Dogfooding the Structural Operational Semantics of mCRL2

F.P.M. Stappers[1], M.A. Reniers[2], J.F. Groote[1], and S. Weber[3]

[1]*Dept. of Mathematics and Computer Science,*
Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands,
{f.p.m.stappers, j.f.groote@tue.nl}@tue.nl
[2]*Dept. of Mechanical Engineering,*
Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands,
m.a.reniers@tue.nl
[3]*Dept. for Architecture and Platform, ASML,*
P.O. Box 324, NL-5500 AH Veldhoven, The Netherlands
s.weber@asml.com

December 9, 2011

Abstract

The mCRL2 language is a formal specification language that is used to specify and model the behavior of distributed systems and protocols. With the accompanying toolset, it is possible to simulate, visualize, analyze and verify behavioral properties of mCRL2 models automatically. The semantics of the mCRL2 language is defined formally using Structural Operational Semantics (SOS) but implemented manually in the underlying toolset using C++. Like with most formal languages, the underlying toolset was created with the formal semantics in mind but there is no way to actually guarantee that the implementation matches the intended semantics.

To validate that the implemented behavior for the mCRL2 language corresponds to its formal semantics, we describe the SOS deduction rules of the mCRL2 language, and perform the transformation from the mCRL2's SOS deduction rules to a Linear Process Specification. As our transformation directly takes the SOS deduction rules and transforms them into mCRL2 data equations, we are basically feeding the mCRL2 toolset its own formal language definition.

This report describes (i) the semantics for the untimed fragment of the mCRL2 language, (ii) the transformation of the deduction rules into data equations including the underlying design decisions and (iii) the experiments that have been conducted with our semantic transformation.

Despite its formal characterization, thorough study and broad use in many areas, our semantic dogfooding approach revealed a number of (subtle) differences between the mCRL2's intended semantics, the defined semantics and its actual implementation.

Contents

Chapter 1

Introduction

To validate the behavior of software *and* to experience its usage, software engineering deploys a technique that has become known as "eating your own dogfood" [20] or *dogfooding* for short. When dogfooding, a company uses its own products in their development process to develop new, or extend existing, products. Dogfooding is appealing as it is a simple and practical way to validate and improve the quality of software products as developers *really use* rather than *just test* the products. Google, Microsoft and the Eclipse community provide successful examples of industrial application of this technique [10, 19, 25, 32].

Theoretical software engineers and scientists also tend to eat their own dogfood to a limited extend. Examples from the field of formal specification, simulation and (formal) behavioral analysis can be found in languages such as χ [2], CIF [3], POOSL [42], Uppaal [4], Promela/SPIN [22], CSP and FDR2 [14], CADP [12], and mCRL2 [16]. Here, the formal specification languages, and the toolsets that implement these languages, are actively used by their respective research groups to model and analyze the behavior of various systems.

In general, the aforementioned specification languages have their semantics *defined formally* as rules written in some paper, chapter or book. The corresponding formal semantics for a simulator, model-checker or theorem prover are *implemented manually* in programming languages like C, C++, Java or Lisp. Although these manual implementations are written and reviewed with great care, they may still contain subtle, yet crucial deviations from their intended formal semantics. To the best of our knowledge, we have never seen that the semantics of these specification languages have been simulated or model checked by the language itself.

In this report we show how to effectively analyze the gap between the formal semantics, given in Structural Operational Semantics (SOS) [31], and the corresponding manual implementation, by dogfooding the formal definition of a specification language to our own model checker.

The language that we consider is an extended subset of the mCRL2 language [16] and the used model checker is part of the mCRL2 toolset [16, 40]. The strength of the mCRL2 language, includes sets, quantification, higher-order functions, which are essential to perform the transformation. To this end, we have extended the formal framework as described in [37], by taking the Transition System Specification (TSS) [7] of the mCRL2 language, along with a concrete specification (i.e., a model), and transform them into a Linear Process

Specification (LPS) [6, 13], which is a restricted mCRL2 specification. Since we specify these transformation rules directly in the mCRL2 language, the underlying model-checker is forced to eat, rewrite and execute its own formal language definition. Next to the transformation, we also show the design decisions taken as well as the merits and drawbacks of dogfooding a formal language like mCRL2.

In all, the approach captures the entire untimed semantics of the mCRL2 language in (roughly) 1000 lines of mCRL2 code, for which we are able to simulate, generate and verify properties on models. Moreover, the semantic dogfooding approach also shows that — despite the formal characterization of the mCRL2 language, its application in many areas, a committed team of developers, and the thorough study and broad usage by many students and researchers — a number of (subtle) differences exist between the mCRL2's intended semantics, the defined semantics and the actual implementation. Based on the work performed in this report, these semantic issues have been discussed and are currently being resolved.

This report is outlined as follows. Chapter 2 describes the preliminaries, which include a short introduction to Structural Operational Semantics, the formal operational semantics of the mCRL2 language, and the formal definition of a Linear Process Specification. We outline our approach in Chapter 3. Chapter 4 discusses a number of design decisions for the implementation. Chapter 5 describes the mCRL2 data equations that model the SOS deduction rules. In Chapter 6 we illustrate some of the examples that have been actually used to dogfood the Structural Operational Semantics of the mCRL2 language. Chapter 7 outlines the restrictions that come with both our framework and approach as outlined in this report. We evaluate our approach in Chapter 8 and present related work in Chapter 9. Finally, we conclude our work and provide directions for future research in Chapter 10.

Chapter 2

Preliminaries

2.1 Structural Operational Semantics

Structural Operational Semantics (SOS) defines the possible actions that a piece of syntax can perform. SOS is typically represented by a Transition System Specification (TSS) [7, 18]. The syntax for which the semantics is defined, is represented by a signature. A signature fixes the composition operators and their corresponding arities, where a function with arity zero represents a constant. We start with a set of variables $\mathcal{V}_{\mathrm{SOS}}$ and a set of action labels $\mathcal{A}_{\mathrm{SOS}}$. The signature used within this report is defined by a multi-sorted signature. The transition relation is defined by a single sorted relation.

Definition 2.1.1 (Signature, Terms, Transitions).
A *signature* Σ_{SOS} consists of

- a collection $\mathcal{S}_{\mathrm{SOS}}$ of sorts represented by $S, S_1, \ldots, S_n$,
- a collection of function symbols together with their arities. Let f be a function symbol of sort S. Then the arity of a function symbol is denoted by $ar(f)$, such that $S_1 \times \ldots \times S_{ar(f)} \to S$ defines the sorts of the arguments. Note that $S_1 \times \ldots \times S_{ar(f)}$ may be empty.

The collection of *terms* over signature Σ_{SOS}, denoted $\mathcal{T}(\Sigma_{\mathrm{SOS}})$, is the smallest set such that

- a variable $x_S \in \mathcal{V}^S_{\mathrm{SOS}}$ is a term of sort S, and
- $f(t_1, \ldots, t_n)$ is a term of sort S, if $t_1, \ldots, t_n$ are terms, where t_i is of sort S_i and $f \in \Sigma_{\mathrm{SOS}}$ is an n-ary function symbol of sort $S_1 \times \ldots \times S_n \to S$.

The set of *closed terms* over signature Σ_{SOS}, denoted $\mathcal{C}(\Sigma_{\mathrm{SOS}})$, is the set of all terms over Σ_{SOS} in which no variables occur. The *variables* that occur in a term p are denoted by $vars(p)$. A *valuation* σ is a collection of sort preserving functions from variables of sort S to values of the same sort S. A *transition formula* is of the form $(p, \sigma) \xrightarrow{l} (p', \sigma')$ for $p, p' \in \mathcal{T}(\Sigma_{\mathrm{SOS}})$, $l \in \mathcal{A}_{\mathrm{SOS}}$ and σ, σ' being data valuations.

Definition 2.1.2 (Transition System Specification).
A *Transition System Specification* (TSS) is a tuple $(\Sigma_{\mathrm{SOS}}, \mathcal{D}_{\mathrm{SOS}})$ where Σ_{SOS} is a signature and $\mathcal{D}_{\mathrm{SOS}}$ is a set of deduction rules. A deduction rule is of the form $\frac{H}{C}$ where H is a set of transition formulas, called the set of *premises*, and C is a transition formula, called the *conclusion*.

2.2 The mCRL2 language

mCRL2 [15, 16, 17] is a formal specification language with an associated toolset. The toolset [16, 40] can be used for the specification, the validation and the verification of concurrent systems and protocols. This section describes the syntactic elements of the mCRL2 language in BNF notation and the associated formal semantics in SOS.

In this report we restrict ourselves to the untimed fragment of the mCRL2 language. The semantics for the timed fragment describes time over a dense domain. This causes computational problems during the analysis in the current toolset. Therefore this fragment is removed from this report. In itself, incorporating time in our interpretation framework would not pose a problem as we will illustrate in Chapter 7. Apart from considering the untimed fragment of the language, we do consider an extension of the deduction rules as described in [16]. That is, we include a data valuation σ that is used to represent local variables and their values. This section introduces the untimed fragment of the mCRL2 language as used throughout this report.

2.2.1 Syntactic concepts

An mCRL2 specification consists of a data specification part and a process specification part.

Data specification part. We assume that the set of sorts $\mathcal{S}^{\mathrm{mCRL2}}$, the set of constructors $\mathcal{C}^{\mathrm{mCRL2}}$ and a set of mappings $\mathcal{M}^{\mathrm{mCRL2}}$, are available and are well-typed.

Definition 2.2.1 (Signature).
Let $\mathcal{S}^{\mathrm{mCRL2}}$ be a set of sorts, $\mathcal{C}^{\mathrm{mCRL2}}_{\mathcal{S}}$ a set of function symbols over $\mathcal{S}^{\mathrm{mCRL2}}$ called constructors, and $\mathcal{M}^{\mathrm{mCRL2}}_{\mathcal{S}}$ a set of function symbols over $\mathcal{S}^{\mathrm{mCRL2}}$ called mappings. We call the triple $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ a signature.

Definition 2.2.2 (Constructor sort).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ be a signature. Sort $S \in \mathcal{S}^{\mathrm{mCRL2}}$ is a *constructor sort* if there exists a constructor function declaration $f : S_1 \times \ldots \times S_n \to S \in \mathcal{C}^{\mathrm{mCRL2}}$.

Definition 2.2.3 (Function sort).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ be a signature. Sort $S \in \mathcal{S}^{\mathrm{mCRL2}}$ is a *function sort* if there exists a mapping $f : S_1 \times \ldots \times S_n \to S \in \mathcal{M}^{\mathrm{mCRL2}}$.

Definition 2.2.4 (Well-typed signature).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ be as signature. Then Σ^{mCRL2} is well-typed iff:

- $\mathcal{C}_{\mathcal{S}}^{\mathrm{mCRL2}} \cap \mathcal{M}_{\mathcal{S}}^{\mathrm{mCRL2}} = \emptyset$
- $\mathbb{B}$ is a sort, with exactly the constructors *true*:$\mathbb{B}$ and *false*:$\mathbb{B}$
- Constructor sorts are *syntactically non empty*, i.e., a sort D is called *syntactically non empty* iff there is a constructor $f{:}D_1 \times \ldots \times D_n \to D \in \mathcal{C}_{\mathcal{S}}^{\mathrm{mCRL2}} (n \geq 0)$ such that for all $1 \leq i \leq n$ if D_i is a constructor sort, D_i is also syntactically non empty.

Definition 2.2.5 (Data expressions).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}_{\mathcal{S}}^{\mathrm{mCRL2}}, \mathcal{M}_{\mathcal{S}}^{\mathrm{mCRL2}})$ be a signature, and let $\mathcal{X}_{\mathcal{S}}^{\mathrm{mCRL2}}$ be a set of $\mathcal{S}^{\mathrm{mCRL2}}$-typed variable symbols. We inductively define typed data expressions (over $\mathcal{X}_{\mathcal{S}}^{\mathrm{mCRL2}}$) as follows:

- every variable symbol $x{:}D \in \mathcal{X}_{\mathcal{S}}^{\mathrm{mCRL2}}$ is a data expression of sort D.
- every function symbol $f{:}D \in \mathcal{C}_{\mathcal{S}}^{\mathrm{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\mathrm{mCRL2}}$ is a data expression of sort D.
- Let p be a data expression of sort $D_1 \times \ldots \times D_n \to D$ and for $1 \leq i \leq n$ let p_i be data expressions of sort D_i, then $p(p_1, \ldots, p_n)$ is a data expression of sort D.
- For $1 \leq i \leq n$ let $x_i{:}D_i \notin (\mathcal{C}_{\mathcal{S}}^{\mathrm{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\mathrm{mCRL2}})$, and let p be a data expression of sort D over $\mathcal{X}_{\mathcal{S}}^{\mathrm{mCRL2}} \cup \{x_i{:}D_i | 1 \leq i \leq n\}$, then $\lambda x_1{:}D_1, \ldots, x_n{:}D_n.p$ is a data expression of sort $D_1 \times \ldots \times D_n \to D$.
- For $1 \leq i \leq n$ let $x_i{:}D_i \notin (\mathcal{C}_{\mathcal{S}}^{\mathrm{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\mathrm{mCRL2}})$, and let p be a data expression of sort $\mathbb{B}$ over $\mathcal{X}_{\mathcal{S}}^{\mathrm{mCRL2}} \cup \{x{:}D\}$, then $\exists x{:}D.p$ is a data expression of sort $\mathbb{B}$.
- For $1 \leq i \leq n$ let $x_i{:}D_i \notin (\mathcal{C}_{\mathcal{S}}^{\mathrm{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\mathrm{mCRL2}})$, and let p be a data expression of sort $\mathbb{B}$ over $\mathcal{X}_{\mathcal{S}}^{\mathrm{mCRL2}} \cup \{x{:}D\}$, then $\forall x{:}D.p$ is a data expression of sort $\mathbb{B}$.
- For $1 \leq i \leq n$ let p_i be data expressions of sorts D_i, $x_i{:}D_i \notin (\mathcal{C}_{\mathcal{S}}^{\mathrm{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\mathrm{mCRL2}})$, and let p be a data expression of sort D over $\mathcal{X}_{\mathcal{S}}^{\mathrm{mCRL2}} \cup \{x_i{:}D_i | 1 \leq i \leq n\}$ then p **whr** $x_1 = p_1, \ldots, x_n = p_n$ **end** is a data expression of sort D.

Definition 2.2.6 (Data specification).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ be a well-typed signature. Then the tuple $\mathcal{D}^{\mathrm{mCRL2}} = (\Sigma^{\mathrm{mCRL2}}, E)$ is a *data specification*, such that E is a set of *conditional equations*. Each equation in E is a pair $\langle \mathcal{X}^{\mathrm{mCRL2}}, c \to p_1 = p_2 \rangle$. Here $\mathcal{X}^{\mathrm{mCRL2}}$ is a set of variable declarations and $c{:}\mathbb{B}$, $p_1{:}D$ and $p_2{:}D$ are data expressions, where $D \in \mathcal{S}^{\mathrm{mCRL2}}$.

Process Specification Part. We assume that a set of action labels $\mathcal{A}^{\mathrm{mCRL2}}$ is available.

Definition 2.2.7 (Action declaration).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ be a signature, let $\mathcal{A}^{\mathrm{mCRL2}}$ be a set of action labels, and for $1 \leq i \leq n$ let $D_i \in \mathcal{S}^{\mathrm{mCRL2}}$, then an *action declaration* is a set of expressions of the form $a{:}D_1 \times \ldots \times D_n$.

