in the definition of the semantics of μCRL.

3.1 The signature of a specification

The signature of a specification is an important ingredient in defining the static semantics. It consists of a five-tuple of which each component is a set containing all elements of a main syntactical category declared in a *specification* E.

Definition 3.1. Let E be a *specification*. The *signature* $Sig(E) = (Sort, Fun, Act, Comm, Proc)$ of E is defined as follows:

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

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

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

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

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

  $$Sig(E) \stackrel{\text{def}}{=} (Sort_1 \cup Sort_2, Fun_1 \cup Fun_2, Act_1 \cup Act_2, Comm_1 \cup Comm_2, Proc_1 \cup Proc_2).$$

Definition 3.2. Let $Sig = (Sort, Fun, Act, Comm, Proc)$ be a signature. We write

$$\begin{array}{l} Sig.Sort \text{ for } Sort, \\ Sig.Fun \text{ for } Fun, \\ Sig.Act \text{ for } Act, \\ Sig.Comm \text{ for } Comm, \\ Sig.Proc \text{ for } Proc. \end{array}$$

3.2 Variables

Variables play an important role in specifications. The next definition says which *names* can play the role of a variable without confusion with defined constants. Moreover, variables must have an unambiguous and declared sort.

Definition 3.3. Let *Sig* be a signature. A set $\mathcal{V}$ containing elements $\langle x : S\rangle$ with x and S *names*, is called a *set of variables* over *Sig* iff for each $\langle x : S\rangle \in \mathcal{V}$:

- for each *name* S' and *process-expression* p it holds that $x :\to S' \notin Sig.Fun$, $x \notin Sig.Act$ 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 3.4. Let *var-dec* be a *variable-declaration-section*. The function *Vars* is defined by:

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

In the following definitions we give functions yielding the sort and the variables in a *data-term* t. If for some reason no answer can be obtained, for instance because an undeclared *name* appears in t, a $\perp$ results. Of course this only works properly if $\perp$ does not occur in *names*.

Definition 3.5. Let t be a *data-term* and *Sig* a signature. Let $\mathcal{V}$ be a set of variables over *Sig*. We define:

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

Definition 3.6. Let *Sig* be a signature, $\mathcal{V}$ a set of variables over *Sig* and let t be a *data-term*.

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

We call a *data-term* t *closed* w.r.t. a signature *Sig* and a set of variables $\mathcal{V}$ iff $Var_{Sig,\mathcal{V}}(t) = \emptyset$. Note that $Var_{Sig,\mathcal{V}}(t) \subseteq \mathcal{V} \cup \{\perp\}$ for any *data-term* t.

3.3 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 rewrite rules satisfy a usual condition that the variables that are used at the right hand side of a equality sign must also occur at the left hand side. Because all these properties can be statically decided, a *specification* that is internally consistent is called SSC (*Statically Semantically Correct*). For a better understanding of the next definition, it may be helpful to read definition 3.8 first.

Definition 3.7. Let Sig be a signature and $\mathcal{V}$ be a set of variables over Sig. We define the predicate 'is SSC w.r.t. Sig' inductively over the syntax of a *specification*.

- A *specification* **sort** $n_1 \ ... \ n_m$ with $m \geq 1$ is SSC w.r.t. Sig iff all *names* $n_1, ..., n_m$ are pairwise different.

- A *specification* **func** $n_{11}, ..., n_{1l_1} : S_{11} \times ... \times S_{1k_1} \to S_1$
  $\vdots$
  $n_{m1}, ..., n_{ml_m} : S_{m1} \times ... \times S_{mk_m} \to S_m$

  with $m \geq 1$, $l_i \geq 1$, $k_i \geq 0$ for $1 \leq i \leq m$ is SSC w.r.t. Sig iff

  – for all $1 \leq i \leq m$ the *names* $n_{i1}, ..., n_{il_i}$ 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-section* and *rew-rul* is a *rewrite-rules-section* is SSC w.r.t. Sig iff

  – *var-dec* is SSC w.r.t. Sig,

  – *rew-rul* is SSC w.r.t. Sig and $Vars(var\text{-}dec)$.

⋆ The empty *variable-declaration-section* is SSC w.r.t. Sig.

  A *variable-declaration-section* **var** $n_{11}, ..., n_{1k_1} : S_1$
  $\vdots$
  $n_{m1}, ..., n_{mk_m} : S_m$

  with $m \geq 1$, $k_i \geq 1$ for $1 \leq i \leq m$ is SSC w.r.t. Sig iff

  – $n_{ij} \not\equiv n_{i'j'}$ 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 $\mathit{Vars}(\mathbf{var}\ n_{11},...,n_{1k_1} : S_1\ ...\ n_{m1},...,n_{mk_m} : S_m)$ is a set of variables over Sig.

⋆ A *rewrite-rules-section* $\mathbf{rew}\ rw_1\ ...\ rw_m$ with $m \geq 1$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

- if $rw_i \equiv n = t$ for some $1 \leq i \leq m$, then
  - $n :\rightarrow sort_{\mathit{Sig},\emptyset}(t) \in \mathit{Sig.Fun}$,
  - t is SSC w.r.t. Sig and $\emptyset$,
- if $rw_i \equiv n(t_1,...,t_{k_i}) = t$ for some $1 \leq i \leq m$ and $k_i \geq 1$, then
  - $n : sort_{\mathit{Sig},\mathcal{V}}(t_1) \times ... \times sort_{\mathit{Sig},\mathcal{V}}(t_{k_i}) \rightarrow sort_{\mathit{Sig},\mathcal{V}}(t) \in \mathit{Sig.Fun}$,
  - t, t_j $(1 \leq j \leq k_i)$ are SSC w.r.t. Sig and $\mathcal{V}$,
  - $\mathit{Var}_{\mathit{Sig},\mathcal{V}}(t) \subseteq \bigcup_{1 \leq j \leq k_i} \mathit{Var}_{\mathit{Sig},\mathcal{V}}(t_j)$.

⋆ A *data-term* n with n a *name* is SSC w.r.t. Sig and $\mathcal{V}$ iff $\langle n : S\rangle \in \mathcal{V}$ for some S, or $n :\rightarrow sort_{\mathit{Sig},\mathcal{V}}(n) \in \mathit{Sig.Fun}$.

A *data-term* $n(t_1,...,t_m)$ $(m \geq 1)$ is SSC w.r.t. Sig and $\mathcal{V}$ iff $n : sort_{\mathit{Sig},\mathcal{V}}(t_1) \times ... \times sort_{\mathit{Sig},\mathcal{V}}(t_m) \rightarrow sort_{\mathit{Sig},\mathcal{V}}(n(t_1,...,t_m)) \in \mathit{Sig.Fun}$ and all t_i $(1 \leq i \leq m)$ are SSC w.r.t. Sig and $\mathcal{V}$.

- A *specification* $\mathbf{act}\ ad_1\ ...\ ad_m$ with $m \geq 1$ is SSC w.r.t. Sig iff
  - for all $1 \leq i \leq m$ the *action-declaration* ad_i is SSC w.r.t. Sig,
  - for all $1 \leq i < j \leq m$ it holds that $\mathit{Sig}(\mathbf{act}\ ad_i).\mathit{Act} \cap \mathit{Sig}(\mathbf{act}\ ad_j).\mathit{Act} = \emptyset$.

⋆ An *action-declaration* n is SSC w.r.t. Sig iff for each *name* S' it holds that $n :\rightarrow S' \notin \mathit{Sig.Fun}$.

An *action-declaration* $n_1,...,n_m : S_1 \times ... \times S_k$ with $k, m \geq 1$ is SSC w.r.t. 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 $n_i : S_1 \times ... \times S_k \rightarrow S' \notin \mathit{Sig.Fun}$.

- A *specification* $\mathbf{comm}\ n_{11}|n_{12} = n_{13}\ ...\ n_{m1}|n_{m2} = n_{m3}$ with $m \geq 1$ is SSC w.r.t. 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 $n_{i1} \in \mathit{Sig.Act}$ or there is a $k \geq 1$ such that $n_{i1} : S_1 \times ... \times S_k \in \mathit{Sig.Act}$,
  - for each $1 \leq i \leq m$, $k \geq 1$ and *names* $S_1,...,S_k$ it holds that if $n_{i1} : S_1 \times ... \times S_k \in \mathit{Sig.Act}$ then $n_{i2} : S_1 \times ... \times S_k \in \mathit{Sig.Act}$ and $n_{i3} : S_1 \times ... \times S_k \in \mathit{Sig.Act}$,
  - for each $1 \leq i \leq m$, $k \geq 1$ and *names* $S_1,...,S_k$ it holds that if $n_{i2} : S_1 \times ... \times S_k \in \mathit{Sig.Act}$ then $n_{i1} : S_1 \times ... \times S_k \in \mathit{Sig.Act}$ and $n_{i3} : S_1 \times ... \times S_k \in \mathit{Sig.Act}$,

  - 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** $pd_1 \dots pd_m$ with $m \geq 1$ is SSC w.r.t. Sig iff
  - for each $1 \leq i < j \leq m$:
    - if $pd_i \equiv n_i = p_i$ and $pd_j \equiv n_j = p_j$ then $n_i \not\equiv n_j$,
    - if for some $k \geq 1$ it holds that $pd_i \equiv n_i(x_1 : S_1, \dots, x_k : S_k) = p_i$ and $pd_j \equiv n_j(x'_1 : S_1, \dots, x'_k : S_k) = p_j$ then $n_i \not\equiv n_j$,
    - for all *names* S' it holds that $n_i :\to S_i \notin Sig.Fun$,
  - if $pd_i \equiv n_i = p_i$ $(1 \leq i \leq m)$, then $n_i \notin Sig.Act$ and p_i is SSC w.r.t. Sig and $\emptyset$,
  - if $pd_i \equiv n_i(x_{i1} : S_{i1}, \dots, x_{ik_i} : S_{ik_i}) = p_i$ $(1 \leq i \leq m)$, then
    - $n_i : S_{i1} \times \dots \times S_{ik_i} \notin Sig.Act$,
    - for all *names* S' it holds that $n_i : S_{i1} \times \dots \times S_{ik_i} \to S' \notin Sig.Fun$,
    - the *names* $x_{i1}, \dots, x_{ik_i}$ are pairwise different and $\{\langle x_{ij} : S_{ij} \rangle \mid 1 \leq j \leq k_i\}$ is a set of variables over Sig,
    - p_i is SSC w.r.t. Sig and $\{\langle x_{ij} : S_{ij} \rangle \mid 1 \leq j \leq k_i\}$.

⋆ A *process-expression* $p_1 + p_2$, *parallel-expressions* $p_1 \parallel p_2$, $p_1 ⌊⌊ p_2$, $p_1 \mid p_2$, a *dot-expression* $p_1 \cdot p_2$ are SSC w.r.t. Sig and $\mathcal{V}$ iff

  - p_1 is SSC w.r.t. Sig and $\mathcal{V}$,
  - p_2 is SSC w.r.t. Sig and $\mathcal{V}$.

A *cond-expression* $p_1 \triangleleft t \triangleright p_2$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

  - p_1 is SSC w.r.t. Sig and $\mathcal{V}$,
  - p_2 is SSC w.r.t. Sig and $\mathcal{V}$,
  - t is SSC w.r.t. Sig and $\mathcal{V}$ and $sort_{Sig,\mathcal{V}}(t) = \mathbf{Bool}$.

The *basic-expressions* δ and τ are SSC w.r.t. Sig and $\mathcal{V}$.

The *basic-expressions* $\partial(\{n_1, \dots, n_m\}, p)$ and $\tau(\{n_1, \dots, n_m\}, p)$ with $m \geq 1$ are SSC w.r.t. Sig and $\mathcal{V}$ iff

  - 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 $n_i : S_1 \times \dots \times S_k \in Sig.Act$ for some $k \geq 1$ and *names* $S_1, \dots, S_k$,
  - p is SSC w.r.t. Sig and $\mathcal{V}$.

The *basic-expression* $\rho(\{n_1 \to n'_1, \dots, n_m \to n'_m\}, p)$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

  - for $1 \leq i \leq m$ either $n_i \in Sig.Act$ or $n_i : S_1 \times \dots \times S_k \in Sig.Act$ for some $k \geq 1$ and *names* $S_1, \dots, S_k$,

- 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 $n_i : S_1 \times ... \times S_k \in Sig.Act$, then also $n'_i : S_1 \times ... \times S_k \in Sig.Act$,
- for $1 \leq i \leq m$ it holds that if $n_i \in Sig.Act$, then also $n'_i \in Sig.Act$,
- p is SSC w.r.t. Sig and $\mathcal{V}$.

A *basic-expression* $\Sigma(x : S, p)$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

- $\mathcal{V} \backslash \{\langle x : S' \rangle \mid S' \text{ a } name\} \cup \{\langle x : S \rangle\}$ is a set of variables over Sig,
- p is SSC w.r.t. Sig and $\mathcal{V} \backslash \{\langle x : S' \rangle \mid S' \text{ a } name\} \cup \{\langle x : S \rangle\}$.

A *basic-expression* n is SSC w.r.t. Sig and $\mathcal{V}$ iff $n = p \in Sig.Proc$ for some *process-expression* p or $n \in Sig.Act$.

A *basic-expression* $n(t_1, ..., t_m)$ with $m \geq 1$ is SSC w.r.t. Sig and $\mathcal{V}$ iff

- $n(x_1 : sort_{Sig,\mathcal{V}}(t_1), ..., x_m : sort_{Sig,\mathcal{V}}(t_m)) = p \in Sig.Proc$ for some *names* $x_1, ..., x_m$ and *process-expression* p, or
  $n : sort_{Sig,\mathcal{V}}(t_1) \times ... \times sort_{Sig,\mathcal{V}}(t_m) \in Sig.Act$,
- for $1 \leq i \leq m$ the *data-term* t_i is SSC w.r.t. Sig and $\mathcal{V}$.

A *basic-expression* (p) is SSC w.r.t. Sig and $\mathcal{V}$ iff p is SSC w.r.t. Sig and $\mathcal{V}$.

- A *specification* $E_1\ E_2$ is SSC w.r.t. Sig iff
  - E_1 and E_2 are SSC w.r.t. Sig,
  - $Sig(E_1).Sort \cap Sig(E_2).Sort = \emptyset$,
  - if $n : S_1 \times ... \times S_m \to S \in Sig(E_1).Fun$ for some $m \geq 0$ then $n : S_1 \times ... \times S_m \to S' \notin Sig(E_2).Fun$ 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 $m \geq 1$ $pd_1 \equiv n_1(x_1 : S_1, ..., x_m : S_m) = p_1$ and $pd_2 \equiv n_2(x'_1 : S_1, ..., x'_m : S_m) = p_2$, then $n_1 \not\equiv n_2$.

Definition 3.8. Let E be a *specification*. We say that E is SSC iff E is SSC w.r.t. $Sig(E)$.

The following lemma is helpful in checking that the predicate 'is SSC' is correctly defined.

Lemma 3.9. *Let Sig be a signature and $\mathcal{V}$ be a set of variables over Sig. Let t be a data-term that is SSC w.r.t. Sig and $\mathcal{V}$. Then $sort_{Sig,\mathcal{V}}(t) \neq \perp$ and $\perp \notin \mathrm{Var}_{Sig,\mathcal{V}}(t)$.*

3.4 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 as is done in the syntax (cf. LOTOS [15] where this is not the case).

Definition 3.10. 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. We say that 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.

4 Well-formed μCRL specifications

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

Definition 4.1. Let E be a *specification* that is SSC. We say that E has *no empty sorts* iff for all $S \in Sig(E).Sort$ there is a *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$ such that $sort_{Sig(E),\emptyset}(t) \equiv S$.

Definition 4.2. Let E be a *specification*. E is called *well-formed* iff

- E is SSC,
- E is communication-associative,
- E has no empty sorts,
- $\mathbf{Bool} \in Sig(E).Sort$,
- $T :\rightarrow \mathbf{Bool} \in Sig(E).Fun$ and
- $F :\rightarrow \mathbf{Bool} \in Sig(E).Fun$.

5 Algebraic semantics

In this section we present the semantics of well-formed μCRL specifications. Given a signature Sig we introduce the class of Sig-algebras. Then 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 and in which the terms T and F of sort **Bool** are interpreted different. Then given such a model, we give an operational semantics for *process-expressions* in E.

5.1 Algebras

First we adapt the standard definitions of algebras etc. to μCRL (see e.g. [8] for these definitions).

Definition 5.1. Let E be a well-formed *specification*. A $Sig(E)$*-algebra* $\boldsymbol{A}$ is a structure containing

- for each $S \in Sig(E).Sort$ a non-empty domain $D(\boldsymbol{A}, S)$,
- for each $n :\to S \in Sig(E).Fun$ a constant $C(\boldsymbol{A}, n) \in D(\boldsymbol{A}, S)$,
- for each $n : S_1 \times ... \times S_m \to S \in Sig(E).Fun$ a function $F(\boldsymbol{A}, n : S_1 \times ... \times S_m)$ from $D(\boldsymbol{A}, S_1) \times ... \times D(\boldsymbol{A}, S_m)$ to $D(\boldsymbol{A}, S)$.

For two elements $a_1 \in D(\boldsymbol{A}, S_1)$ and $a_2 \in D(\boldsymbol{A}, S_2)$, we write $a_1 = a_2$ iff $S_1 \equiv S_2$ and a_1 and a_2 represent exactly the same element.