An action with action label a and data expressions $d_1, \ldots, d_n$, denoted $\overrightarrow{d}$, is written as $a(\overrightarrow{d})$. Actions may be declared without any sorts, denoting actions without any data parameters. These actions may be written without brackets as a, so without $(\overrightarrow{d})$.

All the actions that are specified inside an *mCRL2 specification* (Definition 2.2.11) must be declared by an *action declaration*. We assume that all actions that occur in a *process expression* (Definition 2.2.10) are declared.

Note that data parameters in the syntactic (multi-)actions are data expressions and that the data parameters in the semantic (multi-)actions are values. The set of action labels is shared among the syntactic and semantic (multi-)actions.

Definition 2.2.8 (Syntactic multi-action).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ be a signature and let $\mathcal{A}^{\mathrm{mCRL2}}$ be a set of action labels. A syntactic multi-action represents a collection of actions that are specified to occur at the same time instant. Syntactic multi-actions have the following BNF grammar:

$$\alpha ::= \tau \;\mid\; a(\overrightarrow{d}) \;\mid\; \alpha|\beta,$$

Here, τ represents the empty multi-action; the term $a \in \mathcal{A}^{\mathrm{mCRL2}}$ denotes an action label and $\overrightarrow{d}{:}\overrightarrow{D}$ a vector of data expressions such that $D_i \in \mathcal{S}^{\mathrm{mCRL2}}$ for each $D_i \in \overrightarrow{D}$; and α, β are syntactic multi-actions. The syntactic multi-action $\alpha|\beta$ consists of the actions from both the syntactic multi-actions α and β.

Definition 2.2.9 (Process expression).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ be a signature, such that the tuple $\mathcal{D}^{\mathrm{mCRL2}} = (\Sigma^{\mathrm{mCRL2}}, E)$ is a *data specification*. Process expressions are expressions with the following syntax:

$$\begin{aligned} p ::= \delta \mid \alpha \mid p + p \mid p{\cdot}p \mid c{\to}p \mid c{\to}p{\diamond}p \mid \sum_{v:D} p \mid p \| p \mid p \between p \mid \\ \mid p|p \mid \Gamma_C(p) \mid \nabla_V(p) \mid \partial_B(p) \mid \rho_R(p) \mid \tau_I(p) \mid \Upsilon_U(p) \mid \\ X(v_1{=}d_1, \ldots, v_n{=}d_n). \end{aligned}$$

In the above BNF, p denotes a process term, α is a syntactic multi-action, $c : \mathbb{B}$ is a Boolean data-expression, v, $v_1, \ldots, v_n \in \mathcal{X}^{\mathrm{mCRL2}}$ ($n{\geq}0$) are variables, $D \in \mathcal{S}^{\mathrm{mCRL2}}$ is a sort, $C \subseteq \overrightarrow{\mathcal{A}^{\mathrm{mCRL2}}} \times \mathcal{A}^{\mathrm{mCRL2}}$ a set of communications, $V \subseteq \overrightarrow{\mathcal{A}^{\mathrm{mCRL2}}}$ a set of multi-action labels, $B \subseteq \mathcal{A}^{\mathrm{mCRL2}}$, $I \subseteq \mathcal{A}^{\mathrm{mCRL2}}$ and $U \subseteq \mathcal{A}^{\mathrm{mCRL2}}$ are sets of action labels, $R \subseteq \mathcal{A}^{\mathrm{mCRL2}} \times \mathcal{A}^{\mathrm{mCRL2}}$ is a set of renamings and $d_1, \ldots, d_n$ are data expressions.

For processes, $+$ denotes the non-deterministic choice, $c{\to}p$ denotes the conditional if-then execution, $c{\to}p{\diamond}p$ denotes the conditional if-then-else execution, $\sum_{v:D} p$ denotes the non-deterministic choice over the domain of D by selecting a value for variable v, $p \cdot p$ denotes the sequential composition, $p \between p$ denotes the left merge composition, $p|p$ denotes the sync operator and $p \parallel p$ denotes the parallel composition. The operator $\nabla_A(p)$ allows only the multi-actions from the set A of multi-action labels. The operator $\partial_B(p)$ blocks all actions from the set B of action labels. The operator $\Gamma_C(p)$ applies the communications described by the set C to a process. The operator $\tau_I(p)$ hides all actions from the set I of action

labels. The operator $\Upsilon_U(p)$ pre-hides all actions from the set U of action labels. The process term X is a recursion variable, and $X(v_1{=}d_1, \ldots, v_n{=}d_n)$ denotes a process reference to a *process equation* of the form $X(v_1{:}D_1, \ldots, v_n{:}D_n) = p$, i.e., the process $X(v_1{=}d_1, \ldots, v_n{=}d_n)$ behaves as p where the occurrences of $v_1, \ldots, v_n$ are substituted with $d_1, \ldots, d_n$.

Definition 2.2.10 (Process equation).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ be a signature. A *process equation* is an expression of the form $X(v_1{:}D_1, \ldots, v_n{:}D_n) = p$ where $n{\geq}0$, p is a process expression, and for $(1 \leq i \leq n)$, v_i are variables of sort D_i from $\mathcal{S}^{\mathrm{mCRL2}}$.

Definition 2.2.11 (Process specification).
A *process specification* is a five tuple $PS = (\mathcal{D}^{\mathrm{mCRL2}}, AD, PE, p, \mathcal{X}^{\mathrm{mCRL2}})$ where

- $\mathcal{D}^{\mathrm{mCRL2}}$ is a data specification,
- AD is an action declaration,
- PE is a set of process equations,
- p is a process expression, and
- $\mathcal{X}^{\mathrm{mCRL2}}$ is a set of global variables.

For reasons of simplicity, we assume that all process specifications and their underlying components are well-typed as described in [23].

2.2.2 Semantic concepts

A semantic multi-action is the interpretation of a syntactic multi-action, in which the model elements (i.e., a vector of data expressions) are interpreted under the data valuation σ. The data valuation σ represents a variable to value mapping, i.e., $\{v_1{:}D_1 \mapsto w_1{:}M_{D_1}, \ldots, v_n{:}D_n \mapsto w_n{:}M_{D_n}\}$ where $v_1, \ldots, v_n$ are variables, $w_1, \ldots, w_n$ are values and $M_{D_1}, \ldots M_{D_n}$ denote the respective corresponding (semantic) sorts (applicative $\mathcal{D}^{\mathrm{mCRL2}}$-structure). An element from a data valuation is called a *field*.

Definition 2.2.12 (Applicative $\mathcal{D}^{\mathrm{mCRL2}}$-structure).
Let $\mathcal{D}^{\mathrm{mCRL2}} = (\Sigma^{\mathrm{mCRL2}}, E)$ be a data specification. Then the collection of nonempty sets $\{M_D | D \in \mathcal{S}_S^{\mathrm{mCRL2}}\}$ is an *applicative $\mathcal{D}^{\mathrm{mCRL2}}$-structure* iff:

- $\mathcal{D}_{\mathbb{B}}^{\mathrm{mCRL2}}$ is a set with two elements, denoted by **true** and **false**, for which **true** $\neq$ **false** holds.
- $D \in \mathcal{S}^{\mathrm{mCRL2}}$ and D is not a function sort, then M_D is a nonempty set.
- $D = D_1 \times \ldots \times D_n \to D'$, then M_D is the set of all functions from $M_{D_1} \times \ldots \times M_{D_n} \to M_{D'_n}$.

Definition 2.2.13 (Valuation).
Let $\sigma{:}\mathcal{X}^{\mathrm{mCRL2}} \to \bigcup_{D \in \mathcal{S}_S^{\mathrm{mCRL2}}}$, then σ is a *valuation* if for all $v{:}\mathcal{X}_D^{\mathrm{mCRL2}}$ holds $\sigma(v) \in M_D$.

Definition 2.2.14 (Semantic interpretation).
Let $\Sigma^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, \mathcal{C}^{\mathrm{mCRL2}}, \mathcal{M}^{\mathrm{mCRL2}})$ be a signature, let $\mathcal{D}^{\mathrm{mCRL2}} = (\Sigma^{\mathrm{mCRL2}}, E)$ be a data specification and let σ be a data valuation.

We write $\sigma[v \mapsto w]$ for a valuation σ with function update $[v \mapsto w]$, that maps all variables according to σ, except for variable v. This variable is mapped to the value w.

We write $\sigma[v_i \mapsto w_i]_{1 \leq i \leq n}$ (or $\sigma[\overrightarrow{v} \mapsto \overrightarrow{w}]$) for a valuation σ with for $1 \leq i \leq n$ the function updates $[v_i \mapsto w_i]$, that maps all variables according to σ, except for variable v_i $(1 \leq i \leq n)$. These variables are mapped to the corresponding values w_i.

Then $\{\!|\cdot|\!\}^\sigma$ is the semantic interpretation function on a data expression defined through:

- $\{\!|x|\!\}^\sigma = \sigma(v)$ for every variable $v \in X_D$ $(D \in \mathcal{S}^{\mathrm{mCRL2}})$.
- $\{\!|f|\!\}^\sigma = \{\!|f|\!\}$ for every function symbol $f \in \mathcal{C}^{\mathrm{mCRL2}}_{\mathcal{S}} \cup \mathcal{M}^{\mathrm{mCRL2}}_{\mathcal{S}}$ and $\{\!|f|\!\} \in M_D$.
- $\{\!|p(p_1, \ldots, p_n)|\!\}^\sigma = \{\!|p|\!\}^\sigma(\{\!|p_1|\!\}^\sigma, \ldots, \{\!|p_n|\!\}^\sigma)$
- $\{\!|\lambda x_1{:}D_1, \ldots, x_n{:}D_n.p|\!\}^\sigma = f$ where $f : M_{D_1} \times \ldots \times M_{D_n} \to D$ is the function satisfying $f(d_1, \ldots, d_n) = \{\!|p|\!\}^{\sigma[x_i \mapsto d_i]_{1 \leq i \leq n}}$ for all $d{:}M_D$.
- $\{\!|\forall x{:}D.p|\!\}^\sigma = \mathbf{true}$ iff for all $d \in M_D$ it holds that $\{\!|p|\!\}^{\sigma[x \mapsto d]} = \mathbf{true}$.
- $\{\!|\exists x{:}D.p|\!\}^\sigma = \mathbf{true}$ iff for some $d \in M_D$ it holds that $\{\!|p|\!\}^{\sigma[x \mapsto d]} = \mathbf{true}$.
- $\{\!|p\ \mathbf{whr}\ x_1{=}p_1, \ldots, x_n{=}p_n\ \mathbf{end}|\!\}^\sigma = \{\!|p|\!\}^{\sigma[x_i \mapsto \{\!|p_i|\!\}^\sigma]_{1 \leq i \leq n}}$.

Definition 2.2.15 ($\mathcal{D}^{\mathrm{mCRL2}}$-model).
Let $\mathcal{D}^{\mathrm{mCRL2}} = (\mathcal{S}^{\mathrm{mCRL2}}, E)$ be a data specification and let σ be a data valuation Then a $\mathcal{D}^{\mathrm{mCRL2}}$-*model* is defined through $\{\!|\cdot|\!\}$:

- for every equation $c{\to}p_1{=}p_2 \in E_{\mathcal{S}}$ is holds that if $\{\!|c|\!\}^\sigma = \mathbf{true}$ then $\{\!|p_1|\!\}^\sigma = \{\!|p_2|\!\}^\sigma$ for every valuation σ.
- $\{\!|\mathit{true}|\!\}^\sigma = \mathbf{true}$ and $\{\!|\mathit{false}|\!\}^\sigma = \mathbf{false}$ for every valuation σ.
- If a basic sort D is a constructor sort (i.e. there is a constructor $f \in \mathcal{C}^{\mathrm{mCRL2}}_{\mathcal{S}}$ of sort $D_1 \times \ldots \times D_n \to D$), then every element $d \in M_D$ is a constructor element. A constructor element is inductively defined by:
  - Every element $d \in M_D$ is a constructor element, if D is a constructor sort and a constructor function $f \in \mathcal{C}^{\mathrm{mCRL2}}_{\mathcal{S}}$ of sort $D_1 \times \ldots \times D_n \to D$ exists such that $d = \{\!|f|\!\}(e_1, \ldots, e_n)$ where e_i is either a constructor element of sort D_i, or
  - sort D_i is not a constructor sort.

Definition 2.2.16 (Semantic multi-action).
Let $\mathcal{D}^{\mathrm{mCRL2}} = (\Sigma^{\mathrm{mCRL2}}, E)$ be a data specification, $\{\!|\cdot|\!\}^\sigma$ a $\mathcal{D}^{\mathrm{mCRL2}}$-model, E a set of data equations, $a \in \mathcal{A}^{\mathrm{mCRL2}}$ and $w_1 : M_{D_1}, \ldots, w_n : M_{D_n}$ are values. The interpretation of any syntactic multi-action α, β be inductively defined for any data-valuation σ by:

- $[\![\tau]\!]^\sigma = \tau$.
- $[\![a(w_1, \ldots, w_n)]\!]^\sigma = a(\{\![w_1]\!\}^\sigma, \ldots, \{\![w_n]\!\}^\sigma)$.
- $[\![\alpha|\beta]\!]^\sigma = [\![\alpha]\!]^\sigma | [\![\beta]\!]^\sigma$.

A semantic action that has no data parameter, i.e., $a()$, can be written as a.

Definition 2.2.17 (Semantic multi-action equivalence class).
Let α, β be semantic multi-actions, then the semantic multi-action equivalence relation is defined as the smallest equivalence relation that satisfies:

$$\begin{cases} \alpha \sim \alpha \\ \alpha \mid \tau \sim \alpha \\ \alpha \mid \beta \sim \beta \mid \alpha \\ (\alpha \mid \beta) \mid \gamma \sim \alpha \mid (\beta \mid \gamma) \end{cases}$$

The equivalence class with respect to $\sim$ of a multi-action α is denoted by a $\sim$ subscript:

$$\alpha_\sim = \{\beta \mid \beta \sim \alpha\}^1$$

Furthermore, we define a function that can merge separate equivalence classes into a new equivalence class. Let $a \in \mathcal{A}^{\text{mCRL2}}$ and $w_1 : M_{D_1}, \ldots, w_n : M_{D_n}$ denote values of sort $D_n \in \mathcal{S}^{\text{mCRL2}}$, then the function is represented by the following interpretation:

$$\begin{cases} (\tau_\sim)_\sim = \tau_\sim \\ (a(w_1, \ldots, w_n)_\sim)_\sim = a(w_1, \ldots, w_n))_\sim \\ (\alpha_\sim \mid \beta_\sim)_\sim = (\alpha \mid \beta)_\sim \end{cases}$$

The semantics of the processes are defined using so-called inference rules. These rules extract information from semantic multi-action equivalence classes.

Definition 2.2.18 (Functions on semantic multi-action equivalence classes).
Let $\alpha_\sim$ be a semantic multi-action equivalence class on which we define the functions

- $\alpha_\sim^{\{\}}$ is the set of all action labels occurring in the semantic multi-action equivalence class $\alpha_\sim$, i.e.:
  - $\tau_\sim^{\{\}} = \emptyset$
  - $a(w_1, \ldots, w_n)_\sim^{\{\}} = \{a\}$
  - $\alpha|\beta_\sim^{\{\}} = \alpha_\sim^{\{\}} \cup \beta_\sim^{\{\}}$
- $\underline{\alpha_\sim}$ denotes the semantic multi-action equivalence class $\alpha_\sim$ where all data has been removed, i.e.:
  - $\underline{\tau_\sim} = \tau_\sim$
  - $\underline{a(w_1, \ldots, w_n)_\sim} = a_\sim$
  - $\underline{(\alpha \mid \beta)_\sim} = (\underline{\alpha_\sim} \mid \underline{\beta_\sim})_\sim$

[1] The '|' denotes the separator symbol for set comprehension: not the symbol to separate actions in a multi-action.