Definition 5.2. Let E be a well-formed *specification* and let $\boldsymbol{A}$ be a $Sig(E)$-algebra. We define the interpretation $[\![\cdot]\!]_{\boldsymbol{A}}$ from *data-terms* that are SSC w.r.t. $Sig(E)$ and $\emptyset$ into the domains of $\boldsymbol{A}$ as follows:

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

We say that a $Sig(E)$-algebra $\boldsymbol{A}$ is *minimal* iff for each $a \in D(\boldsymbol{A}, S)$ and $S \in Sig(E).Sort$, there is some *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$ such that $[\![t]\!]_{\boldsymbol{A}} = a$. For *data-terms* t_1, t_2 that are SSC w.r.t. $Sig(E)$ and $\emptyset$ we write $\boldsymbol{A} \models t_1 = t_2$ iff $[\![t_1]\!]_{\boldsymbol{A}} = [\![t_2]\!]_{\boldsymbol{A}}$.

Definition 5.3. Let E be a well-formed *specification* and let $\boldsymbol{A}$ be a minimal $Sig(E)$-algebra. A function r mapping pairs of a sort S and an element from $D(\boldsymbol{A}, S)$ to *data-terms* that are SSC w.r.t. to $Sig(E)$ and $\emptyset$ is called a *representation function* of E and $\boldsymbol{A}$ iff $\boldsymbol{A} \models t = r(sort_{Sig(E),\emptyset}(t), [\![t]\!]_{\boldsymbol{A}})$ for each *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$.

5.2 Substitutions

We define substitutions on *data-terms*. These substitutions are immediately extended to *process-expressions* because this is required for the definition of the operational semantics.

Definition 5.4. Let E be a well-formed *specification* and $\mathcal{V}$ a set of variables over $Sig(E)$. Let *Term* be the set of *data-terms* that are SSC w.r.t. $Sig(E)$ and $\mathcal{V}$. A *substitution* σ over $Sig(E)$ and $\mathcal{V}$ is a mapping

$$\sigma : \mathcal{V} \to Term$$

such that for each $\langle x : S \rangle \in \mathcal{V}$ it holds that $sort_{Sig(E),\mathcal{V}}(\sigma(\langle x : S \rangle)) = S$. Substitutions are extended to *data-terms* by:

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

Definition 5.5. Let E be a well-formed *specification* and $\mathcal{V}$ a set of variables over $Sig(E)$. Let σ be a substitution over $Sig(E)$ and $\mathcal{V}$. We extend σ to *process-expressions* that are SSC w.r.t. $Sig(E)$ and $\mathcal{V}$ as follows:

- If $p_1 \Box p_2$ is a *process-expression*, a *parallel-expression* or a *dot-expression* ($\Box \in \{+, \|, \lfloor\!\lfloor, |, \cdot\}$), then $\sigma(p_1 \Box p_2) \stackrel{\text{def}}{=} \sigma(p_1) \Box \sigma(p_2)$,
- $\sigma(p_1 \lhd t \rhd p_2) \stackrel{\text{def}}{=} \sigma(p_1) \lhd \sigma(t) \rhd \sigma(p_2)$ for a *cond-expression* $p_1 \lhd t \rhd p_2$,
- $\sigma(\delta) \stackrel{\text{def}}{=} \delta$ and $\sigma(\tau) \stackrel{\text{def}}{=} \tau$ for *basic-expressions* δ and τ,
- if $\Box(gl, p)$ is a *basic-expression* ($\Box \in \{\partial, \tau, \rho\}$), then $\sigma(\Box(gl, p)) \stackrel{\text{def}}{=} \Box(gl, \sigma(p))$,
- $\sigma(\Sigma(x : S, p)) \stackrel{\text{def}}{=} \Sigma(x : S, \sigma'(p))$ where σ' is defined by

$$\sigma'(\langle x' : S' \rangle) \stackrel{\text{def}}{=} \begin{cases} \langle x : S \rangle & \text{if } x' \equiv x \\ \sigma(\langle x' : S' \rangle) & \text{otherwise,} \end{cases}$$

  for a *basic-expression* $\Sigma(x : S, p)$,
- $\sigma(n(t_1, ..., t_m)) \stackrel{\text{def}}{=} n(\sigma(t_1), ..., \sigma(t_m))$ for a *basic-expression* $n(t_1, ..., t_m)$,
- $\sigma(n) \stackrel{\text{def}}{=} n$ for a *basic-expression* n,
- $\sigma((p)) \stackrel{\text{def}}{=} (\sigma(p))$ for a *basic-expression* (p).

The validity of the following lemma gives us confidence that substitutions are indeed correctly defined.

Lemma 5.6. *Let E be a well-formed specification and $\mathcal{V}$ a set of variables over $Sig(E)$. Let σ be a substitution over $Sig(E)$ and $\mathcal{V}$.*

- *For any data-term t that is SSC w.r.t. $Sig(E)$ and $\mathcal{V}$, $\sigma(t)$ is also a data-term that is SSC w.r.t. $Sig(E)$ and $\mathcal{V}$. Moreover, $sort_{Sig(E),\mathcal{V}}(t) \equiv sort_{Sig(E),\mathcal{V}}(\sigma(t))$.*
- *For any process-expression p that is SSC w.r.t. $Sig(E)$ and $\mathcal{V}$, $\sigma(p)$ is a process-expression that is SSC w.r.t. $Sig(E)$ and $\mathcal{V}$.*

5.3 Boolean preserving models

A $Sig(E)$-algebra $\boldsymbol{A}$ is a model of a well-formed *specification* E iff the equations defining the data in E hold in $\boldsymbol{A}$. Moreover, we say that $\boldsymbol{A}$ is *boolean preserving* iff T and F of sort **Bool** represent exactly the two different elements of $D(\boldsymbol{A}, \mathbf{Bool})$. Note that there are specifications which have no boolean preserving models of E, for instance a specification containing the equation $T = F$. For μCRL we are only interested in the minimal $Sig(E)$-algebras that are boolean preserving.

First we define the function *rewrites* that extracts the rewrite clauses together with declared variables from a *specification*.

Definition 5.7. We define the function *rewrites* on a *specification* E inductively as follows:

- If $E \equiv$ *sort-spec* with *sort-spec* a *sort-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.
- If $E \equiv$ *func-spec* with *func-spec* a *function-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.
- If $E \equiv V\ R$ with V a *variable-declaration-section* and R a *rewrite-rules-section* with $R \equiv \textbf{rew}\ rd_1\ ...\ rd_m$ for some $m \geq 1$, then
$$rewrites(E) \stackrel{\text{def}}{=} \{\langle\{rd_i \mid 1 \leq i \leq m\}, \mathit{Vars}(V)\rangle\}.$$
- If $E \equiv$ *act-spec* with *act-spec* an *action-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.
- If $E \equiv$ *comm-spec* with *comm-spec* a *communication-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.
- If $E \equiv$ *proc-spec* with *proc-spec* a *process-specification*, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.
- If $E \equiv E_1\ E_2$ where E_1 and E_2 are *specifications*, then $rewrites(E) \stackrel{\text{def}}{=} rewrites(E_1) \cup rewrites(E_2)$.

Definition 5.8. Let E be a well-formed *specification*. A $Sig(E)$-algebra $\boldsymbol{A}$ is a *model* of E, notation $\boldsymbol{A} \models_D E$, iff whenever $t = t' \in R$ with $\langle R, \mathcal{V}\rangle \in rewrites(E)$, then for any substitution σ over $Sig(E)$ and $\mathcal{V}$ such that $Var_{Sig(E),\mathcal{V}}(\sigma(t)) = Var_{Sig(E),\mathcal{V}}(\sigma(t')) = \emptyset$ it holds that $\boldsymbol{A} \models \sigma(t) = \sigma(t')$.

We write $\boldsymbol{A} \models_D E$ with a subscript D because the model only concerns the data in E.

Definition 5.9. Let E be a well-formed *specification*. A $Sig(E)$-algebra $\boldsymbol{A}$ is called *boolean preserving* w.r.t. E iff

- it is not the case that $\boldsymbol{A} \models T = F$,
- $|D(\boldsymbol{A}, \mathbf{Bool})| = 2$, i.e. T and F are exactly the two elements of sort **Bool**.

5.4 The process part

In this section we define for each *process-expression* p that is SSC w.r.t. $Sig(E)$ and $\emptyset$, and each minimal model $\boldsymbol{A}$ of E that preserves the booleans and where E is some well-formed *specification*, a meaning in terms of a referential transition system (cf. the operational semantics in [2, 21, 22]).

Definition 5.10. A transition system $\mathcal{A}$ is a quadruple $(S, L, \longrightarrow, s)$ where

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

Elements $(s', l, s'') \in \longrightarrow$ are generally written as $s' \xrightarrow{l} s''$.

Definition 5.11. Let E be a well-formed *specification*, $\boldsymbol{A}$ be a minimal model of E that is boolean preserving and r be a representation function of E and $\boldsymbol{A}$. Let p be a *process-expression* that is SSC w.r.t. $Sig(E)$ and $\emptyset$. The *meaning* of p **from** E in $\boldsymbol{A}$ with representation function r is the *referential* transition system $\mathcal{A}(\boldsymbol{A}, r, p \textbf{ from } E)$ defined by

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

where

- $S \stackrel{\text{def}}{=} \{q \mid \text{where } q \text{ is a } \textit{process-expression} \text{ that is SSC w.r.t. } Sig(E) \text{ and } \emptyset\} \cup \{\surd\}$,
- $L \stackrel{\text{def}}{=} \{n(t_1, ..., t_m) \mid m \geq 0, n \in Sig(E).Act$ and for $1 \leq i \leq m$ it holds that $t_i \equiv r(S_i, a)$ for some $a \in D(\boldsymbol{A}, S_i)$ where $S_i \equiv sort_{Sig(E),\emptyset}(t_i)\} \cup \{\tau, \surd\}$,
- $s \stackrel{\text{def}}{=} p$,
- $\longrightarrow$ is the transition relation that contains exactly all transitions provable using the rules below (see for provability e.g. [9]). Let p, p', q, q' range over the set $S \setminus \{\surd\}$, P is a *process-expression* that is SSC w.r.t. $Sig(E)$ and some set of variables over $Sig(E)$, l ranges over the set L of labels, n, n_1, n_2 are *names*, $m \geq 0$ and $t_1, ..., t_m, u_1, ..., u_m$ are *data-terms* (note that there is no rule for δ):
  - $\surd \xrightarrow{\surd} \delta$.
  - $\tau \xrightarrow{\tau} \surd$.
  - $n \xrightarrow{n()} \surd$ if $n \in Sig(E).Act$,
  - $n(u_1, ..., u_m) \xrightarrow{n(t_1,...,t_m)} \surd$ with $m \geq 1$ if
    - $n : sort_{Sig(E),\emptyset}(u_1) \times ... \times sort_{Sig(E),\emptyset}(u_m) \in Sig(E).Act$,

  - $t_i \equiv r(sort_{Sig(E),\emptyset}(u_i), [\![u_i]\!]_{\boldsymbol{A}})$.

- $\dfrac{p \xrightarrow{l} p'}{n \xrightarrow{l} p'}$ if $n = p \in Sig(E).Proc$,
  - $\dfrac{p \xrightarrow{l} \surd}{n \xrightarrow{l} \surd}$ if $n = p \in Sig(E).Proc$,
  - $\dfrac{\sigma(P) \xrightarrow{l} p'}{n(u_1, ..., u_m) \xrightarrow{l} p'}$ with $m \geq 1$ if
    - $n(x_1 : sort_{Sig(E),\emptyset}(u_1), ..., x_m : sort_{Sig(E),\emptyset}(u_m)) = P \in Sig(E).Proc$,
    - there is a substitution σ over $Sig(E)$ and $\{\langle x_1 : sort_{Sig(E),\emptyset}(u_1)\rangle, ..., \langle x_m : sort_{Sig(E),\emptyset}(u_m)\rangle\}$ such that $\sigma(\langle x_i : sort_{Sig(E),\emptyset}(u_i)\rangle) \equiv u_i$ for $1 \leq i \leq m$,
  - $\dfrac{\sigma(P) \xrightarrow{l} \surd}{n(u_1, ..., u_m) \xrightarrow{l} \surd}$ with $m \geq 1$ if
    - $n(x_1 : sort_{Sig(E),\emptyset}(u_1), ..., x_m : sort_{Sig(E),\emptyset}(u_m)) = P \in Sig(E).Proc$,
    - there is a substitution σ over $Sig(E)$ and $\{\langle x_1 : sort_{Sig(E),\emptyset}(u_1)\rangle, ..., \langle x_m : sort_{Sig(E),\emptyset}(u_m)\rangle\}$ such that $\sigma(\langle x_i : sort_{Sig(E),\emptyset}(u_i)\rangle) \equiv u_i$ for $1 \leq i \leq m$.
- $\dfrac{p \xrightarrow{l} p'}{p + q \xrightarrow{l} p'}$,
  - $\dfrac{p \xrightarrow{l} \surd}{p + q \xrightarrow{l} \surd}$,
  - $\dfrac{q \xrightarrow{l} q'}{p + q \xrightarrow{l} q'}$,
  - $\dfrac{q \xrightarrow{l} \surd}{p + q \xrightarrow{l} \surd}$.
- $\dfrac{p \xrightarrow{l} p'}{p \cdot q \xrightarrow{l} p' \cdot q}$,
  - $\dfrac{p \xrightarrow{l} \surd}{p \cdot q \xrightarrow{l} q}$.
- $\dfrac{p \xrightarrow{l} p'}{p \lhd t \rhd q \xrightarrow{l} p'}$ if $\boldsymbol{A} \models t = T$,
  - $\dfrac{p \xrightarrow{l} \surd}{p \lhd t \rhd q \xrightarrow{l} \surd}$ if $\boldsymbol{A} \models t = T$,

- $\dfrac{q \xrightarrow{l} q'}{p \triangleleft t \triangleright q \xrightarrow{l} q'}$ if $\boldsymbol{A} \models t = F$,
- $\dfrac{q \xrightarrow{l} \surd}{p \triangleleft t \triangleright q \xrightarrow{l} \surd}$ if $\boldsymbol{A} \models t = F$.

• $\dfrac{p \xrightarrow{l} p'}{p \parallel q \xrightarrow{l} p' \parallel q}$,

- $\dfrac{q \xrightarrow{l} q'}{p \parallel q \xrightarrow{l} p \parallel q'}$,
- $\dfrac{p \xrightarrow{l} \surd}{p \parallel q \xrightarrow{l} q}$,
- $\dfrac{q \xrightarrow{l} \surd}{p \parallel q \xrightarrow{l} p}$,
- $\dfrac{p \xrightarrow{n_1(t_1,\ldots,t_m)} p' \quad q \xrightarrow{n_2(t_1,\ldots,t_m)} q'}{p \parallel q \xrightarrow{n(t_1,\ldots,t_m)} p' \parallel q'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,
- $\dfrac{p \xrightarrow{n_1(t_1,\ldots,t_m)} \surd \quad q \xrightarrow{n_2(t_1,\ldots,t_m)} q'}{p \parallel q \xrightarrow{n(t_1,\ldots,t_m)} q'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,
- $\dfrac{p \xrightarrow{n_1(t_1,\ldots,t_m)} p' \quad q \xrightarrow{n_2(t_1,\ldots,t_m)} \surd}{p \parallel q \xrightarrow{n(t_1,\ldots,t_m)} p'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,
- $\dfrac{p \xrightarrow{n_1(t_1,\ldots,t_m)} \surd \quad q \xrightarrow{n_2(t_1,\ldots,t_m)} \surd}{p \parallel q \xrightarrow{n(t_1,\ldots,t_m)} \surd}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$.

• $\dfrac{p \xrightarrow{l} p'}{p \mathbin{\underline{\|}} q \xrightarrow{l} p' \parallel q}$,

- $\dfrac{p \xrightarrow{l} \surd}{p \mathbin{\underline{\|}} q \xrightarrow{l} q}$.

• $\dfrac{p \xrightarrow{n_1(t_1,\ldots,t_m)} p' \quad q \xrightarrow{n_2(t_1,\ldots,t_m)} q'}{p \mid q \xrightarrow{n(t_1,\ldots,t_m)} p' \parallel q'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,

- $\dfrac{p \xrightarrow{n_1(t_1,\ldots,t_m)} \surd \quad q \xrightarrow{n_2(t_1,\ldots,t_m)} q'}{p \mid q \xrightarrow{n(t_1,\ldots,t_m)} q'}$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,

- $$\frac{p \xrightarrow{n_1(t_1,...,t_m)} p' \quad q \xrightarrow{n_2(t_1,...,t_m)} \surd}{p \mid q \xrightarrow{n(t_1,...,t_m)} p'}$$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$,
- $$\frac{p \xrightarrow{n_1(t_1,...,t_m)} \surd \quad q \xrightarrow{n_2(t_1,...,t_m)} \surd}{p \mid q \xrightarrow{n(t_1,...,t_m)} \surd}$$ if $n_1 \mid n_2 = n \in Sig(E).Comm^*$.

* $$\frac{p \xrightarrow{l} p'}{\tau(\{n_1,...,n_k\},p) \xrightarrow{l} \tau(\{n_1,...,n_k\},p')}$$
  if $l \equiv n(t_1,...,t_m)$ and $n \not\equiv n_i$ for all $1 \le i \le k$, or $l \equiv \tau$,
  - $$\frac{p \xrightarrow{l} \surd}{\tau(\{n_1,...,n_k\},p) \xrightarrow{l} \surd}$$
    if $l \equiv n(t_1,...,t_m)$ and $n \not\equiv n_i$ for all $1 \le i \le k$, or $l \equiv \tau$,
  - $$\frac{p \xrightarrow{n(t_1,...,t_m)} p'}{\tau(\{n_1,...,n_k\},p) \xrightarrow{\tau} \tau(\{n_1,...,n_k\},p')}$$ if $n \equiv n_i$ for some $1 \le i \le k$,
  - $$\frac{p \xrightarrow{n(t_1,...,t_m)} \surd}{\tau(\{n_1,...,n_k\},p) \xrightarrow{\tau} \surd}$$ if $n \equiv n_i$ for some $1 \le i \le k$.