- Let R be a set of renamings. Then the function $R \bullet (\alpha_\sim)$ denotes the renaming on the semantic multi-action equivalence class where the action labels are renamed according to R, i.e.:
  - $R \bullet (\tau_\sim) = \tau_\sim$
  - $R \bullet (a(w_1, \ldots, w_n)_\sim) = \begin{cases} b(w_1, \ldots, w_n)_\sim & \text{if } a \to b \in R \text{ for some } b \\ a(w_1, \ldots, w_n)_\sim & \text{if } a \to b \notin R \text{ for all } b \end{cases}$
  - $R \bullet ((\alpha|\beta)_\sim) = (R \bullet (\alpha_\sim) \mid R \bullet (\beta_\sim))_\sim$
- $\theta_I(\alpha_\sim)$ hides the actions in a semantic multi-action equivalence class $\alpha_\sim$ with labels that occur in I, i.e.:
  - $\theta_I(\tau_\sim) = \tau_\sim$
  - $\theta_I(a(w_1, \ldots, w_n)_\sim) = \begin{cases} \tau_\sim & \text{if } a \in I \\ a(w_1, \ldots, w_n)_\sim & \text{if } a \notin I \end{cases}$
  - $\theta_I((\alpha|\beta)_\sim) = (\theta_I(\alpha_\sim) \mid \theta_I(\beta_\sim))_\sim$
- $\eta_U(\alpha_\sim)$ prehides the actions in a semantic multi-action equivalence class $\alpha_\sim$ with labels that occur in U by removing the data parameters and relabeling the action label to *int*, i.e.:
  - $\eta_U(\tau_\sim) = \tau_\sim$
  - $\eta_U(a(w_1, \ldots, w_n)_\sim) = \begin{cases} int_\sim & \text{if } a \in U \\ a(w_1, \ldots, w_n)_\sim & \text{if } a \notin U \end{cases}$
  - $\eta_U((\alpha|\beta)_\sim) = (\eta_U(\alpha_\sim) \mid \eta_U(\beta_\sim))_\sim$
- Communication is defined using γ_C. Let $c_i \equiv c_i^1 | \ldots | c_i^{m_i}$, then we define the communication function $C = \{c_1 \to c'_1, \ldots, c_n \to c'_n\}$[2], where $c_1, \ldots, c_n \in \overrightarrow{\mathcal{A}^{\text{mCRL2}}}$ and $c'_1, \ldots, c'_n \in \mathcal{A}^{\text{mCRL2}}$. The specification assumes that all action labels of the domain of a single communication function are pairwise disjoint, i.e.:
$$\forall_{I,J \in dom(C)} I \neq J \Rightarrow I \cap J = \emptyset$$
Communication takes place over a semantic multi-action equivalence class, only for those actions for which the arguments have the same semantic logic equivalent values. Let $\overrightarrow{w} : \overrightarrow{M_D}$, let $c_i(\overrightarrow{w}) \equiv c_i^1(\overrightarrow{w}) \mid \ldots \mid c_i^{m_i}(\overrightarrow{w})$, and let $\alpha_\sim \sqsubseteq \beta_\sim$ be the inclusion of $\alpha_\sim$ in $\beta_\sim$. Then we specify the $\gamma_C(\alpha_\sim)$ as:
$$\gamma_C(\alpha_\sim) = \begin{cases} (c'_i(\overrightarrow{w}) \mid \gamma_C(\alpha_\sim \backslash c_i(\overrightarrow{w})_\sim))_\sim & \text{if } \exists_{\overrightarrow{w}} \exists_{c_i} (c_i \to c'_i) \in C \\ & \quad \wedge\ c_i(\overrightarrow{w})_\sim \sqsubseteq \alpha_\sim \\ \alpha_\sim & \text{if } \forall_{\overrightarrow{w}} \forall_{c_i} (c_i \to c'_i) \notin C \\ & \quad \vee\ c_i(\overrightarrow{w})_\sim \not\sqsubseteq \alpha_\sim \end{cases}$$
The function above defines the communication recursively. Intuitively, if we find a set of actions labels (obtained after the data elimination of a semantic multi-action equivalence class) that occurs in the domain of communication function, it is replaced by the corresponding image with the appropriate data values, and the communication function is again applied

[2]The $\to$ is here used as a syntactic expression

to the remainder of multi-action equivalence class. The communication return the input, if no instances can be found. While the synchronization domains of a communication function C are all pairwise disjoint, it implies that all communication functions are orthogonal. Hence, the order in which the functions are applied does not effect the outcome of the communication function C.

2.2.3 Operational semantics

Given a data specification and a process expression, we express the semantics of the process expression through a transition system. The way in which a process expression relates to a transition system is described via deduction rules.

Definition 2.2.19 (Semantics of a process).
Let $PS = (\mathcal{D}^{\text{mCRL2}}, AD, PE, p, \mathcal{X}^{\text{mCRL2}})$ be a process specification. Furthermore, let $\{M_D | D \in \mathcal{S}^{\text{mCRL2}}\}$ be a $\mathcal{D}^{\text{mCRL2}}$-model where M_D is the domain of sort D, $[\![\cdot]\!]^\sigma$ a semantic-interpretation and σ a data valuation. We define the semantics of a process specification PS given $\mathcal{A}^{\text{mCRL2}}$ and σ as a transition system $A = (S, Act, \longrightarrow, s_0, T)$ as follows:

- The states S contain all pairs (p', σ') for process expressions p' and valuations σ'. There is one special termination state, denoted by the $\checkmark$ predicate.
- A label denotes a semantic multi-action equivalence class.
- The transitions are inductively defined by the operational rules in Tables 2.1, 2.2, 2.3, 2.4, 2.5 and 2.6. These rules describe the semantics of the untimed fragment of the mCRL2 language. The transition relation is denoted by $(p', \sigma) \xrightarrow{m} (p'', \sigma')$ or $(p', \sigma) \xrightarrow{m} \checkmark$.
- The initial state is (p, σ_0) where σ_0 is the initial data valuation.

2.2.4 Discussion

Some of the deduction rules that are presented in Tables 2.1, 2.2, 2.3, 2.4, 2.5 and 2.6 are not suitable for the transformation. The deduction rules that potentially cause difficulties are described in this subsection, accompanied with a alternative representation. The alternative representation will be used in the following chapters to transform and conduct the analysis.

The deduction rule def_2 introduces fresh variables with respect to σ. Since we assume an infinite set of variables and the deduction rules impose no restriction on the generated fresh variables, there are infinitely many ways to instantiate $\overrightarrow{v}'$. Hence, the recursion operator would dictate infinite branching. In theory this kind of behavior is poses no problems. However in practice this results in behavior that cannot be analyzed, if no abstractions or restrictions are applied. Therefore, we assume that the rules only specify one $\overrightarrow{v}'$, for which all of the variables are disjoint from $dom(\sigma)$. For our convenience, and for the purpose of abstracting from the details of generating fresh variable names, we assume given a predicate $fresh{:}fresh(\sigma, \overrightarrow{v})$ holds only for those variables $\overrightarrow{v}$ that are generated

$$(ma)\frac{}{(\alpha,\sigma)\xrightarrow{[\![\alpha]\!]^\sigma_\sim}\checkmark}$$

$$(alt_1)\frac{(p,\sigma)\xrightarrow{m}\checkmark}{(p+q,\sigma)\xrightarrow{m}\checkmark} \qquad (alt_2)\frac{(p,\sigma)\xrightarrow{m}(p',\sigma')}{(p+q,\sigma)\xrightarrow{m}(p',\sigma')}$$

$$(alt_3)\frac{(q,\sigma)\xrightarrow{m}\checkmark}{(p+q,\sigma)\xrightarrow{m}\checkmark} \qquad (alt_4)\frac{(q,\sigma)\xrightarrow{m}(q',\sigma')}{(p+q,\sigma)\xrightarrow{m}(q',\sigma')}$$

$$(seq_1)\frac{(p,\sigma)\xrightarrow{m}\checkmark}{(p{\cdot}q,\sigma)\xrightarrow{m}(q,\sigma)} \qquad (seq_2)\frac{(p,\sigma)\xrightarrow{m}(p',\sigma')}{(p{\cdot}q,\sigma)\xrightarrow{m}(p'{\cdot}q,\sigma')}$$

$$(cond_1)\frac{[\![b]\!]^\sigma=\mathbf{true}\quad (p,\sigma)\xrightarrow{m}\checkmark}{(b{\rightarrow}p,\sigma)\xrightarrow{m}\checkmark} \qquad (cond_2)\frac{[\![b]\!]^\sigma=\mathbf{true}\quad (p,\sigma)\xrightarrow{m}(p',\sigma')}{(b{\rightarrow}p,\sigma)\xrightarrow{m}(p',\sigma')}$$

$$(cond'_1)\frac{[\![b]\!]^\sigma=\mathbf{true}\quad (p,\sigma)\xrightarrow{m}\checkmark}{(b{\rightarrow}p{\diamond}q,\sigma)\xrightarrow{m}\checkmark} \qquad (cond'_2)\frac{[\![b]\!]^\sigma=\mathbf{true}\quad (p,\sigma)\xrightarrow{m}(p',\sigma')}{(b{\rightarrow}p{\diamond}q,\sigma)\xrightarrow{m}(p',\sigma')}$$

$$(cond'_3)\frac{[\![b]\!]^\sigma=\mathbf{false}\quad (q,\sigma)\xrightarrow{m}\checkmark}{(b{\rightarrow}p{\diamond}q,\sigma)\xrightarrow{m}\checkmark} \qquad (cond'_4)\frac{[\![b]\!]^\sigma=\mathbf{false}\quad (q,\sigma)\xrightarrow{m}(q',\sigma')}{(b{\rightarrow}p{\diamond}q,\sigma)\xrightarrow{m}(q',\sigma')}$$

Table 2.1: Structural Operational Semantics for the basic operators

by the mechanism of generating fresh variable names. Reflecting this discussion, we provide the following deduction rule for def_2:

$$(def_2)\frac{(q[\overrightarrow{v}\mapsto\overrightarrow{v}'],\sigma[\overrightarrow{v}'\mapsto[\![\overrightarrow{d}]\!]^\sigma])\xrightarrow{m}(q',\sigma') \qquad fresh(\sigma,\overrightarrow{v}')}{(X(\overrightarrow{w}),\sigma)\xrightarrow{m}(q',\sigma')}$$

where $X(\overrightarrow{v}{:}\overrightarrow{D})=q\in PE$.

The deduction rules par_8 and $sync_4$ silently assume that:

$$\sigma'(v)=\sigma''(v)$$

for all $v\in dom(\sigma)$ and

$$dom(\sigma')\cap dom(\sigma'')=dom(\sigma)$$

This means that the freshly introduced variables (if any) are different for the two premises of the deduction rules. Since this assumption is not mentioned in the semantics, we explicitly state these assumptions here. For the first assumption we introduce the notation $\sigma' =_{dom(\sigma)} \sigma''$.

$$(sum_1)\frac{(p,\sigma[v\mapsto w]) \xrightarrow{m} \checkmark}{(\sum_{v:D} p,\sigma) \xrightarrow{m} \checkmark}\ w{\in}\mathcal{M}_D \qquad (sum_2)\frac{(p,\sigma[v\mapsto w]) \xrightarrow{m} (p',\sigma')}{(\sum_{v:D} p,\sigma) \xrightarrow{m} (p',\sigma')}\ w{\in}\mathcal{M}_D$$

Table 2.2: Structural Operational Semantics for the sum operator

$$(par_1)\frac{(p,\sigma) \xrightarrow{m} \checkmark}{(p\|q,\sigma) \xrightarrow{m} (q,\sigma)} \qquad (par_2)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma')}{(p\|q,\sigma) \xrightarrow{m} (p'\|q,\sigma')}$$

$$(par_3)\frac{(q,\sigma) \xrightarrow{m} \checkmark}{(p\|q,\sigma) \xrightarrow{m} (p,\sigma)} \qquad (par_4)\frac{(q,\sigma) \xrightarrow{m} (q',\sigma')}{(p\|q,\sigma) \xrightarrow{m} (p\|q',\sigma')}$$

$$(par_5)\frac{(p,\sigma) \xrightarrow{m} \checkmark,\ (q,\sigma) \xrightarrow{n} \checkmark}{(p\|q,\sigma) \xrightarrow{m|n} \checkmark} \qquad (par_6)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma'),\ (q,\sigma) \xrightarrow{n} \checkmark}{(p\|q,\sigma) \xrightarrow{m|n} (p',\sigma')}$$

$$(par_7)\frac{(p,\sigma) \xrightarrow{m} \checkmark,\ (q,\sigma) \xrightarrow{n} (q',\sigma')}{(p\|q,\sigma) \xrightarrow{m|n} (q',\sigma')} \qquad (par_8)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma'),\ (q,\sigma) \xrightarrow{n} (q',\sigma'')}{(p\|q,\sigma) \xrightarrow{m|n} (p'\|q',\sigma'\cup\sigma'')}$$

Table 2.3: Structural Operational Semantics for the parallel operator

Since we have adapted deduction rule def_2 to only generate specific fresh variables, the variables generated freshly by two different premises can have the exact same variables.

Example 2.2.20. Assume that we have the process equation: $P(v{:}\mathbb{B}) = a_1(v) \cdot a_2(v)$. Then if we model $P(\mathit{false}) \parallel P(\mathit{true})$, two fresh variables are generated. Since the variable generation is performed on the same data valuation, the exact same variables are generated. This would mean that we have a premise $(a_1(v) \cdot a_2(v), \{v \mapsto \mathit{false}\})$ on the left and that we have the premise (q,σ') that is defined as $(a_1(v) \cdot a_2(v), \{v \mapsto \mathit{true}\})$ on the right. Based on the individual behaviors we would expect that par_8 performs the semantic multi-action equivalence class $a_1(\mathit{true}) \mid a_1(\mathit{false})$. As the data valuations are merged after the transition, we would obtain an ill defined function, for which it is not possible to observe the second transition $a_2(\mathit{true}) \mid a_2(\mathit{false})$.

To avoid the potential clash of variables, we rename all freshly generated variables from the second premise that are also introduced by the first premise. For this we use the same mechanism as for generating fresh variables.

$$(par_8)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma'),\ (q,\sigma) \xrightarrow{n} (q',\sigma'') \qquad \sigma' =_{dom(\sigma)} \sigma'' \qquad \mathit{fresh}(\sigma', \overrightarrow{v'})}{(p\|q,\sigma) \xrightarrow{m|n} (p'\|q'[\overrightarrow{v} \mapsto \overrightarrow{v'}], \sigma'[\overrightarrow{v} \mapsto \overrightarrow{v'}])}$$

$$(lmerge_1)\frac{(p,\sigma) \xrightarrow{m} \checkmark}{(p \between q,\sigma) \xrightarrow{m} (q,\sigma)} \qquad (lmerge_2)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma')}{(p \between q,\sigma) \xrightarrow{m} (p' \| q,\sigma')}$$

$$(sync_1)\frac{(p,\sigma) \xrightarrow{m} \checkmark,\ (q,\sigma) \xrightarrow{n} \checkmark}{(p|q,\sigma) \xrightarrow{m|n} \checkmark} \qquad (sync_2)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma'),\ (q,\sigma) \xrightarrow{n} \checkmark}{(p|q,\sigma) \xrightarrow{m|n} (p',\sigma')}$$

$$(sync_3)\frac{(p,\sigma) \xrightarrow{m} \checkmark,\ (q,\sigma) \xrightarrow{n} (q',\sigma')}{(p|q,\sigma) \xrightarrow{m|n} (q',\sigma')} \qquad (sync_4)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma'),\ (q,\sigma) \xrightarrow{n} (q',\sigma'')}{(p|q,\sigma) \xrightarrow{m|n} (p' \| q',\sigma' \cup \sigma'')}$$

Table 2.4: Structural Operational Semantics for the auxiliary parallel operators

where $\overrightarrow{v}$ are the variables freshly generated due to the first premise, i.e., $dom(\sigma') \setminus dom(\sigma)$.

Similar to par_8 we also redefine $sync_4$ as

$$(sync_4)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma'),\ (q,\sigma) \xrightarrow{n} (q',\sigma'') \qquad \sigma' =_{dom(\sigma)} \sigma'' \qquad fresh(\sigma', \overrightarrow{v}')}{(p|q,\sigma) \xrightarrow{m|n} (p' \| q'[\overrightarrow{v} \mapsto \overrightarrow{v}'], \sigma'[\overrightarrow{v} \mapsto \overrightarrow{v}'])}$$

2.3 (Simplified) Linear Process Specifications

A Linear Process Specification (LPS) is a symbolic representation for capturing (possibly infinite) Labeled Transition Systems (LTS). Informally, an LPS consists of a signature, variable declarations, a collection of data equations, action declarations, a linear process equation, and an initialization. A (full) formal definition of an LPS and its components can be found in [16].

An LPS is a restricted mCRL2 specification. That is, the process specification is defined through a single process equation that represents the behavior of the mCRL2 specification. A full explanation on the relation between the specifications can be found in [41]. The signature, variables, data equations and action declarations of the LPS respectively correspond to counterparts in mCRL2.

Definition 2.3.1 (Linear Process Equation).
A Linear Process Equation (LPE) is an equation of the form:

$$X(d{:}D) \quad = \quad \sum_{i \in I} \sum_{e_i : E_i} c_i(d,e_i) \to a_i(f_i(d,e_i)) \cdot X(g_i(d,e_i))$$

where I is a finite index set, where for $i \in I$ holds:

- e_i and E_i respectively denote a variable name and a sort expression,
- $c_i(d,e_i)$ is a term of sort $\mathbb{B}$ (denoting the set of Boolean values) that serves as a Boolean guard to allow actions,
- $a_i(f_i(d,e_i))$, where $a \in \mathcal{A}^{\text{mCRL2}}$ is a set of action labels, and f_i is a data expression on d and e_i that represents the (possibly empty) list of data parameters,

$$(allow_1)\frac{\underline{m}_{\sim} \cap (V \cup \{\tau\}) \neq \emptyset \quad (p,\sigma) \xrightarrow{m} \checkmark}{(\nabla_V(p),\sigma) \xrightarrow{m} \checkmark} \qquad (allow_2)\frac{\underline{m}_{\sim} \cap (V \cup \{\tau\}) \neq \emptyset \quad (p,\sigma) \xrightarrow{m} (p',\sigma')}{(\nabla_V(p),\sigma) \xrightarrow{m} (\nabla_V(p'),\sigma')}$$

$$(encap_1)\frac{m^{\{\}} \cap B = \emptyset \quad (p,\sigma) \xrightarrow{m} \checkmark}{(\partial_B(p),\sigma) \xrightarrow{m} \checkmark} \qquad (encap_2)\frac{m^{\{\}} \cap B = \emptyset \quad (p,\sigma) \xrightarrow{m} (p',\sigma')}{(\partial_B(p),\sigma) \xrightarrow{m} (\partial_B(p'),\sigma')}$$

$$(ren_1)\frac{(p,\sigma) \xrightarrow{m} \checkmark}{(\rho_R(p),\sigma) \xrightarrow{R\bullet(m)} \checkmark} \qquad (ren_2)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma')}{(\rho_R(p),\sigma) \xrightarrow{R\bullet(m)} (\rho_R(p'),\sigma')}$$

$$(comm_1)\frac{(p,\sigma) \xrightarrow{m} \checkmark}{(\Gamma_C(p),\sigma) \xrightarrow{\gamma_C(m)} \checkmark} \qquad (comm_2)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma')}{(\Gamma_C(p),\sigma) \xrightarrow{\gamma_C(m)} (\Gamma_C(p'),\sigma')}$$

$$(hide_1)\frac{(p,\sigma) \xrightarrow{m} \checkmark}{(\tau_I(p),\sigma) \xrightarrow{\theta_I(m)} \checkmark} \qquad (hide_2)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma')}{(\tau_I(p),\sigma) \xrightarrow{\theta_I(m)} (\tau_I(p'),\sigma')}$$

$$(prehide_1)\frac{(p,\sigma) \xrightarrow{m} \checkmark}{(\Upsilon_U(p),\sigma) \xrightarrow{\eta_U(m)} \checkmark} \qquad (prehide_2)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma')}{(\Upsilon_U(p),\sigma) \xrightarrow{\eta_U(m)} (\Upsilon_U(p'),\sigma')}$$

Table 2.5: Structural Operational Semantics for the auxiliary operators

$$(def_1)\frac{(q,\sigma[\overrightarrow{v} \mapsto \{\!|\overrightarrow{d}|\!\}^{\sigma}]) \xrightarrow{m} \checkmark}{(X(\overrightarrow{v}=\overrightarrow{d}),\sigma) \xrightarrow{m} \checkmark}$$

$$(def_2)\frac{(q[\overrightarrow{v} \mapsto \overrightarrow{v}'],\sigma[\overrightarrow{v}' \mapsto \{\!|\overrightarrow{d}|\!\}^{\sigma}]) \xrightarrow{m} (q',\sigma')}{(X(\overrightarrow{v}=\overrightarrow{d}),\sigma) \xrightarrow{m} (q',\sigma')}$$

where $X(\overrightarrow{v}{:}\overrightarrow{D}) = q \in PE$ and $\overrightarrow{v}'$ are fresh variables of sort $\overrightarrow{D}$ with respect to σ, i.e $\overrightarrow{v}' \notin dom(\sigma)$

Table 2.6: Operational rules for recursion

- $g_i(d, e_i)$ is a term of sort D that denotes the next state.

The original definition of an LPE allows more features such as multi-actions, time annotations, and LPE termination, which are not needed in this paper and are therefore omitted. The initialization is a statement of the form $X(d)$, where d is a term of sort D.

2.3.1 Discussion

It might seem peculiar that we define the LPS without multi-actions, time annotations, and LPE termination as the mCRL2 language facilitates these features. These concepts will be represented as the outcome of functions, denoted within the data specification of an LPS. Since it is not possible to use data directly as an action we capture these concepts in the data parameters of the action. This means that the different transition relations are represented by different labeled actions, multi-actions are represented by action data parameters, and termination of an mCRL2 specification is encoded as a special state in which the LPS cannot perform an action.

Chapter 3

Approach

In this report we describe the transformation of the TSS which belongs to the mCRL2 language, along with a mCRL2 model into an LPS. The idea of the approach is already described in [37, 36]. In short, based on a set of deduction rules and an instance of a model, we transform the deduction rules (with the help of the framework for semantic interpretation) into a data specification of an LPS. The provided concrete model serves as the initial value for the LPE. Subsequently, the obtained LPS can be subjected to different kinds of analysis, e.g. simulation, state space exploration and even verification of modal properties. We already demonstrated feasibility of our approach by formalizing and using a prototype implementation of the semantics of a domain specific language [38]. In this report we show how to exactly implement the framework and deductions rules for a large and complete formal specification language. That is, we decided to take our framework and approach to the next level by dogfooding the Structural Operational Semantics of mCRL2 to the mCRL2 toolset. A schematic overview of our approach is given in Figure 3.1.

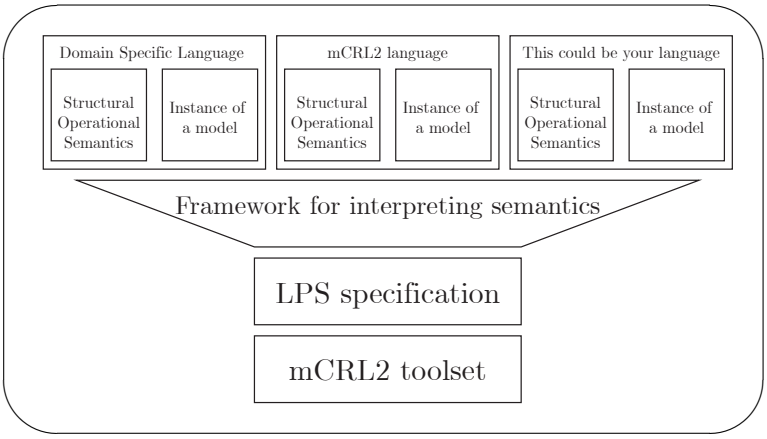


Figure 3.1: Decomposition of the semantic interpretation

To succeed we (i) construct a sort that captures the signature of an mCRL2 process term. Then we (ii) transform the SOS deduction rules into mCRL2 data equations and (iii) compute the (different) transition relations that belong to

a particular term with the help of an LPE. The domain in which we describe steps (i), (ii) and (ii) is called the *meta notation*.

The approach described in [37] only discusses the transformation for deduction rules given in De Simone format [35]. Given the fact that there exists a lattice of SOS formats [29] and the fact that we are dealing with a much richer language in this report, we have to extend our framework. The framework extensions covered in this report include the use of data valuations, data parameters in action transitions, multi-actions, action renaming and the generation of fresh variables. In this report, we design and implement the framework and its extensions in such a way that we can deal with the current computational limits of the mCRL2 toolset.

For clarity, we provide the mapping of the syntactic and semantic concepts within the mCRL2 language and the way they are represented in the meta notation. We also indicate whether they are model specific or language specific. Table 3.1 provides the conceptual mapping.

mCRL2 concept	**meta notation**	**specification**
process expression signature	structured sort $\mathcal{P}$	language
✓predicate	element of $\mathcal{P}$	language
process term	element of $\mathcal{P}$	model
syntactic mulit-action	container sort $List(Act_{\Xi})$	language
syntactic action	structured sort Act_{Ξ}	language
syntactic action label	structure sort Act_{Lab}	model
data expression	structured $\mathcal{E}$	language
build-in sorts $Sort_i$	-	language
user defined sorts $Sort_u$	sort $Sort_u$	model
typed variable	structured sort $\mathcal{V}$	model
variable label	structured sort $\mathcal{V}_{Lab}$	model
typed value	structured sort Λ	model
data valuation	container sort $List(\mathcal{I})$	language
	lambda abstraction $\mathcal{V} \to \Lambda$	language
data valuation field	structured sort $\mathcal{I}$	language
semantic action	structured sort Act_{Σ}	language
sem. multi-acton equiv. class $\alpha_{\sim}$	container sort $List(Act_{\Sigma})$	language
action transition: $\_ \xrightarrow{\_} \_$	transition relation $\underline{t}_{\rightarrow}$	language
deduction rule $\frac{H}{C}$	set comprehension $\{\_:\_ \mid \_:\mathbb{B}\}$	language
semantic interpretation $[\![\cdot]\!]^{\sigma}$ / $\{\!|\cdot|\!\}^{\sigma}$:		
data expression	mapping $sem_{\mathcal{E}}$	model
data expression list	mapping $sem_{\mathcal{E}}^{List}$	language
syntactic multi-action	mapping $sem_{Act_{\Xi}}^{List}$	language
syntactic action	mapping $sem_{Act_{\Xi}}$	language
functions on $\alpha_{\sim}$ i.e., $f(\alpha_{\sim})$	mapping f	language
process labels	sort $\mathcal{X}$	model
process parameter update	sort $\mathcal{Q}$	model
system of process equations	mapping PES	model

Table 3.1: Mapping mCRL2 language concepts to meta notation concepts

Chapter 4

Language specific design decisions

An SOS rule can describe arbitrary premises that are suitable for computation (e.g. $P \stackrel{?}{=} NP$) or are stated in a meta notation (e.g. "let there be a fresh variable d'"). Therefore we first evaluate the deduction rules such that we provide suitable modeling concepts for the meta notation. The remainder of this section explains the formalization for:

- the interpretation of the ✓ predicate (Chapter 4.1),
- the modeling of the signature of a process term (Chapter 4.2),
- the modeling of the data valuation (Chapter 4.3),
- the representation of data expressions in the meta notation (Chapter 4.4),
- the computation of syntactic multi-actions into semantic multi-action equivalence classes (Chapter 4.5),
- the representation of the LPE (Chapter 4.6).

4.1 Successful termination

The successful termination of some behavior is denoted by the ✓ predicate. In [18] we see that these predicates are coded as binary relations. Predicates can be used for various purposes and can have different representations, e.g. divergence [1], enabledness [5], probabilistic behavior [24], priorities [11], etc.

Basically, the termination predicate can be represented and modeled in four different ways. The first transition relation describes the ordinary action transition relation. The second transition relation describes the transition relation for termination.

1. The first option is the one as presented in the mCRL2 language:

$$\begin{array}{l} \_ \xrightarrow{a} \_ \\ \_ \xrightarrow{a} \checkmark \end{array}$$

2. The second option is to present the predicate as a different transformation relation:

$$\begin{array}{c} \_ \xrightarrow{a} \_ \\ \_ \stackrel{a}{\Longrightarrow} \_ \end{array}$$

3. The third option is to extend the action label:

$$\_ \xrightarrow{a|\checkmark} \_$$

4. The fourth option introduce a special process to model the ✓ predicate:

$$(p, \sigma) \xrightarrow{a} (\checkmark_{\mathrm{p}}, \sigma)$$

In [37] the authors show how (1) can be modeled. Basically, for the case that a process term successfully terminates, we compute a transition relation function for which we are only interested in the transition. The resulting process term and data valuation are considered to be irrelevant. If this transition relation function provides a result, we deal with a termination predicate. Unfortunately, when modeled like this we require a second transition relation function that is only used to describe the termination predicate. As the function is nearly identical to the transition relation function, we will require an additional 500 lines of code extra.

For (2) we introduce an additional transition relation, i.e., the termination relation. Since the solutions for each of the relations are computed separately, we see a similar amount of replication as with the previous solution (1). Extending the action label is another option, as seen in (3). However, the extension of the action label alters the semantics significantly. This implies that we also redefine the deduction rules. Since we want to model (and study) the current semantics of the mCRL2 language, we do not consider this as a wise approach.

In the last proposal (4), we model the predicate as a (special) process term. Then, based on the inspection of a state, we can determine if the process resides in a successful terminated state. Since the data valuation is irrelevant in a successfully terminating state, we assume that the data valuation remains unchanged with respect to the valuation prior to the transition. This assumption propagates for all deduction rules that describe successful termination. Based on these observations we consider model (4) to provide the cleanest solution. Therefore, we model the ✓ predicate by adding a (special) process term, i.e., $\checkmark_{\mathrm{p}}$.