* $$\frac{p \xrightarrow{l} p'}{\rho(\{n_1 \to n_1',...,n_k \to n_k'\},p) \xrightarrow{l} \rho(\{n_1 \to n_1',...,n_k \to n_k'\},p')}$$
  if $l \equiv n(t_1,...,t_m)$ and $n \not\equiv n_i$ for all $1 \le i \le k$, or $l \equiv \tau$,
  - $$\frac{p \xrightarrow{l} \surd}{\rho(\{n_1 \to n_1',...,n_k \to n_k'\},p) \xrightarrow{l} \surd}$$
    if $l \equiv n(t_1,...,t_m)$ and $n \not\equiv n_i$ for all $1 \le i \le k$, or $l \equiv \tau$,
  - $$\frac{p \xrightarrow{n(t_1,...,t_m)} p'}{\rho(\{n_1 \to n_1',...,n_k \to n_k'\},p) \xrightarrow{n'(t_1,...,t_m)} \rho(\{n_1 \to n_1',...,n_k \to n_k'\},p')}$$
    if $n \equiv n_i$ and $n' \equiv n_i'$ for some $1 \le i \le k$,
  - $$\frac{p \xrightarrow{n(t_1,...,t_m)} \surd}{\rho(\{n_1 \to n_1',...,n_k \to n_k'\},p) \xrightarrow{n'(t_1,...,t_m)} \surd}$$
    if $n \equiv n_i$ and $n' \equiv n_i'$ for some $1 \le i \le k$.

* $$\frac{p \xrightarrow{l} p'}{\partial(\{n_1,...,n_k\},p) \xrightarrow{l} \partial(\{n_1,...,n_k\},p')}$$
  if $l \equiv n(t_1,...,t_m)$ and $n \not\equiv n_i$ for all $1 \le i \le k$, or $l \equiv \tau$,

- $$\frac{p \xrightarrow{l} \surd}{\partial(\{n_1, ..., n_k\}, p) \xrightarrow{l} \surd}$$
  if $l \equiv n(t_1, ..., t_m)$ and $n \not\equiv n_i$ for all $1 \leq i \leq k$, or $l \equiv \tau$.

- $$\frac{\sigma(P) \xrightarrow{l} p'}{\Sigma(x : S, P) \xrightarrow{l} p'}$$
  where σ is a substitution over $Sig(E)$ and $\{\langle x : S\rangle\}$ such that $\sigma(\langle x : S\rangle) = t$ for some *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$,

- $$\frac{\sigma(P) \xrightarrow{l} \surd}{\Sigma(x : S, P) \xrightarrow{l} \surd}$$
  where σ is a substitution over $Sig(E)$ and $\{\langle x : S\rangle\}$ such that $\sigma(\langle x : S\rangle) = t$ for some *data-term* t that is SSC w.r.t. $Sig(E)$ and $\emptyset$.

According to the convention in 2.12 we often write $\mathcal{A}(\boldsymbol{A}, r, p)$ instead of $\mathcal{A}(\boldsymbol{A}, r, p$ **from** $E)$. Again, the following lemma serves as a justification for our definition.

Lemma 5.12. *Let E be a well-formed specification, $\boldsymbol{A}$ be a minimal model of E that is boolean preserving and r a representation function of E and $\boldsymbol{A}$. Consider a process-expression p that is SSC w.r.t. $Sig(E)$ and $\emptyset$ and let $(S, L, \longrightarrow, s) \stackrel{\text{def}}{=} \mathcal{A}(\boldsymbol{A}, r, p)$. If for some sequence of labels $l_1, ..., l_m$ it holds that $p \xrightarrow{l_1} \; ... \; \xrightarrow{l_m} p'$, then either $p' \equiv \surd$ or p' is SSC w.r.t. $Sig(E)$ and $\emptyset$.*

We feel that our operational semantics is somewhat ad hoc; we can easily provide an alternative that is also satisfactory in the sense that for each *process-expression* the generated transition system is strongly bisimilar with that generated by the rules above. Therefore, we generally consider transition systems modulo strong bisimulation equivalence. This means that the operational semantics for μCRL as given in this document has only a *referential* meaning, and any generated transition system is therefore called a *referential transition system*. A consequence of this view is that for the generation of transition systems for a μCRL-*process-expression* an operational semantics generating a smaller number of states can be used.

Definition 5.13. Let $\mathcal{A}_1 = (S_1, L_1, \longrightarrow_1, s_1)$ and $\mathcal{A}_2 = (S_2, L_2, \longrightarrow_2, s_2)$ be two transition systems. We say that $\mathcal{A}_1$ and $\mathcal{A}_2$ are bisimilar, notation $\mathcal{A}_1 \leftrightarrow \mathcal{A}_2$, iff there is a relation $R \subseteq S_1 \times S_2$ such that

- $(s_1, s_2) \in R$,
- for each pair $(t_1, t_2) \in R$:
  - $t_1 \xrightarrow{a}_1 t_1' \Rightarrow \exists t_2' \; t_2 \xrightarrow{a}_2 t_2'$ and $(t_1', t_2') \in R$,
  - $t_2 \xrightarrow{a}_2 t_2' \Rightarrow \exists t_1' \; t_1 \xrightarrow{a}_1 t_1'$ and $(t_1', t_2') \in R$.

Let E be a well-formed *specification*, $\boldsymbol{A}$ a minimal boolean preserving model of E, and r a representation function of E and $\boldsymbol{A}$. For two μCRL-*process-expressions* p and q that are SSC w.r.t. $Sig(E)$ and $\emptyset$, we write

$$p \textbf{ from } E \leftrightarrows_{\boldsymbol{A},r} q \textbf{ from } E$$

iff $\mathcal{A}(\boldsymbol{A}, r, p \textbf{ from } E) \leftrightarrows \mathcal{A}(\boldsymbol{A}, r, q \textbf{ from } E)$.

The following lemma allows us to write $\leftrightarrows_{\boldsymbol{A}}$ instead of $\leftrightarrows_{\boldsymbol{A},r}$. Moreover, it gives us a useful property of bisimulation, i.e. that it is a congruence for all process operators. Note that according to our own convention we do not explicitly say where p and q stem from as they can only come from E.

Lemma 5.14. *Let E be a specification, $\boldsymbol{A}$ a minimal, boolean preserving model of E and p, q process-expressions that are SSC w.r.t. E and $\emptyset$.*

- *If $p \leftrightarrows_{\boldsymbol{A},r} q$ for some representation function r of E and $\boldsymbol{A}$, then $p \leftrightarrows_{\boldsymbol{A},r'} q$ for each representation function r' of E and $\boldsymbol{A}$.*

- *For all representation functions of E and $\boldsymbol{A}$, $\leftrightarrows_{\boldsymbol{A},r}$ is a congruence for all μCRL operators working on process-expressions.*

6 Effective μCRL-specifications

In order to provide a process language with tools, such as for instance a simulator, it is very important that the language has a computable operational semantics, i.e. it is decidable what the next (finite number of) steps of a process are. This is not at all the case for μCRL. Due to the undecidability of data equivalence, the use of possibly unguarded recursion and infinite sums, the next step relation need not be enumerable. We deal with this situation by restricting μCRL to *effective* μCRL. In effective μCRL data equivalence is decidable, only finite sums are allowed and recursion must be guarded. For effective μCRL the next step relation is indeed decidable.

6.1 Semi complete rewriting systems

For the data we require that the rewriting system is semi-complete (= weakly terminating and confluent) [16]. This implies that data equivalence between closed terms is decidable. Moreover, this is (in some sense) not too restrictive: every data type for which data equivalence is decidable, can be specified by a complete (= strongly terminating and confluent) term rewriting system [5]. As a complete term rewriting system is also semi-complete, all decidable data types can be expressed in effective μCRL.

We first define all required rewrite relations.

Definition 6.1. Let E be a well-formed *specification*. We define the *elementary rewrite relation* $\longrightarrow_E^e$ by:

$$\begin{aligned}\longrightarrow_E^e \stackrel{\text{def}}{=} \; & \{\sigma(u) \longrightarrow \sigma(u') \mid \\ & u = u' \in R \text{ with } \langle R, \mathcal{V}\rangle \in \mathit{rewrites}(E), \\ & \sigma \text{ is a substitution over } \mathit{Sig}(E) \text{ and } \mathcal{V} \text{ such that } \mathit{Var}_{\mathit{Sig}(E),\mathcal{V}}(\sigma(u)) = \emptyset\}.\end{aligned}$$

The one-step reduction relation $\longrightarrow_E$ is inductively defined by:

- $u \longrightarrow u' \in \longrightarrow_E$ if $u \longrightarrow u' \in \longrightarrow_E^e$.
- $n(t_1, ..., t_m) \longrightarrow n(t_1', ..., t_m') \in \longrightarrow_E$ if for some $1 \leq i \leq m$
  - $t_i \longrightarrow t_i' \in \longrightarrow_E$,
  - for $j \neq i$ it holds that $t_j \equiv t_j'$ and $n(t_1, ..., t_m)$ is SSC w.r.t. $\mathit{Sig}(E)$ and $\emptyset$.