4.2 Process term

Process terms in mCRL2 are multi-sorted. The multi-sorted terms are introduced by the arguments of the BNF element. These describe e.g. the condition in the choice operator, or the action labels that need to synchronize in the communication. For each of these arguments we introduce appropriate sorts. The designated sorts are discussed in Chapter 5. For now we assume that we know the appropriate sorts.

To capture the signature of a process term, we introduce the structured sort $\mathcal{P}$. The signature is modeled by the structured elements such that the signature of a process term is represented by a prefix notation. For every BNF element in Definition 2.2.9, we introduce a constructor that carries the textual characterization of the element. To model the ✓ predicate, we include the (special) aforementioned process term, modeled via the $\checkmark_{\mathrm{p}}$ constructor.

Projection functions are added to access the arguments of the modeled signature. Here, π_n denotes the n^{th} argument of a process term. Also, to access other arguments, like C in a conditional choice, we add projection functions as well. In addition, recognizer functions are added to recognize process terms. These recognizer functions are provided after the question mark. They only evaluate to *true* iff the instance of the sort matches the corresponding constructor function.

$$
\begin{aligned}
\textbf{sort}\quad \mathcal{P} = \textbf{struct}\ & \checkmark_{\mathrm{p}}?is_{\checkmark} \\
& \mid deadlock?is_{\delta} \\
& \mid alpha(multiaction{:}List(Act_{\Xi}))?is_{\alpha} \\
& \mid alt(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{alt} \\
& \mid seq(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{seq} \\
& \mid cond1(C{:}\mathcal{E}, \pi_1{:}\mathcal{P})?is_{cond1} \\
& \mid cond2(C{:}\mathcal{E}, \pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{cond2} \\
& \mid Sum(d{:}\mathcal{V}, \pi_1{:}\mathcal{P})?is_{sum} \\
& \mid par(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{par} \\
& \mid lmerge(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{lmerge} \\
& \mid sync(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{sync} \\
& \mid Allow(V{:}Set(Bag(Act_{Lab})), \pi_1{:}\mathcal{P})?is_{allow} \\
& \mid Block(B{:}Set(Act_{Lab}), \pi_1{:}\mathcal{P})?is_{block} \\
& \mid Rename(Ren{:}Act_{Lab} \to Act_{Lab}, \pi_1{:}\mathcal{P})?is_{rename} \\
& \mid Hide(I{:}Set(Act_{Lab}), \pi_1{:}\mathcal{P})?is_{hide} \\
& \mid Prehide(U{:}Set(Act_{Lab}), \pi_1{:}\mathcal{P})?is_{prehide} \\
& \mid Comm(CL{:}List(\mathcal{C}), \pi_1{:}\mathcal{P})?is_{comm} \\
& \mid Def(P{:}\mathcal{X}, ppl{:}List(\mathcal{Q}))?is_{def} \\
& ;
\end{aligned}
$$

4.2.1 Discussion

For convenience, the process term signature $\mathcal{P}$ represents a multi-action as a list of actions. The way in which they are modeled are explained in Chapter 4.5.

If we precisely would have modeled the signature, we would have had to introduce a separate sort to model a syntactic action. This sort would incorporate the structure of the '|' in a syntactic multi-action, by:

$$
\begin{aligned}
\textbf{sort}\quad MultiAction = \textbf{struct}\ & tau \mid Act_{\Xi} \\
& \mid bar(multiaction_1{:}MultiAction, multiaction_2{:}MultiAction);
\end{aligned}
$$

By transforming the '|' into a list of actions we provide a somewhat less verbose, but still recognizable, structure that represents a syntactic multi-action.

4.3 Data valuation

A valuation consists of a set of variable to value mappings, which is represented by σ in the deduction rules. In the meta notation we model a data valuation in two separate ways. The first representation is used to manipulate a data valuation.

The second representation is used to compute the semantic interpretation of a syntactic multi-action. For both valuations we need a suitable meta notation. This also includes suitable notations for variables and values. Furthermore, we motivate the necessity of having two instances of a data valuation modeled in the meta notation.

Note that the data valuation will only be used to compute the semantic interpretation for values, variables and data expressions. Therefore it is not possible to denote lambda expressions, quantifiers and where-clauses in the meta notation, although it is possible to specify them in a mCRL2 specification. We believe that it is possible to provide a suitable interpretation for these concepts in the meta notation, but we have decided that these are out of scope.

4.3.1 Values

In an mCRL2 specification different values can belong to different sorts. Within the meta notation we model them as one sort. To incorporate the different sorts modeled by a single sort, we introduce the structured sort Λ that can represent all sorts with the help of constructor functions. Since infinitely many sorts exist, we only add those sorts to the meta notation that occur in the original mCRL2 specification. So, we can define the meta notation sort Λ as

$$\textbf{sort} \quad \Lambda = \textbf{struct } \mathit{Sort}^1_\Lambda(s_1{:}\mathit{Sort}_1)?\mathit{is}_{\mathit{Sort}_1} \mid \ldots \mid \mathit{Sort}^n_\Lambda(s_n{:}\mathit{Sort}_n)?\mathit{is}_{\mathit{Sort}_n} \mid \bot;$$

where for $i \in [1, n]$, $\mathit{Sort}_i \in \mathcal{S}^{\text{mCRL2}}$ denotes the sorts occurring in the mCRL2 specification, Sort^i_Λ denotes the constructor function for Sort_i in the meta notation, $\mathit{is}_{\mathit{Sort}_i}$ denotes the recognizer function in the meta notation and $\bot$ denotes the ill formed or undefined value. This results in a type system where typed variables are encoded in prefix notation, e.g. "$s{:}S$" becomes "$S(s)$".

Example 4.3.1. Let the mCRL2 specification define the sorts $\mathbb{B}$ and $\mathbb{N}$. Then we define the structured sort Λ in the meta notation as

$$\textbf{sort} \quad \Lambda = \textbf{struct } \mathbb{B}_\Lambda(b{:}\mathbb{B})\mathit{is}_{\mathbb{B}} \mid \mathbb{N}_\Lambda(n{:}\mathbb{N})?\mathit{is}_{\mathbb{N}} \mid \bot;$$

Note, that we introduce appropriate names for the projection functions.

4.3.2 Variables

Like values, variables are to be modeled as a sort in the meta notation. We introduce two sorts. The first sort $\mathcal{V}_{Lab}$ models the different variable labels. The second sort $\mathcal{V}$ models a variable, where the constructor function is used to indicate its sort, i.e., $\mathcal{V}$ models $\mathcal{X}^{\text{mCRL2}}$. The constructor's argument models the designated variable. By modeling the variables in this way we retain the option to model different typed variables in the meta notation. So, we introduce

$$\begin{array}{ll}\textbf{sort} & \mathcal{V}_{Lab} = \textbf{struct } v_1 \mid \ldots \mid v_n;\\ \textbf{sort} & \mathcal{V} \quad = \textbf{struct } \mathit{Sort}^1_{\mathcal{V}}(v_L{:}\mathcal{V}_{Lab})?\mathit{is}_{\mathit{Sort}_1} \mid \ldots \mid \mathit{Sort}^n_{\mathcal{V}}(v_L{:}\mathcal{V}_{Lab})\mathit{is}_{\mathit{Sort}_n};\end{array}$$

The elements in $\mathcal{V}_{Lab}$ and $\mathcal{V}$ are derived from the variables that occur in the original mCRL2 specification. Note that by modeling variables in this way we allow all variables that are in the Cartesian product of $\mathcal{V}_{Lab} \times \Lambda$. The excessive variables are mostly harmless in the meta notation as they are not used in the original mCRL2 specification.

Example 4.3.2. Let the mCRL2 specification define the sorts $\mathbb{B}$ and $\mathbb{N}$, with the variable $b{:}\mathbb{B}$ and $n{:}\mathbb{N}$. Then we define the structured sorts $\mathcal{V}$ and $\mathcal{V}_{Lab}$ as

sort $\mathcal{V}_{Lab} = \textbf{struct } b \mid n;$
sort $\mathcal{V} \quad = \textbf{struct } \mathbb{B}_{\mathcal{V}}(v{:}\mathcal{V}_{Lab}) \mid \mathbb{N}_{\mathcal{V}}(v{:}\mathcal{V}_{Lab});$

4.3.3 Mutable data valuation

A mutable data valuation is the data valuation used at the left and right side of the arrow in the deduction rules. The data valuation needs to be mutable because:

- the deduction rules for the parallel and synchronize operator merge the data valuation from the first premise σ' and the second premise σ''. Here the fields need to be unique with respect to the variables.
- the process definition adds new fields to the existing data valuation and substitutes variables in a data valuation. To add new fields, additional fresh variables are required, which are generated with the help of a data valuation.

To model a field, we introduce a sort $\mathcal{I}$. A field in the mutable data valuation is modeled as a tuple that consists of a variable and a value. The mutable data valuation $\mathcal{S}$ is modeled as an (ordered) list of fields.

sort $\mathcal{S} = \mathit{List}(\mathcal{I});$
sort $\mathcal{I} = \textbf{struct } \mathit{field}(\mathit{variable}{:}\mathcal{V}, \mathit{valvalue}{:}\Lambda);$

We assume that a $\mathcal{S}$ is an ordered list. However, adding or updating fields may cause the data valuation to become unordered. To correct any unordered lists we add the function $\mathcal{S}_{<}{:}\mathcal{S} \to \mathcal{S}$ that orders a mutable valuation list.

4.3.4 Functional data valuation

To compute the semantic interpretation of a syntactic multi-action we use a different representation of the data valuation. For that we use a functional description, i.e., a lambda abstraction with function updates. Here, the function updates model a field. To model this valuation we introduce a sort $\mathcal{S}_f$ that describes the signature of the function

sort $\mathcal{S}_f = \mathcal{V} \to \Lambda;$

The lambda abstraction that represents the functional data valuation is defined through

map $\sigma_f{:}\mathcal{S}_f;$
eqn $\sigma_f = \lambda v{:}\mathcal{V}.\bot;$

All (non-defined) variables are mapped to our special undefined value $\bot$. With the help of function updates, variables are mapped to their respective values.

Example 4.3.3. Let $v{:}\mathcal{V}$ and $\Lambda = \mathbb{N}$. Then the assignment of the value 5 to variable v can be defined as: $\sigma_f[v \mapsto 5]$.

4.3.5 Data valuation conversion

As the functional data valuation is used internally, and the mutable data valuation is used 'externally', we convert the mutable data valuation to the functional data valuation at some point. To achieve this we introduce the function *ToInternalValuation*.

map $ToInternalValuation{:}\mathcal{S} \to \mathcal{S}_f$;
$ToInternalValuation{:}\mathcal{S} \times \mathcal{S}_f \to \mathcal{S}_f$;
var $as{:}\mathcal{I}$;
$lass{:}\mathcal{S}$;
$\sigma{:}\mathcal{S}_f$;
eqn $ToInternalValuation(lass) = ToInternalValuation(lass, \sigma_f)$;
$ToInternalValuation([], \sigma) = \sigma$;
$ToInternalValuation(as \triangleright lass, \sigma) =$
$ToInternalValuation(lass, \sigma[variable(as) \mapsto valvalue(as)])$;

4.3.6 Discussion

The sorts that are required to model variables, values and data valuations in the meta notation depend on each other. Figure 4.1 illustrates the relations and dependencies among them. Here, rectangles denote the modeled sorts. An inverted open white arrow denote the inclusion of all the sorts at the origin of the arrow. Dashed arrows denote the transformation between sorts. Arrows, with black arrowheads and dots, express that a sort can be constructed from one of the incoming arrows, iff all sorts attached at the side of the dot are included. This means that if we want to construct $\mathcal{S}$, we require the sort $\mathcal{I}$ and can transform $\mathcal{S}$ into $\mathcal{S}_f$, with the help of function *ToInternalValuation*.

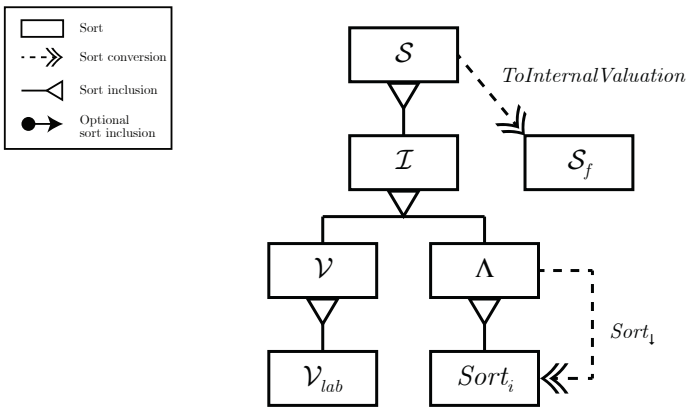


Figure 4.1: A graphical relationship between the modeled sorts

Since the data valuation could be modeled in different ways, we provide alternative methods and explain the rationale for our chosen solution.

In deduction rule par_8 and $sync_4$ we observe that the conclusion unifies the data valuations from the premises. Assume that σ' and σ'' are both modeled in the functional representation. Then we can distinguishable variables in the meta

notation by:

$$if(\sigma'(v) \neq \bot, \sigma'(v), \sigma''(v))$$

Hereby we informally state that if a variable is not defined by the first data valuation, it must be defined by the second valuation. To collect duplicate variables (e.g. to determine the duplicate generated fresh variables) we can construct a collection function like *col*

map $col{:}\mathcal{S}_f \times \mathcal{S}_f \to Set(\mathcal{V})$;
var $f, g{:}\mathcal{S}_f$;
eqn $col(f,g) = \{v{:}\mathcal{V} \mid f(v) \not\approx g(v)\}$;

The rename of variables inside a valuation can then simply be modeled as a preprocessing function on variables, i.e., a lambda abstraction that renames relevant variables. These concepts can all be modeled using the functional representation. However, the generation of fresh variables is somewhat problematic as we generate a new variable for every duplicate variable. For that, we require enumeration over the set of duplicate variables. As the tools within the toolset cannot enumerate sets we are unable to generate fresh variables. Hence, we decided to model two kinds of data valuations.

We model a mutable data valuation as a tuple, i.e., a variable and a value that are both constructed from a meta notated sort. Although we do not allow occurrences of "$\mathbb{B}_{\mathcal{V}}(v) \mapsto \mathbb{N}_{\Lambda}(0)$" in the meta notation, they are mCRL2 type correct. Similarly, we can construct variables that are not in the mCRL2 specification but are in the meta notation, resulting from the Cartesian product. Alternatively, we could have defined a separate data validation for each sort. This would lead to a more concise notation. However, by modeling it like we did, we illustrate that we can potentially deal with ill-defined semantics.

Furthermore, we assume that all variables are semantically evaluated to a value that is not ill-defined. As variables that cannot be semantically evaluated are represented by the value $\bot$, this could pose a problem in the communication. If we perform a communication $\{a \mid b \to c\}$, for the process $a(v_1) \mid b(v_2)$, for which the variables v_1, v_2 are undefined, then the above would reduce to $c(\bot)$. Obviously this is undesired behavior, since we cannot tell that the value of v_1 is and should be equal to the value of v_2. This could have been resolved by redefining $\bot$ with a additional variable argument. Since we consider undefined variables as undesired behavior, we decide not to incorporate this change.

4.4 Data expressions

Data expressions describe functions on syntactic model variables. Within a mCRL2 specification, data expressions are used to express action data parameters, process parameter updates and conditional choices. Internally, data expressions are specified through abstract data types. This means that if we write a value, the value is internally represented by an application of constructor functions and variables. So, if we write the natural number 2, it will be represented internally as $successor(successor(zero))$. Here, $successor(n)$ and $zero$ are constructor functions for the build in sort $\mathbb{N}$. Like $\mathbb{N}$, other (basic) data types such as $\mathbb{Z}, \mathbb{B}, List, Set, \ldots$, are part of mCRL2.

Data expressions in the meta notation are modeled in a similar fashion as data expressions in mCRL2. By that, we mean that a data expression can be

modeled by a variable but also by a function (possibly) having some arguments. Furthermore, we provide the option to model values as data expressions. By adding this extension we are able to convert meta notation data expressions into mCRL2 data expressions, as shown in Chapter 4.4.3. Furthermore, they provide an elegant shorthand notation to represent values with respect to the notation in constructor functions.

To model a data expression, we introduce the sort $\mathcal{E}$

$$\begin{aligned}\textbf{sort}\quad \mathcal{E} = \textbf{struct}\ & \mathcal{E}_{\mathcal{V}}(dvr{:}\mathcal{V})?is^{\mathcal{V}}_{\mathcal{E}} \\ & \mid \mathcal{E}_{\Lambda}(dvl{:}\Lambda)?is^{\Lambda}_{\mathcal{E}} \\ & \mid \mathcal{E}^{0}_{expr}(f{:}\mathcal{F})?is^{expr_0}_{\mathcal{E}} \\ & \mid \mathcal{E}^{1}_{expr}(f{:}\mathcal{F}, expr_1{:}\mathcal{E})?is^{expr_1}_{\mathcal{E}} \\ & \mid \quad \vdots \qquad\qquad\qquad \ddots \\ & \mid \mathcal{E}^{n-1}_{expr}(f{:}\mathcal{F}, expr_1{:}\mathcal{E}, \ldots, expr_{n-1}{:}\mathcal{E})?is^{expr_{n-1}}_{\mathcal{E}}; \\ & \mid \mathcal{E}^{n}_{expr}(f{:}\mathcal{F}, expr_1{:}\mathcal{E}, \ldots, expr_n{:}\mathcal{E})?is^{expr_n}_{\mathcal{E}};\end{aligned}$$

where $\mathcal{E}_{\mathcal{V}}$ models a data expression being a variable, $\mathcal{E}_{\Lambda}$ models a data expression being a value and $\mathcal{E}^{i}_{expr}$ models a function for a function symbol f having arity i.

Example 4.4.1. This example models the Boolean variable b as a data expression variable in the meta notation.

$$\mathcal{E}_{\mathcal{V}}(\mathcal{V}_{\mathbb{B}}(b))$$

Example 4.4.2. This example models the Natural value 2 as a data expression variable in the meta notation.

$$\mathcal{E}_{\Lambda}(\Lambda_{\mathbb{N}}(2))$$

4.4.1 Data expression functions and function operators

Function operators in the meta notation are defined by the sort $\mathcal{F}$. A function operator may describe a label of a constructor function, but it can also describe a label of a mapping function. To preserve the well-typed-ness of the functions operators, every operator is typed. Function operators are typed in the same way as variables, that is, the structure sort $\mathcal{F}$ is constructed from an operator label sort $\mathcal{O}$. So, we define:

$$\begin{aligned}&\textbf{sort}\quad \mathcal{O} = \textbf{struct}\ O_1 \mid \ldots \mid O_n; \\ &\textbf{sort}\quad \mathcal{F} = \textbf{struct}\ Sort^{1}_{\mathcal{O}}(op{:}\mathcal{O})?is_{Sort_1} \mid \ldots \mid Sort^{n}_{\mathcal{O}}(op{:}\mathcal{O})?is_{Sort_n};\end{aligned}$$

Example 4.4.3. This example shows the way in which constructor functions can be specified. Assume that we want to model a sort that represents a color from a gray scale. Hereto we introduce sort *GrayScale* in the meta notation

$$\textbf{sort}\quad GrayScale;$$

To model the level of white, we assume two constructor functions *white* and *darker* that model the scale. These are modeled in the meta notation as:

$$\begin{aligned}&\textbf{sort}\quad \mathcal{O} = \textbf{struct}\ darker \mid white; \\ &\textbf{sort}\quad \mathcal{F} = \textbf{struct}\ GrayScale_{\mathcal{O}}(op{:}\mathcal{O})?is_{GrayScale};\end{aligned}$$

To model a date expression variable v belonging to the sort *GrayScale* we model

sort $\mathcal{V}_{Lab} = \textbf{struct } v$;
sort $\mathcal{V} = \textbf{struct } GrayScale_{\mathcal{V}}(\mathcal{V}_{Lab})$;

Then we are able to express:

- a *GrayScale* variable by: $\mathcal{E}_{\mathcal{V}}(GrayScale_{\mathcal{V}}(v))$,
- the white constructor function by: $\mathcal{E}^0_{expr}(GrayScale_{\mathcal{O}}(white))$
- the darker constructor function by: $\mathcal{E}^1_{expr}(GrayScale_{\mathcal{O}}(darker), \mathcal{E}_{\mathcal{V}}(GrayScale_{\mathcal{V}}(v)))$

4.4.2 Semantic interpretation of data expressions

The semantic interpretation of a data expression is represented by a value. To compute the semantic values we introduce function $sem_{\mathcal{E}}$.

map $sem_{\mathcal{E}}{:}\mathcal{E} \times \mathcal{S}_f \to \Lambda$;
var $vr{:}\mathcal{V}$;
$vl{:}\Lambda$;
$\sigma{:}\mathcal{S}_f$;
eqn $sem_{\mathcal{E}}(dvr(vr), \sigma) = \sigma(vr)$;
$sem_{\mathcal{E}}(dvl(vl), \sigma) = vl$;

Separate auxiliary functions are required to compute the semantic values for data expressions if they represent functions. These auxiliary functions extend the above equations, by adding extra data equations. We only add the data equations for the operators that are defined in the mCRL2 model.

Example 4.4.4. To illustrate the semantic evaluation of a function, we model the equality of the gray values. Hereto, we extend Example 4.4.3. The resulting type of the function is a Boolean. Sort $\mathbb{B}$ is a build-in/predefined sort, so we do not have to introduce it the sort in the specification. In the meta-notation we extend the sort Λ with the sort element $\mathbb{B}_\Lambda$.. Since $\approx$, like $\mathbb{B}$, is defined for *a*ll sorts, we write the semantic interpretation for the data expression as

var $expr_1, expr_2{:}\mathcal{E}$;
eqn $sem_{\mathcal{E}}(\mathcal{E}^2_{expr}(\mathcal{E}_{\mathcal{O}}(\text{“}\approx\text{”}), expr_1, expr_2), \sigma) = \mathbb{B}_\Lambda(sem_{\mathcal{E}}(expr_1, \sigma) \approx sem_{\mathcal{E}}(expr_2, \sigma)))$;

4.4.3 Casting data expressions between meta notation and mCRL2 data specification

Although it is possible to define for each modeled sort in the meta notation the corresponding rewrite rules, in practice this will turn out to be a time consuming task. To circumvent the specification of these rewrite rules we allow values to be used in meta notation data expressions. By allowing them, we create a notational short-hand for abstract data types.

As these values also come with predefined rewrite rules, and we want to avoid the (re)specification of this functionality in the meta notation, we provide casts from data expressions in the meta notation to data expressions in the mCRL2 language. This is accomplished using function $Sort_{\downarrow}{:}\Lambda \to Sort$, where $Sort \in \mathcal{S}^{\text{mCRL2}}$ denotes a sort in the mCRL2 specification. After a value has been casted to the mCRL2 language, it is possible to use the function/operations that are defined by the mCRL2 specification/language. To cast mCRL2 data expressions to the meta notation, we simply use the appropriate constructor from the sort Λ.

Example 4.4.5. In this example we show how we can express the addition on natural numbers in our meta notation. Let an mCRL2 model define the sort $\mathbb{N}$ and the operator symbols "+" (denoting the addition on Natural numbers). Now we can specify $\mathcal{O}$ and $\mathcal{F}$ as

sort $\mathcal{O} = \textbf{struct}$ " + ";
sort $\mathcal{F} = \textbf{struct}\ \mathbb{N}_{\mathcal{O}}(op{:}\mathcal{O})?is_{\mathbb{N}}$;

where an appropriate name is chosen for the projection function. To model the addition of the value 3 and the value 2 (i.e., the mCRL2 data expression $3 + 2$), we write in the meta notation

$$\mathcal{E}^2_{expr}(\mathbb{N}_{\mathcal{O}}(\text{" + "}),\ \mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(3)),\ \mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(2)))$$

Since sort $\mathbb{N}$ is already accompanied with a set of rewrite rules, which also cover the addition operator, we decide to use the internal rewrite rules. For that we cast sort $\mathbb{N}_{\Lambda}$ to $\mathbb{N}$. As such we specify function $\mathbb{N}_{\downarrow}{:}\Lambda \to \mathbb{N}$ as

map $\mathbb{N}_{\downarrow}{:}\Lambda \to \mathbb{N}$;
var $n{:}\mathbb{N}$;
eqn $\mathbb{N}_{\downarrow}(\mathbb{N}_{\Lambda}(v)) = n$;

Using this cast function we can provide the data equation

var $expr_1, expr_2{:}\mathcal{E}$;
eqn $sem_{\mathcal{E}}(\mathcal{E}^2_{expr}(\mathcal{E}_{\mathcal{O}}(\text{" + "}), expr_1, expr_2), \sigma) = \mathbb{N}_{\Lambda}(\mathbb{N}_{\downarrow}(sem_{\mathcal{E}}(expr_1, \sigma)) + \mathbb{N}_{\downarrow}(sem_{\mathcal{E}}(expr_2, \sigma))))$;

Then if we want to compute the semantic interpretation we get

$$sem_{\mathcal{E}}(\mathcal{E}^2_{expr}(\mathbb{N}_{\mathcal{O}}(\text{" + "}),\ \mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(3)),\ \mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(2))))$$

for some σ, the above function will return:

$$\mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(5))$$

4.5 Multi-actions

4.5.1 Syntactic and semantic actions

The mCRL2 language defines two kinds of actions. The first kind are the syntactic actions. The second kind are the semantic actions. To model these concepts we introduce two sorts, namely the sort Act_{Ξ}, to model syntactic actions and the sort Act_{Σ}, to models semantic actions.

The syntactic action specifies two constructors. The first constructor function is used to define an external syntactic action, i.e., an action consisting of an action label and (optional) data parameters. The first argument of the constructor function defines the action label, the second defines the action data parameters. Within a syntactic action every action parameter corresponds to a data expression. The second constructor function defines the internal syntactic action, i.e., τ.

sort $Act_{\Xi} = \textbf{struct}\ Act(actionlabel{:}Act_{Lab}, args{:}List(\mathcal{E})) \mid tau$;

For the semantic action we specify a sort with only one constructor function. This constructor function corresponds an external semantic action. A separate constructor function for an internal action is omitted, since the semantic multi-action equivalence class corresponds to the empty list. Hence, a semantic internal action can not exist in isolation. So, we specify

sort $Act_\Sigma = \textbf{struct}\ ActSem(actionlabel{:}Act_{Lab}, args{:}List(\Lambda))$;

Syntactic actions are transformed into semantic actions with the help of function sem_{Act_Ξ}.

map $sem_{Act_\Xi}{:}Act_\Xi \times \mathcal{S}_f \to Act_\Sigma$;
var $\sigma{:}\mathcal{S}_f$;
$a{:}Act_{Lab}$;
$args{:}List(\mathcal{E})$;
eqn $sem_{Act_\Xi}(Act(a, args), \sigma) = ActSem(a, sem_{\mathcal{E}}^{List}(args, \sigma))$;

Note that we do not facilitate a data equation for $sem_{Act_\Xi}(tau)$. The equation is intentionally not specified, since the function that describes the syntactic multi-action interpretation (Chapter 4.5.2), removes all occurrences of tau such that the semantic equivalence class of an internal action $\tau_\sim$ is represented by the empty list.

The semantic interpretations are performed on the action parameters. To transform these data expressions into a list of semantic values, we introduce function $sem_{\mathcal{E}}^{List}$.

map $sem_{\mathcal{E}}^{List}{:}List(\mathcal{E}) \times \mathcal{S}_f \to List(\Lambda)$;
var $\sigma{:}\mathcal{S}_f$;
$des{:}List(\mathcal{E})$;
$de{:}\mathcal{E}$;
eqn $sem_{\mathcal{E}}^{List}([], \sigma) = []$;
$sem_{\mathcal{E}}^{List}(de \triangleright des, \sigma) = sem_{\mathcal{E}}(de, \sigma) \triangleright sem_{\mathcal{E}}^{List}(des, \sigma)$;

4.5.2 Syntactic multi-action and semantic multi-action equivalence classes

The meta notation represents both syntactic multi-actions and semantic multi-action equivalence classes as lists. In the list, every element corresponds to an action. The syntactic multi-action can (and may be) specified as an unordered list. For the semantic multi-action equivalence class, we assume that the list is presented in normal form, i.e., an ordered list. The ordered list then represents all members of the class, i.e., all possible lists that after ordering are represented by the normal form. To test and construct semantic multi-action equivalence classes that can be ordered, we introduce two functions.

To test if a semantic multi-action equivalence class is ordered we specify the function $Act_{\Sigma<}^{?}{:}List(Act_\Sigma) \to \mathbb{B}$. The function returns *true* if the list is ascending and *false* otherwise. The test is required to restrict the list of semantic actions within the body of some of the set comprehensions that could not be computed otherwise.

map $Act_{\Sigma<}^{?}{:}List(Act_\Sigma) \to \mathbb{B}$;
var $x, y{:}Act_\Sigma$;
$xs{:}List(Act_\Sigma)$;
eqn $Act_{\Sigma<}^{?}([]) = true$;
$Act_{\Sigma<}^{?}(x \triangleright []) = true$;
$Act_{\Sigma<}^{?}(x \triangleright y \triangleright xs) = (x \leq y) \wedge\ Act_{\Sigma<}^{?}(y \triangleright xs)$;

To construct an ordered list of semantic multi-actions we specify the function $Act_{\Sigma<}{:}List(Act_\Sigma) \to List(Act_\Sigma)$. The function is required since some deduction rules can potentially reorder the list of semantic actions, e.g. ma in Table 2.1,

deduction rules $par_5, \ldots, par_8$ and deduction rules ren_1 and ren_2 in Table 2.5. The function orders a list of semantic with an implementation of the merge sort algorithm, and a predicate for sorting. The insertion of a field is performed with the help of the auxiliary function *InsAct*.

map $Act_{\Sigma<}$:$List(Act_\Sigma) \to List(Act_\Sigma)$;
$Act_{\Sigma<}$:$(Act_\Sigma \times Act_\Sigma \to \mathbb{B}) \times List(Act_\Sigma) \to List(Act_\Sigma)$;
var x:Act_Σ;
xs:$List(Act_\Sigma)$;
$pred$:$Act_\Sigma \times Act_\Sigma \to \mathbb{B}$;
eqn $Act_{\Sigma<}(xs) = Act_{\Sigma<}(\lambda i, j{:}Act_\Sigma.(i \leq j), xs)$;
$Act_{\Sigma<}(pred, []) = []$;
$Act_{\Sigma<}(pred, x \triangleright xs) = InsAct(pred, x, Act_{\Sigma<}(pred, xs))$;

map $InsAct$:$(Act_\Sigma \times Act_\Sigma \to \mathbb{B}) \times Act_\Sigma \times List(Act_\Sigma) \to List(Act_\Sigma)$;
var x, y:Act_Σ;
ys:$List(Act_\Sigma)$;
$pred$:$Act_\Sigma \times Act_\Sigma \to \mathbb{B}$;
eqn $InsAct(pred, x, []) = [x]$;
$pred(x, y) \to InsAct(pred, x, y \triangleright ys) = x \triangleright y \triangleright ys$;
$(\neq pred(x, y)) \to InsAct(pred, x, y \triangleright ys) = y \triangleright InsAct(pred, x, ys)$;

4.5.3 Interpreting syntactic multi-actions into multi-action equivalence classes

To compute the semantic multi-action equivalence class for a syntactic multi-action, we introduce the function $sem^{List}_{Act_\Xi}$:$List(Act_\Xi) \times \mathcal{S}_f \to List(Act_\Sigma)$. This function takes a syntactic multi-action (e.g. a list of syntactic actions) and produces a semantic multi-action equivalence class (e.g. a list of semantic actions) using the functional data valuation. The ordering of the semantic multi-actions needs to be performed separately afterwards.