The *reduction relation* $\twoheadrightarrow_E$ is the reflexive and transitive closure of $\longrightarrow_E$. We write $t \longrightarrow_E u$ and $t \twoheadrightarrow_E u$ for $t \longrightarrow u \in \longrightarrow_E$ and $t \twoheadrightarrow u \in \twoheadrightarrow_E$, respectively.

The following lemma is meant to reassure ourselves that the definitions of the rewrite relations are correct. Moreover, it gives a basic but useful property.

Lemma 6.2. *Let E be a well-formed specification. Let t be a data-term that is SSC w.r.t. $\mathit{Sig}(E)$ and $\emptyset$. If $t \twoheadrightarrow_E t'$, then t' is also SSC w.r.t. $\mathit{Sig}(E)$ and $\emptyset$.*

With these rewrite relations it is easy to define confluence and termination.

Definition 6.3. Let E be a well-formed *specification*. E is *data-confluent* iff for *data-terms* t, t' and t'' that are SSC w.r.t. $\mathit{Sig}(E)$ and $\emptyset$ it holds that:

$$\left.\begin{array}{l} t \twoheadrightarrow_E t' \\ t \twoheadrightarrow_E t'' \end{array}\right\} \text{ implies that there is a } \mathit{data\text{-}term}\ t''' \text{ such that } \left\{\begin{array}{l} t' \twoheadrightarrow_E t''' \\ t'' \twoheadrightarrow_E t'''. \end{array}\right.$$

A *data-term* t that is SSC w.r.t. $\mathit{Sig}(E)$ and $\emptyset$ is a *normal form* if for no *data-term* u it holds that $t \longrightarrow_E u$. E is *data-terminating* if for each *data-term* t that is SSC w.r.t. $\mathit{Sig}(E)$ and $\emptyset$ there is some normal form t'' such that $t \twoheadrightarrow_E t''$. E is *data-semi-complete* if E is data-confluent and data-terminating.

The following lemma states that in μCRL we can find a unique normal form for each *data-term* that can be obtained from a well-formed *specification*.

Lemma 6.4. *Let E be a well-formed specification that is data-semi-complete. For any data-term t that is SSC with respect to $\mathit{Sig}(E)$ and $\emptyset$, there is a unique data-term $N_E(t)$ satisfying*

$$t \twoheadrightarrow_E N_E(t) \text{ and } N_E(t) \text{ is a normal form.}$$

$N_E(t)$ is called the normal form of t and there is an algorithm to find $N_E(t)$ for each data-term t that is SSC w.r.t. $\mathit{Sig}(E)$ and $\emptyset$.

Effective μCRL is based on the following algebra of normal forms.

Definition 6.5. Let E be a well-formed *specification* that is data-semi-complete. The $Sig(E)$-algebra $\boldsymbol{A}_{N_E}$ of normal forms is defined by:

- for each name $S \in Sig(E).Sort$ there is a domain $D(\boldsymbol{A}_{N_E}, S) \stackrel{\text{def}}{=} \{N_E(t) \mid sort_{Sig(E),\emptyset}(t) = S$ and t is a *data-term* that is SSC w.r.t. $Sig(E)$ and $\emptyset\}$,
- $C(\boldsymbol{A}_{N_E}, n) \stackrel{\text{def}}{=} N_E(n)$ provided $n :\rightarrow S \in Sig(E).Fun$,
- $F(\boldsymbol{A}_{N_E}, n : S_1 \times ... \times S_m) = f$ where the function f is defined by:

  $$f(t_1, ..., t_m) = N_E(n(t_1, ..., t_m))$$

  with $t_i \in D(\boldsymbol{A}_{N_E}, S_i)$ for $1 \leq i \leq m$ provided $n : S_1 \times ... \times S_m \rightarrow S \in Sig(E).Fun$.

Note that in $\boldsymbol{A}_{N_E}$ it is easy to determine that $T \neq F$. It is however undecidable that the sort **Bool** has at most two elements. We must use the *finite sort tool* of section 6.5 to determine this. Often the algebra $\boldsymbol{A}_{N_E}$ is called the *canonical term algebra* of E.

6.2 Finite sums

If a μCRL specification contains infinite sums, then the operational behaviour is not finitely branching anymore. Consider for instance the behaviour of the following process:

$$\begin{array}{lll} X \ \textbf{from} & \textbf{sort} & \textbf{Bool} \\ & \textbf{func} & T, F :\rightarrow \textbf{Bool} \\ & \textbf{sort} & Nat \\ & \textbf{func} & 0 : Nat \\ & & succ : Nat \rightarrow Nat \\ & \textbf{act} & a : Nat \\ & \textbf{proc} & X = \sum(x : Nat, a(x)) \end{array}$$

The process X can perform an $a(m)$ step for each natural number m. We judge an infinitely branching operational behaviour undesirable and therefore exclude sums over infinite sorts from effective μCRL.

Definition 6.6. Let E be a well-formed *specification* and let $\boldsymbol{A}$ be a model of E. We say that E has *finite sums* w.r.t. $\boldsymbol{A}$ iff for each occurrence $\Sigma(x : S, p)$ in E the set $D(\boldsymbol{A}, S)$ is finite.

6.3 Guarded recursive specifications

Also unguarded recursion may lead to an infinitely branching operational behaviour. Consider for instance the following example:

$$\begin{array}{lll} X \ \textbf{from} & \textbf{sort} & \textbf{Bool} \\ & \textbf{func} & T, F :\rightarrow \textbf{Bool} \\ & \textbf{act} & a \\ & \textbf{proc} & X = X \cdot a + a \end{array}$$

The *process-expression* $X \cdot a$ can perform an a step to any *process-expression* a^m $(m \geq 1)$ where a^m is the sequential composition of m a's. Therefore, we also exclude unguarded recursion from effective μCRL.

In the next definition it is said what a guarded μCRL specification is in very general terms.

Definition 6.7. Let E be a well-formed *specification* and $\boldsymbol{A}$ be a model of E that is boolean preserving. Let p be a *process-expression* of the form n or $n(t_1, ..., t_m)$ for some *name* n that is SSC w.r.t. $Sig(E)$ and $\emptyset$. Let q be a *process-expression* that is SSC w.r.t. $Sig(E)$ and $\emptyset$. We say that p is *guarded* w.r.t. $\boldsymbol{A}$ in q iff

- $q \equiv q_1 + q_2$, $q \equiv q_1 \parallel q_2$ or $q \equiv q_1 \mid q_2$, and p is guarded w.r.t. $\boldsymbol{A}$ in q_1 and q_2,
- $q \equiv q_1 \triangleleft c \triangleright q_2$ and either $\boldsymbol{A} \models c = T$ and p is guarded w.r.t. $\boldsymbol{A}$ in q_1, or $\boldsymbol{A} \models c = F$ and p is guarded w.r.t. $\boldsymbol{A}$ in q_2,
- $q \equiv q_1 \cdot q_2$, $q \equiv q_1 \mathbin{\|\!\_} q_2$, $q \equiv \partial(\{n_1, ..., n_m\}, q_1)$, $q \equiv \tau(\{n_1, ..., n_m\}, q_1)$, $q \equiv \rho(\{n_1 \to n'_1, ..., n_m \to n'_m\}, q_1)$ or $q \equiv (q_1)$ and p is guarded w.r.t. $\boldsymbol{A}$ in q_1,
- $q \equiv \Sigma(x : S, q_1)$ and p is guarded w.r.t. $\boldsymbol{A}$ in $\sigma(q_1)$ for any substitution σ over $Sig(E)$ and $\{\langle x : S \rangle\}$,
- $q \equiv \tau$ or $q \equiv \delta$,
- $q \equiv n'$ for a *name* n' and $p \not\equiv n'$ or
- $q \equiv n'(u_1, ..., u_{m'})$ for a *basic-expression* $n'(u_1, ..., u_{m'})$ and $n \not\equiv n'$, $m \neq m'$ or $[\![u_i]\!]_{\boldsymbol{A}} \neq [\![t_i]\!]_{\boldsymbol{A}}$ for some $1 \leq i \leq m$.

If p is not guarded w.r.t. $\boldsymbol{A}$ in q we say that p appears *unguarded* w.r.t. $\boldsymbol{A}$ in q.