A multi-action consists of actions, and an action consists of a label and a list of data expressions. Since we specify syntactic actions and observe semantic actions, we need a function that can converts the actions according to Definition 2.2.16. So we specify:

map $sem^{List}_{Act_\Xi}$:$List(Act_\Xi) \times \mathcal{S}_f \to List(Act_\Sigma)$;
sem_{Act_Ξ}:$Act_\Xi \times \mathcal{S}_f \to Act_\Sigma$;
var as:$List(Act_\Xi)$;
a:Act_Ξ;
σ:$\mathcal{S}_f$;
eqn $sem^{List}_{Act_\Xi}([], \sigma) = []$;
$sem^{List}_{Act_\Xi}(a \triangleright as, \sigma) = \quad if(a \approx tau, sem^{List}_{Act_\Xi}(as, \sigma),$
$sem_{Act_\Xi}(a, \sigma) \triangleright sem^{List}_{Act_\Xi}(as, \sigma))$;

4.5.4 Discussion

The sorts that are required to model syntactic multi-actions, the sorts that are required to transform them into semantic multi-action equivalence classes, and their mutual relationship are depicted in Figure 4.2. The interpretation of the relationships between the sorts is expressed in the same way as in Figure 4.1 in Chapter 4.3.6. Note that here $\mathcal{E}$ can either be created by a sort $\mathcal{V}$, a sort Λ, or a combined sort $\mathcal{F}$ and $\mathcal{E}$.

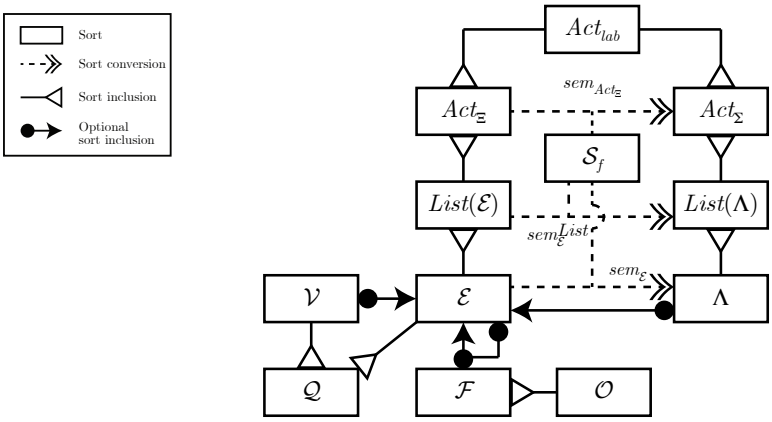


Figure 4.2: A graphical relationship between the modeled sorts supporting actions

As multi-actions are basically a bag of actions, it would have been a more natural choice to model multi-actions as such. If we model multi-actions as a bag in the mCRL2 toolset we require a function, say f, with the signature:

$$Bag(Act_\Xi) \times \mathcal{S}_f \rightarrow Bag(Act_\Sigma)$$

that interprets a bag of syntactic multi-actions and transforms them into a new bag semantic multi-actions. Ideally, f is performed as a map on each of the elements of the bag. Since we deal with infinite bags, we cannot apply f to individual elements. This poses a problem. The characterization of bags is chosen as a function, with function updates to model exceptions. Instead, we define the resulting set comprehension with the help of f^{-1}. This requires that the relation is a bijection. As the semantic interpretation function is not we cannot define the inverse. For that reason we choose to model the multi-action as lists of actions, as they allow enumeration.

4.6 Linear Process Equation representation

A state in an mCRL2 model is represented by a process term p and a data valuation σ. To store the current state in the meta notation we introduce process X. With the help of its process parameter we store the current values of the meta notated term p and the meta notated valuation σ. The first parameter corresponds to the process term p and the second to the data valuation σ. Process X will also implement the transformation relations.

To describe the action transition relation for a state (p, σ), i.e., $(p, \sigma) \xrightarrow{a} (p', \sigma')$ we (i) model the transition relation, and (ii) the signature that captures the transition and the corresponding resulting state.

To model (i) we introduce an action, that decorates the semantic multi-action equivalence class. Since we cannot use the meta notation directly as a transition, we capture the representation of the class as a data parameter of an action. The

action then represents the transition relation. So, we declare and use the mCRL2 action $\underline{t}_{\rightarrow}:List(Act_{\Sigma}) \in \mathcal{A}^{\mathrm{mCRL2}}$.

To model (ii) we first design a signature that captures a solution of an action transition. The solution must consist of a multi-action, a process term and data valuation. So we model the signature by the following sort:

sort $\quad \mathcal{R}_{at} = \textbf{struct}\ at(ac{:}List(Act_{\Sigma}), \pi_t{:}\mathcal{P}, \sigma'{:}\mathcal{S});$

where at is the constructor function for a solution, argument ac denotes the multi-action equivalence class, argument π_t denotes the updated process term and argument σ' denotes the updated data valuation.

The possible transitions that can be taken from a state are represented by a set of solutions. To compute the set, we introduce function $R{:}\mathcal{P} \times \mathcal{S} \rightarrow Set(\mathcal{R}_{at})$. This function specifies the union over the solutions of the individual deduction rules.

map $\quad R, R_{alpha}, R_{alt_1}, \ldots, R_{Def_1}, R_{Def_2}{:}\mathcal{P} \times \mathcal{S} \rightarrow Set(\mathcal{R}_{at});$
var $\quad p{:}\mathcal{P};$
$\quad\quad s{:}\mathcal{S};$
eqn $\quad R(p,s) = R_{alpha}(p,s) \cup R_{alt_1}(p,s) \cup \ldots \cup R_{Def_1}(p,s) \cup R_{Def_2}(p,s);$

Since only applicable functions return a non-empty set, and non applicable function return an empty set, we only compute for the solutions for the deduction rules that hold. The implementation of the individual deduction rules is discussed in Chapter 5.

Together with the information that is provided above, we denote the specification of the LPEas:

proc $\quad X(p{:}\mathcal{P}, s{:}\mathcal{S}) = \sum_{r:\mathcal{R}_{at}} (r \in R(p,s)) \rightarrow \underline{t}_{\rightarrow}(ac(r)) \cdot X(\pi_t(r), \sigma'(r))$

To initialize the LPS, we write

init $\quad X(p_0, \sigma_0);$

where p_0 denotes the initial process term (i.e., the model) in meta notation and σ_0 is a mutable data valuation.

Chapter 5

Data equations for deduction rules

This section describes the data equations that result from the transformation of the deduction rules. For each of the data equations we assume that their computation is performed with process term $p{:}\mathcal{P}$ and the mutable data valuation $s{:}\mathcal{S}$. Design decisions that have been taken and assumptions that have been made are explicitly described. For presentation purposes we introduce a separate data equation for each of the deduction rules. This improves the readability when evaluating the deduction rules.

5.1 Deadlock

A deadlock within an mCRL2 model can be modeled using the process term δ. In the meta notation we use the notation *deadlock* to model this concept. As this term does not have any deduction rules, we do not require any data equations to model them.

5.2 Multi-actions

By performing a syntactic multi-action α, we change the state of the model. In the meta notation we write a syntactic multi-action as

$$alpha([a_1, \ldots, a_n])$$

where $a_i, \ldots, a_n \in Act_\Xi$.

With help of function R_α, we compute the set of solutions that correspond to the deduction rule of a multi-action (Table 2.1, rule *ma*). The set is computed by set comprehension, for which we only allow solutions a iff:

- the input term p is an action process term ($is_\alpha(p)$),
- the semantic multi-action corresponds to the syntactic multi-action, evaluate under the data valuation and ordered subsequently ($ac(a) \approx Act_{\Sigma<}(sem(multiaction(p), s))$),

- the process term mentioned in the solution denotes a successful termination ($is_{\checkmark}(\pi_t(a))$, and
- the data valuation remains unchanged ($\sigma'(a) \approx s$)

With help of the above conditions we express the corresponding data equation as:

eqn $R_\alpha(p,s) = if(is_\alpha(p), \{a{:}\mathcal{R}_{at} \mid ac(a) \approx Act_{\Sigma<}(sem(multiaction(p),s)) \wedge is_{\checkmark}(\pi_t(a)) \wedge \sigma'(a) \approx s\}, \emptyset);$

5.3 Alternative composition

The alternative composition operator $p + q$, allows a non-deterministic choice when both process p as well as process q can perform a transition. In the meta notation we write the alternative composition as

$$alt(p,q)$$

where p and q are process terms in meta notation.

The deduction rules for this operator are provided in Table 2.1 by rules alt_1, alt_2, alt_3 and alt_4. To compute the corresponding solutions, we specify respectively the functions R_{alt_1}, R_{alt_2}, R_{alt_3} and R_{alt_4}. The resulting data equations produce a non-empty solution set if either process term p or process term q can perform a transition. To capture the deduction rules we write the following four equations:

eqn
$R_{alt_1}(p,s) = if(is_{alt}(p), \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_1(p),s) \wedge is_{\checkmark}(\pi_t(a)) \wedge \sigma'(a) \approx s\}, \emptyset);$
$R_{alt_2}(p,s) = if(is_{alt}(p), \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_1(p),s) \wedge \neg is_{\checkmark}(\pi_t(a))\}, \emptyset);$
$R_{alt_3}(p,s) = if(is_{alt}(p), \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_2(p),s) \wedge is_{\checkmark}(\pi_t(a)) \wedge \sigma'(a) \approx s\}, \emptyset);$
$R_{alt_4}(p,s) = if(is_{alt}(p), \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_2(p),s) \wedge \neg is_{\checkmark}(\pi_t(a))\}, \emptyset);$

5.4 Sequential composition

The sequential composition operator, denoted as $p \cdot q$, has two deduction rules for expressing the behavior of the sequential composition. Rule seq_1 in Table 2.1 expresses the successful termination of p, whereas rule seq_2 expresses the continuation as $p' \cdot q$ after performing an action by p. Then the syntax of the sequential composition is in the meta notation expressed through

$$seq(p,q)$$

where p and q are process terms in meta notation.

Since no explicit assumptions have been made, the modeling of the deduction rules is straightforward. Note that for R_{seq_2} we demand that the signature of the resulting process term is a sequential composition again ($is_{seq}(\pi_t(a))$).

eqn
$R_{seq_1}(p,s) = if(is_{seq}(p), \{a{:}\mathcal{R}_{at} \mid at(ac(a), \checkmark_{\mathrm{p}}, \sigma'(a)) \in R(\pi_1(p),s) \wedge \pi_t(a) \approx \pi_2(p) \wedge \sigma'(a) \approx s\}, \emptyset);$
$R_{seq_2}(p,s) = if(is_{seq}(p), \{a{:}\mathcal{R}_{at} \mid is_{seq}(\pi_t(a)) \wedge at(ac(a), \pi_1(\pi_t(a)), \sigma'(a)) \in R(\pi_1(p),s) \wedge \pi_2(\pi_t(a)) \approx \pi_2(p) \wedge \neg is_{\checkmark}(\pi_1(\pi_t(a)))\}, \emptyset);$

5.5 Conditional choice

The conditional choices $c \to p$ and $c \to p \diamond q$ allow the execution of behavior with respect to the result of the evaluation of the corresponding condition. The first operator only executes the behavior of body p if data expression c evaluates to *true*. The second operator has two bodies, i.e., p and q, for which the first body p is only allowed to execute iff the data expression evaluates c to *true*. The second body q is only allowed to execute iff the data expression c evaluates to *false*. In the process terms the first operator is represented by:

$$cond1(c, p)$$

The second operator is represented by:

$$cond2(c, p, q)$$

In the above, $c{:}\mathcal{E}$ is a data expression in meta notation and p, q are meta notation process terms.

Both operators contain a syntactic Boolean data expression that need a semantic interpretation. Therefore we compute the semantic value for $C(p)$ (the projection function C applied to process term p) under the mutable data valuation s, with the help of functions $sem_{\mathcal{E}}$ and *ToInternalValuation*. Since the condition is expressed in the meta notation and we evaluate the expression on the level of the mCRL2 data specification, we cast the semantic value with the help of $\mathbb{B}_{\downarrow}$.

The first operator $c \to p$ corresponds to the deduction rules $cond_1$ and $cond_2$ in Table 2.1. For these rules, we provide the following two equations

$$\begin{aligned}
\textbf{eqn}\quad & R_{cond1_1}(p,s) = if(is_{cond1}(p) \wedge\ \mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(C(p), ToInternalValuation(s))),\\
& \qquad \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_1(p), s) \wedge\ is_{\checkmark}(\pi_t(a)) \wedge\ \sigma'(a) \approx s\}, \emptyset);\\
& R_{cond1_2}(p,s) = if(is_{cond1}(p) \wedge\ \mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(C(p), ToInternalValuation(s))),\\
& \qquad \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_1(p), s) \wedge\ \neg is_{\checkmark}(\pi_t(a))\}, \emptyset);
\end{aligned}$$

For the second operator $c \to p \diamond q$, matching the deduction rules $cond_1$', ..., $cond_4$' in Table 2.1, we provide the following four equations

$$\begin{aligned}
\textbf{eqn}\quad & R_{cond2_1}(p,s) = if(is_{cond2}(p) \wedge\ \mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(C(p), ToInternalValuation(s))),\\
& \qquad \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_1(p), s) \wedge\ is_{\checkmark}(\pi_t(a)) \wedge\ \sigma'(a) \approx s\}, \emptyset);\\
& R_{cond2_2}(p,s) = if(is_{cond2}(p) \wedge\ \mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(C(p), ToInternalValuation(s))),\\
& \qquad \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_1(p), s) \wedge\ \neg is_{\checkmark}(\pi_t(a))\}, \emptyset);\\
& R_{cond2_3}(p,s) = if(is_{cond2}(p) \wedge\ \neg\mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(C(p), ToInternalValuation(s)))\\
& \qquad , \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_2(p), s) \wedge\ is_{\checkmark}(\pi_t(a)) \wedge\ \sigma'(a) \approx s\}, \emptyset);\\
& R_{cond2_4}(p,s) = if(is_{cond2}(p) \wedge\ \neg\mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(C(p), ToInternalValuation(s)))\\
& \qquad , \{a{:}\mathcal{R}_{at} \mid a \in R(\pi_2(p), s) \wedge\ \neg is_{\checkmark}(\pi_t(a))\}, \emptyset);
\end{aligned}$$

5.6 Sum operator

The sum operator $\sum_{v:D} p$ specifies the enumeration of values over the domain of a sort D and assigns the values to variable v. Under the selected values, the execution of process p is performed. To model the sum operator in the meta notation we express

$$Sum(v, p)$$

where $v{:}\mathcal{V}$ is a (typed) variable and p is a process term, both expressed in the meta notation.