Definition 6.8. Let E be a well-formed *specification* and $\boldsymbol{A}$ be a model of E that is boolean preserving. The *Process Name Dependency Graph* of E and $\boldsymbol{A}$, notation $PNDG(E, \boldsymbol{A})$, is constructed as follows:

- for each $n = p \in Sig(E).Proc$, n is a node of $PNDG(E, \boldsymbol{A})$,
- for each $n(x_1 : S_1, ..., x_m : S_m) = p \in Sig(E).Proc$ and *data-terms* $t_1, ..., t_m$ that are SSC w.r.t. $Sig(E)$ and $\emptyset$ such that $sort_{Sig(E),\emptyset}(t_i) = S_i$ $(1 \leq i \leq m)$, $n(t_1, ..., t_m)$ is a node of $PNDG(E, \boldsymbol{A})$,
- if n is a node of $PNDG(E, \boldsymbol{A})$ and $n = p \in Sig(E).Proc$, then there is an edge

  $$n \longrightarrow q$$

  for a node $q \in PNDG(E, \boldsymbol{A})$ iff q is unguarded w.r.t. $\boldsymbol{A}$ in p,
- if $n(x_1 : sort_{Sig(E),\emptyset}(t_1), ..., x_m : sort_{Sig(E),\emptyset}(t_m)) = p \in Sig(E).Proc$ and $n(t_1, ..., t_m)$ is a node of $PNDG(E, \boldsymbol{A})$, then there is an edge

  $$n(t_1, ..., t_m) \longrightarrow q$$

for a node $q \in PNDG(E, \boldsymbol{A})$ iff q is unguarded w.r.t. $\boldsymbol{A}$ in $\sigma(p)$ where σ is the substitution over $Sig(E)$ and $\{\langle x_i : sort_{Sig(E),\emptyset}(t_i)\rangle \mid 1 \leq i \leq m\}$ defined by

$$\sigma(\langle x_i : sort_{Sig(E),\emptyset}(t_i)\rangle) = t_i.$$

Definition 6.9. Let E be a well-formed *specification* and $\boldsymbol{A}$ be a model of E that is boolean preserving. We say that E is *guarded* w.r.t. $\boldsymbol{A}$ iff $PNDG(E, \boldsymbol{A})$ is well founded, i.e. does not contain an infinite path.

6.4 Effective μCRL-specifications

Here we define the operational semantics of effective μCRL by combining all definitions given above.

Definition 6.10. Let E be a *specification*. We call E an *effective* μCRL *specification* or for short an *effective specification* iff

- E is well-formed,
- E is data-semi-complete,
- E has finite sums w.r.t. $\boldsymbol{A}_{N_E}$,
- E is guarded w.r.t. $\boldsymbol{A}_{N_E}$.

Definition 6.11. Let E be an effective μCRL *specification*. Let p be a *process-expression* that is SSC w.r.t. $Sig(E)$ and $\emptyset$. The behaviour of p is the transition system

$$\mathcal{A}(\boldsymbol{A}_{N_E}, r, p \textbf{ from } E)$$

where the representation function r of E and $\boldsymbol{A}_{N_E}$ is the identity.

In effective μCRL data equivalence is indeed decidable and the operational behaviour is finitely branching and computable:

Theorem 6.12. *Let E be an effective μCRL specification and let $(S, L, \longrightarrow, s) = \mathcal{A}(\boldsymbol{A}_{N_E}, r, p)$ for some data-term p that is SSC w.r.t. $Sig(E)$ and $\emptyset$ and let r be the identity. Then*

- *for each pair of data-terms t_1, t_2 that are SSC w.r.t. $Sig(E)$ and $\emptyset$:*

$$t_1 =_E t_2 \text{ is decidable},$$

- *for each process-expression p' that is SSC w.r.t. $Sig(E)$ and $\emptyset$:*

$$\{\langle a, p''\rangle \mid p' \xrightarrow{a} p''\}$$

is finite and effectively computable. Moreover, its cardinality is also effectively computable from E and p.

The second point of the previous theorem says that $\mathcal{A}(\boldsymbol{A}_{N_E}, r, p \textbf{ from } E)$ is a *computable* transition system. In a recursion theoretic setting a *computable* transition system is defined as follows: let $\mathcal{A} = (S, L, \longrightarrow, s_0)$ be a transition system with S and L sets of natural numbers and $s_0 \in S$ is represented by 0. We say that $\mathcal{A}$ is a *computable* transition system iff $\longrightarrow$ is represented by a *total* recursive function ϕ that maps each number in S to (a coding of) a finite set of pairs $\{\langle l, s' \rangle \mid s \xrightarrow{l} s'\}$.

6.5 Proving μCRL-specifications effective

In general it is not decidable whether a μCRL *specification* is effective. But there are many tools available that can prove the effectiveness for quite large classes of *specifications*. These tools provide, given a specification, a 'yes' or a 'don't know' answer.

Definition 6.13. Let $\mathcal{E}$ be the set of all well-formed *specifications*. A *data-semi-completeness tool*, notation DC, a *finite-sort tool*, notation FS, and a *guardedness tool*, notation GD, are all decidable predicates over $\mathcal{E}$, i.e. $DC \subseteq \mathcal{E}, FS \subseteq \mathcal{N} \times \mathcal{E}, GD \subseteq \mathcal{E}$.

A tool is called *sound* if each claim of a certain property it makes about a well-formed *specification* is correct. In the definition of a sound finite-sort tool and a sound guardedness tool we assume that specifications are data-semi-complete because we expect that this is a minimal requirement for these tools to operate.

Definition 6.14. A data-semi-completeness tool DC is called *sound* iff for each *specification* E that is well-formed:

if $DC(E)$ holds, then E is data-semi-complete.

A finite-sort tool FS is called *sound* iff for each *name* n and *specification* E that is well-formed and data-semi-complete:

if $FS(n, E)$ holds, then $n \in Sig(E).Sort$ and $D(\boldsymbol{A}_{N_E}, n)$ is a finite set.

A guardedness tool GD is called *sound* iff for each *specification* E that is well-formed and data-semi-complete:

if $GD(E)$ holds, then E is guarded w.r.t. $\boldsymbol{A}_{N_E}$.

Sometimes a tool needs auxiliary information per *specification* to perform its task. In this case such a tool may work on a tuple containing a specification and a finite amount of such information. There is no prescribed format for this information, and it may vary from tool to tool. If a tool requires auxiliary information, then the soundness of the tool may not depend on this information. In this case the definition of soundness is modified as follows (the definition is only given for DC, the other cases can be defined likewise):

Definition 6.15. A data-semi-completeness tool DC requiring auxiliary information, is called *sound* iff for each well-formed *specification* E and each instance of auxiliary information $\mathcal{I}$:

if $DC(E,\mathcal{I})$ holds, then E is data-semi-complete.

This definition guarantees that even with incorrect auxiliary information DC always produces correct answers. DC has to be *robust*.

Below we describe some techniques for constructing sound tools, except in those cases where techniques are provided in the literature. As time proceeds, more and more powerful techniques will appear. In order to incorporate these technological advancements in μCRL, the techniques mentioned here are only possible candidates for sound tools. They may be replaced by others, as long as these also lead to sound tools.

There are many techniques for proving termination and confluence (see HUET and OPPEN [14] and DERSHOWITZ [7] for termination, NEWMAN [20] for confluence if termination has been shown and KLOP [16] for an overview). Therefore we will not go into details here.

The problem whether a sort has a finite number of elements [4] is undecidable and as far as we know no general techniques have been developed to prove that a sort has only a finite number of elements in a minimal algebra.

We present a possible approach that can only be applied to a restricted case: let E be a *specification* in $\mathcal{E}$ such that $DC(E)$ for some sound data-semi-completeness tool DC and assume that we are interested in the finiteness of sorts $S_1, ..., S_k$ occurring in E. Let F be the set of all functions specified in E that have as target sort one of the sorts S_i $(1 \leq i \leq k)$. We assume that their parameter sorts also originate from $S_1, ..., S_k$. As auxiliary information we use finite sets $\mathcal{I}_i$ of (closed) *data-terms* that ought to represent all elements of sort S_i.

We compute for each function $f \in F$ (with target sort S_j) and for all arguments in the sets $\mathcal{I}_i$ of appropriate sorts, whether application of f leads to a *data-term* equivalent to one of the elements of $\mathcal{I}_j$. This can be done as we assume that $DC(E)$ holds. If this is successful, then obviously the sorts $S_1, ..., S_k$ have a finite number of elements.

Also the question whether a *specification* is guarded is undecidable. Still very good results can be obtained when guardedness is checked abstracting from the data parameters of process names. This is done by the following function HV. Its first argument contains the *process-expression* that is being searched for unguarded occurrences of *names* of processes and its second argument guarantees that the bodies of *process-declarations* are not searched twice.

Definition 6.16. Let E be a well-formed *specification* and let $\mathcal{V}$ be a set of variables over $Sig(E)$. A *process-type* is an expression $\langle n : S_1 \times ... \times S_m \rangle$ for some $m \geq 0$ with n a *name* and $S_1, ..., S_m$ *names*. The function HV maps pairs of a *process-expression* and a set of *process-types* to sets of *process-types*.

- $HV(\delta, PT) \stackrel{\text{def}}{=} \emptyset$.
- $HV(p_1 + p_2, PT) = HV(p_1 \lhd c \rhd p_2, PT) = HV(p_1 \parallel p_2, PT) = HV(p_1 \mid p_2, PT) \stackrel{\text{def}}{=} HV(p_1, PT) \cup HV(p_2, PT)$.
- $HV(p_1 \cdot p_2, PT) = HV(p_1 \mathbin{\lfloor\!\lfloor} p_2, PT) = HV(\partial(\{n_1, ..., n_m\}, p_1), PT) = HV(\tau(\{n_1, ..., n_m\}, p_1), PT) = HV(\rho(\{n_1 \to n'_1, ..., n_m \to n'_m\}, p_1), PT) = HV(\Sigma(x : S, p_1), PT) \stackrel{\text{def}}{=} HV(p_1, PT)$.
- $HV(n(t_1, ..., t_m), PT) \stackrel{\text{def}}{=}$

  - $\{\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle\}$
    if $\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle \in PT$.
  - $HV(p, PT \cup \{\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle\}) \cup$
    $\{\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle\}$
    if $\langle n : sort_{Sig(E),\mathcal{V}}(t_1) \times ... \times sort_{Sig(E),\mathcal{V}}(t_m)\rangle \notin PT$ and $n(x_1 : sort_{Sig(E),\mathcal{V}}(t_1), ..., x_m : sort_{Sig(E),\mathcal{V}}(t_m)) = p \in Sig(E).Proc$ for some *names* $x_1, ..., x_m$.

- $HV(n, PT) \stackrel{\text{def}}{=}$
  - $\{\langle n :\rangle\}$ if $\langle n :\rangle \in PT$,
  - $HV(p, PT \cup \{\langle n :\rangle\}) \cup \{\langle n :\rangle\}$ if $\langle n :\rangle \notin PT$ and $n = p \in Sig(E).Proc$.

- $HV((p), PT) \stackrel{\text{def}}{=} HV(p, PT)$.

Theorem 6.17. *Let E be a well-formed specification. If for each process-declaration $n(x_1 : S_1, ..., x_m : S_m) = p \in Sig(E).Proc$ it holds that $\langle n : S_1 \times ... \times S_m\rangle \notin HV(p, \emptyset)$ and for each process-declaration $n = p \in Sig(E).Proc$ $n \notin HV(p, \emptyset)$, then E is guarded.*

Appendix An SDF-syntax for μCRL

We present an SDF-syntax for μCRL [10] which serves two purposes. It provides a syntax that does not employ special characters and, using it as input for the ASF+SDF-system, it yields an interactive editor for μCRL-specifications (see eg. [11]). The ASF+SDF system is also used to provide a well-formedness checker [17].

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.

In this syntax *names* are arbitrary strings over `a-z`, `A-Z` and `0-9` except that keywords are not *names*. 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 *names* separated by commas.

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. Instead of δ, ∂, τ and ρ we write `delta`, `encap`, `tau`, `hide` and `rename`. These keywords are taken from PSF [18].

The priorities say that ‘`.`’ has highest and `+` has lowest priority on *process-expressions*.

```
exports
 sorts Name
       Name-list
```

```
        X-name-list
        Space-name-list
        Sort-specification
        Function-specification
        Function-declaration
        Rewrite-specification
        Variable-declaration-section
        Variable-declaration
        Data-term
        Rewrite-rules-section
        Rewrite-rule
        Process-expression
        Renaming-declaration
        Single-variable-declaration
        Process-specification
        Process-declaration
        Action-specification
        Action-declaration
        Communication-specification
        Communication-declaration
        Specification

lexical syntax
        [ \t\n]                                       -> LAYOUT
        "%" ~[\n]* "\n"                               -> LAYOUT
        [a-zA-Z0-9]*                                  -> Name
context-free syntax
        { Name ","}+                                  -> Name-list
        { Name "#"}+                                  -> X-name-list
          Name+                                       -> Space-name-list
        sort Space-name-list                          -> Sort-specification
        func Function-declaration+                    -> Function-specification
        Name-list ":" X-name-list "->" Name           -> Function-declaration
        Name-list ":" "->" Name                       -> Function-declaration

        Variable-declaration-section
                Rewrite-rules-section                 -> Rewrite-specification
        var Variable-declaration+                     -> Variable-declaration-section
                                                      -> Variable-declaration-section
        Name-list ":" Name                            -> Variable-declaration
        Name                                          -> Data-term
        Name "(" { Data-term "," }+ ")"               -> Data-term
        rew Rewrite-rule+                             -> Rewrite-rules-section
        Name "(" { Data-term "," }+ ")" "=" Data-term -> Rewrite-rule
        Name "=" Data-term                            -> Rewrite-rule

        Process-expression "+" Process-expression     -> Process-expression right
        Process-expression "||" Process-expression    -> Process-expression right
        Process-expression "||_" Process-expression   -> Process-expression
        Process-expression "|"  Process-expression    -> Process-expression right
        Process-expression "<|" Data-term "|>"
```

```
            Process-expression               -> Process-expression
Process-expression "." Process-expression    -> Process-expression right
delta                                        -> Process-expression
tau                                          -> Process-expression
encap "(" "{" Name-list "}" ","
            Process-expression ")"           -> Process-expression
hide "(" "{" Name-list "}" ","
            Process-expression ")"           -> Process-expression
rename "(" "{" { Renaming-declaration "," }+
          "}" "," Process-expression ")"     -> Process-expression
sum "(" Single-variable-declaration ","
               Process-expression ")"        -> Process-expression
Name "(" { Data-term "," }+ ")"              -> Process-expression
Name                                         -> Process-expression
"(" Process-expression ")"                   -> Process-expression bracket

Name "->" Name                               -> Renaming-declaration
Name ":" Name                                -> Single-variable-declaration
proc Process-declaration+                    -> Process-specification
Name "(" { Single-variable-declaration "," }+ ")"
            "=" Process-expression           -> Process-declaration
Name "=" Process-expression                  -> Process-declaration

act Action-declaration+                      -> Action-specification
Name-list ":" X-name-list                    -> Action-declaration
Name                                         -> Action-declaration

comm Communication-declaration+              -> Communication-specification
Name "|" Name "=" Name                       -> Communication-declaration

Sort-specification                           -> Specification
Function-specification                       -> Specification
Rewrite-specification                        -> Specification
Action-specification                         -> Specification
Communication-specification                  -> Specification
Process-specification                        -> Specification
Specification Specification                  -> Specification  assoc

priorities
       "+" < { "||", "|", "||_"} < "<|" "|>" < "."
```

As an example we provide a μCRL-specification of an alternating bit protocol. This is almost exactly the protocol as described in [2] to which we also refer for an explanation.

```
sort    Bool
func    T,F:->Bool

sort    D
func    d1,d2,d3 : -> D

sort    error
```

```
func    e        : -> error

sort    bit
func    0,1      : -> bit
        invert   : bit -> bit

rew     invert(1)=0
        invert(0)=1

act     r1,s4    : D
        s2,r2,c2 : D#bit
        s3,r3,c3 : D#bit
        s3,r3,c3 : error
        s5,r5,c5 : bit
        s6,r6,c6 : bit
        s6,r6,c6 : error

comm    r2|s2 = c2
        r3|s3 = c3
        r5|s5 = c5
        r6|s6 = c6

proc    S             = S(0).S(1).S
        S(n:bit)      = sum(d:D,r1(d).S(d,n))
        S(d:D,n:bit) = s2(d,n).(r6(invert(n))+r6(e)).S(d,n)+r6(n)

        R             = R(1).R(0).R
        R(n:bit)      = (sum(d:D,r3(d,n))+r3(e)).s5(n).R(n)+
                         sum(d:D,r3(d,invert(n)).s4(d).s5(invert(n)))

        K             = sum(d:D,sum(n:bit,r2(d,n).(tau.s3(d,n)+tau.s3(e)))).K
        L             = sum(n:bit,r5(n).(tau.s6(n)+tau.s6(e))).L

        ABP           = hide({c2,c3,c5,c6},encap({r2,r3,r5,r6,s2,s3,s5,s6},S||R||K||L))
```

References

[1] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. Report P9008, University of Amsterdam, Amsterdam, 1990.

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

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

[4] J.A. Bergstra and J.V. Tucker. A characterisation of computable data types by means of a finite equational specification method. In J.W. de Bakker and J. van Leeuwen, editors, *Proceedings, 1980*, volume 85 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.

[5] J.A. Bergstra and J.V. Tucker. The completeness of the algebraic specification methods for computable data types. *Information and Control*, 12:186–200, 1982.

[6] CCITT Working Party X/1. *Recommendation Z.100 (SDL)*, 1987.

[7] N. Dershowitz. Computing with rewrite systems. *Information and Control*, 65:122–157, 1985.

[8] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications I*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[9] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings* 16^{th} *ICALP,* Stresa, volume 372 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 1989. Full version to appear in *Information and Computation*.

[10] 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.

[11] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991. To appear.

[12] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, 1985.

[13] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[14] G. Huet and D.D. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.

[15] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[16] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, volume 1. Oxford University Press, 1990. To appear.

[17] H. Korver. Private communications, 1991.

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

[19] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[20] M.H.A. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[21] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts – II*, Garmisch, pages 199–225, Amsterdam, 1983. North-Holland.

[22] SPECS-semantics. *Definition of MR and CRL Version 2.1*, 1990.