Although it is illegal to write "$2 \approx true$" in an mCRL2 specification, it is allowed to write such an expression in meta notation

$$\mathcal{E}^2_{expr}(\mathbb{N}_{\mathcal{O}}(\text{“}\approx\text{”}),\ \mathcal{E}_\Lambda(\mathbb{B}_\Lambda(true)),\ \mathcal{E}_\Lambda(\mathbb{N}_\Lambda(2)))$$

According to the mCRL2 typing rules [23] the meta notation is well-typed. As the sum operator generates values in meta notation we restrict the domain of the generated values. They are restricted such that they correspond to the sort of the meta notation variable. To restrict the domain, we introduce the domain restriction function $M_D.$ that basically models $e \in \mathcal{M}_D$ of rules sum_1 and sum_2. So, if we have a model that specifies sorts $Sort_1 \ldots, Sort_n$, we introduce for each sort a conjunct in the data equations of the restriction function. Now the specification of M_D becomes

map $M_D{:}\mathcal{V} \times \Lambda \to \mathbb{B}$;
var $v{:}\mathcal{V}$;
$w{:}\mathcal{V}$;
eqn $M_D(v,w) = (is_{Sort_1}(v) \wedge\ is_{Sort_1}(w)) \vee \ldots \vee (is_{Sort_n}(v) \wedge\ is_{Sort_n}(w))$;

The enumeration is accomplished by defining an existential quantifier in the body of the set comprehension. To update the data valuation s with the variables $d(p)$ and the accompanied found value v we invoke function $\mathcal{S}_{\circlearrowleft}{:}\mathcal{I} \times \mathcal{S} \to \mathcal{S}$, that produces an updated data valuation. The auxiliary function $\mathcal{S}_{\circlearrowleft}$ updates the fields in a mutable data valuation. For non existing fields it adds the variable to the value mapping whereas for existing fields it overwrites the existing fields. The function is described by the following set of equations

map $\mathcal{S}_{\circlearrowleft}{:}\mathcal{I} \times \mathcal{S} \to \mathcal{S}$;
var $a, b{:}\mathcal{I}$;
$bl{:}\mathcal{S}$;
eqn $\mathcal{S}_{\circlearrowleft}(a, []) = [a]$;
$\mathcal{S}_{\circlearrowleft}(a, b \triangleright bl) = if(variable(b) \approx variable(a), a \triangleright bl, b \triangleright \mathcal{S}_{\circlearrowleft}(a, bl))$;

Now, let p' be term that describes a sum operator in the meta notation. Then the enumeration variable is obtained by applying the projection function d to p', i.e., $d(p')$. To find the enumerated values that are valid for this variable we restrict the set of possible values $(M_D(d(p'), v))$.

The data equations that correspond to the rules sum_1 and sum_2 in Table 2.2 are given by

eqn $R_{sum_1}(p,s) = if(is_{sum}(p), \{a{:}\mathcal{R}_{at} \mid \sigma'(a) \approx s \wedge\ is_{\checkmark}(\pi_t(a))$
$\wedge\ \exists_{v:\Lambda} M_D(d(p), v)$
$(at(ac(a), \pi_t(a), Z) \in R(\pi_1(p), Z)$
whr $Z = \mathcal{S}_{\circlearrowleft}(field(d(p), v), s)$
end
$)\}, \emptyset)$;
$R_{sum_2}(p,s) = if(is_{sum}(p), \{a{:}\mathcal{R}_{at} \mid \neg is_{\checkmark}(\pi_t(a))$
$\wedge\ (\exists\ v{:}\Lambda. M_D(d(p), v)$
$\wedge\ a \in R(\pi_1(p), \mathcal{S}_{\circlearrowleft}(field(d(p), v), s)))\}, \emptyset)$;

5.7 Parallel operator

The parallel operator $p \parallel q$ denotes the concurrent execution of p and q. To model the operator in the meta notation we use the following syntax

$par(p, q)$

where p and q are process terms in meta notation.

The semantics of this operator correspond to the rules given in Table 2.3. Here the deduction rules par_1 to par_7 are modeled straightforward. The deduction rules par_8 requires auxiliary functions and is therefore explained separately. The corresponding data equations for the deduction rules par_1 to par_7 are:

eqn
$$R_{par_1}(p,s) = if(is_{par}(p), \{a{:}\mathcal{R}_{at} \mid at(ac(a), \checkmark_{\mathrm{p}}, s) \in R(\pi_1(p), s) \wedge\ \pi_t(a) \approx \pi_2(p) \wedge\ \sigma'(a) \approx s\}, \emptyset);$$
$$R_{par_2}(p,s) = if(is_{par}(p), \{a{:}\mathcal{R}_{at} \mid is_{par}(\pi_t(a)) \wedge\ at(ac(a), \pi_1(\pi_t(a)), \sigma'(a)) \in R(\pi_1(p), s) \wedge \neg is_{\checkmark}(\pi_1(\pi_t(a))) \wedge\ \pi_2(\pi_t(a)) \approx \pi_2(p)\}, \emptyset);$$
$$R_{par_3}(p,s) = if(is_{par}(p), \{a{:}\mathcal{R}_{at} \mid at(ac(a), \checkmark_{\mathrm{p}}, s) \in R(\pi_2(p), s) \wedge\ \pi_t(a) \approx \pi_1(p) \wedge\ \sigma'(a) \approx s\}, \emptyset);$$
$$R_{par_4}(p,s) = if(is_{par}(p), \{a{:}\mathcal{R}_{at} \mid is_{par}(\pi_t(a)) \wedge\ at(ac(a), \pi_2(\pi_t(a)), \sigma'(a)) \in R(\pi_2(p), s) \wedge \neg is_{\checkmark}(\pi_2(\pi_t(a))) \wedge\ \pi_1(\pi_t(a)) \approx \pi_1(p)\}, \emptyset);$$
$$R_{par_5}(p,s) = if(is_{par}(p), \{a{:}\mathcal{R}_{at} \mid \exists_{t_1,t_2:List(Act_\Sigma)} at(t_1, \checkmark_{\mathrm{p}}, s) \in R(\pi_1(p), s) \wedge at(t_2, \checkmark_{\mathrm{p}}, s) \in R(\pi_2(p), s) \wedge\ Act_{\Sigma<}(t_1 +\!\!+ t_2) \approx ac(a) \wedge\ \sigma'(a) \approx s \wedge\ is_{\checkmark}(\pi_t(a))\}, \emptyset);$$
$$R_{par_6}(p,s) = if(is_{par}(p), \{a{:}\mathcal{R}_{at} \mid \exists_{a_1,a_2:\mathcal{R}_{at}} a_1 \in R(\pi_1(p), s) \wedge\ is_{\checkmark}(\pi_t(a_1)) \wedge\ a_2 \in R(\pi_2(p), s) \wedge\ \neg is_{\checkmark}\pi_t(a_2) \wedge\ Act_{\Sigma<}(ac(a_1) +\!\!+ ac(a_2)) \approx ac(a) \wedge\ \pi_t(a_2) \approx \pi_t(a) \wedge\ \sigma'(a) \approx \sigma'(a_2)\}, \emptyset);$$
$$R_{par_7}(p,s) = if(is_{par}(p), \{a{:}\mathcal{R}_{at} \mid \exists_{a_1,a_2:\mathcal{R}_{at}} a_1 \in R(\pi_1(p), s) \wedge\ \neg is_{checkmark}(\pi_t(a_1)) \wedge\ a_2 \in R(\pi_2(p), s) \wedge\ is_{\checkmark}(\pi_t(a_2)) \wedge\ Act_{\Sigma<}(ac(a_1) +\!\!+ ac(a_2)) \approx ac(a) \wedge\ \pi_t(a_1) \approx \pi_t(a) \wedge\ \sigma'(a) \approx \sigma'(a_1)\}, \emptyset);$$

To model par_8 we require several auxiliary functions. The auxiliary functions, along with their descriptions and implementations are provide prior to the implementation of the actual deduction rule. So, we start with the auxiliary functions:

- $DuplicateVariablesInValuation{:}\mathcal{S} \times \mathcal{S} \to List(\mathcal{V})$. This function takes two mutable data valuations and computes the list of variables that occur on both.

  map
  $$DuplicateVariablesInValuation{:}\mathcal{S} \times \mathcal{S} \to List(\mathcal{V});$$
  $$DuplicateVariablesInValuation{:}\mathcal{S} \times Set(\mathcal{V}) \to List(\mathcal{V});$$
  var
  $$fas{:}\mathcal{S};$$
  $$as{:}\mathcal{S};$$
  $$a{:}\mathcal{I};$$
  $$vs{:}Set(\mathcal{V});$$
  $$DuplicateVariablesInValuation(as, fas) = DuplicateVariablesInValuation(as, GetVariablesInValuation(fas));$$
  $$DuplicateVariablesInValuation([], vs) = [];$$
  $$DuplicateVariablesInValuation(a \triangleright as, vs) = if(variable(a) \in vs, [variable(a)], [])+\!\!+ DuplicateVariablesInValuation(as, vs);$$

  The above function uses the function *GetVariablesInValuation* that takes a valuation and returns the set of occurring variables.

map $GetVariablesInValuation{:}\mathcal{S} \to Set(\mathcal{V})$;
var $a{:}\mathcal{I}$;
$as{:}\mathcal{S}$;
eqn $GetVariablesInValuation([]) = \emptyset$;
$GetVariablesInValuation(a \triangleright as) =$
$\{variable(a)\} + GetVariablesInValuation(as)$;

- $GenFreshVars{:}\mathbb{N} \times List(\mathcal{V}) \to List(\mathcal{V})$. This function generates a list of fresh variables. To generate fresh variables we require an identifier to make them unique. This identifier is represented by a natural number. We also require a list of variables to ensure that the fresh variables are well-typed. Fresh variables are prefixed with a label, i.e., d', that may only be used by the $GenFreshVars$ function. In this way the n^{th} generated fresh variable is represented by $d'(n)$.

map $GenFreshVars{:}\mathbb{N} \times List(\mathcal{V}) \to List(\mathcal{V})$;
var $vs{:}List(\mathcal{V})$;
$v{:}\mathcal{V}$;
$n{:}\mathbb{N}$;
eqn $GenFreshVars(n, []) = []$;
$GenFreshVars(n, v \triangleright vs) = GenFreshVar(v, n) \triangleright GenFreshVars(n, vs)$;

where $GenFreshVar$ is a model specific function that actually generates a fresh variable. This function is defined through

map $GenFreshVar{:}\mathcal{V} \times \mathbb{N} \to \mathcal{V}$;
var $l{:}\mathcal{V}_{Lab}$;
$id{:}\mathbb{N}$;
eqn $GenFreshVar(Sort^1_{\mathcal{V}}(l), id) = Sort^1_{\mathcal{V}}(d'(id))$;
$\vdots$
$GenFreshVar(Sort^n_{\mathcal{V}}(l), id) = Sort^n_{\mathcal{V}}(d'(id))$;

- $GetHighestId{:}\mathcal{S} \to \mathbb{N}$. This function computes the highest identifier number in a mutable data valuation.

map $GetHighestId{:}\mathcal{S} \to \mathbb{N}$;
$GetVarId{:}\mathcal{I} \to \mathbb{N}$;
var $as{:}\mathcal{S}$;
$a{:}\mathcal{I}$;
eqn $GetHighestId([]) = 0$;
$GetHighestId(a \triangleright as) = \max(GetVarId(a), GetHighestId(as))$;
$GetVarId(a) = if(is_{d'}(v_L(variable(a))), id(v_L(variable(a))), 0)$;

- $CreateVariableSubstitution{:}List(\mathcal{V}) \times List(\mathcal{V}) \to (List(\mathcal{V}) \to List(\mathcal{V}))$. This function takes two variable list and creates a variable substitution function. The first argument denotes the list variables that are substituted and the second argument is the list of new variables. Both lists must be of equal length and contain elements of the same sort.

map $\mathit{CreateVariableSubstitution}{:}\mathit{List}(\mathcal{V}) \times \mathit{List}(\mathcal{V}) \to (\mathcal{V} \to \mathcal{V})$;
$\mathit{CreateVariableSubstitution}{:}\mathit{List}(\mathcal{V}) \times \mathit{List}(\mathcal{V}) \times (\mathcal{V} \to \mathcal{V}) \to (\mathcal{V} \to \mathcal{V})$;
var $x{:}\mathcal{V}$;
$x'{:}\mathcal{V}$;
$xs{:}\mathit{List}(\mathcal{V})$;
$xs'{:}\mathit{List}(\mathcal{V})$;
$\rho{:}\mathcal{V} \to \mathcal{V}$;
eqn $\mathit{CreateVariableSubstitution}(xs, xs') =$
$\mathit{CreateVariableSubstitution}(xs, xs', \lambda v{:}\mathcal{V}.(v))$;
$\mathit{CreateVariableSubstitution}([], [], \rho) = \rho$;
$\mathit{CreateVariableSubstitution}(x \triangleright xs, x' \mid > xs', \rho) =$
$\mathit{CreateVariableSubstitution}(xs, xs', \rho[x \mapsto x'])$;

- $\mathit{VariableSubstitutionInValuationList}{:}(\mathit{List}(\mathcal{V}) \to \mathit{List}(\mathcal{V})) \times \mathcal{S} \to \mathcal{S}$. This function renames all variables in a mutable data valuation conform the given variable substitution function.

map $\mathit{VariableSubstitutionInValuationList}{:}(\mathcal{V} \to \mathcal{V}) \times \mathcal{S} \to \mathcal{S}$;
var $\rho{:}\mathcal{V} \to \mathcal{V}$;
$as{:}\mathcal{S}$;
$a{:}\mathcal{I}$;
eqn $\mathit{VariableSubstitutionInValuationList}(\rho, []) = []$;
$\mathit{VariableSubstitutionInValuationList}(\rho, a \triangleright as) =$
$\mathit{field}(\rho(\mathit{variable}(a)), \mathit{valvalue}(a)) \triangleright$
$\mathit{VariableSubstitutionInValuationList}(\rho, as)$;

- $\mathit{VariableSubstitutionInProcessTerm}{:}(\mathit{List}(\mathcal{V}) \to \mathit{List}(\mathcal{V})) \times \mathcal{P} \to \mathcal{P}$. This function renames all variables in a process term conform the given variable substitution function.

- $\mathit{ValuationMinusValuation}{:}\mathcal{S} \times \mathcal{S} \to \mathcal{S}$. This function takes two valuations and subtracts the fields from the second valuation from the first valuation.

map $\mathit{ValuationMinusValuation}{:}\mathcal{S} \times \mathcal{S} \to \mathcal{S}$;
var $x{:}\mathcal{I}$;
$xs{:}\mathcal{S}$;
$ys{:}\mathcal{S}$;
eqn $\mathit{ValuationMinusValuation}([], ys) = []$;
$\mathit{ValuationMinusValuation}(x \triangleright xs, ys) =$
$\mathit{if}(x \in ys, \mathit{ValuationMinusValuation}(xs, ys),$
$x \triangleright \mathit{ValuationMinusValuation}(xs, ys))$;

With the help of the aforementioned auxiliary functions above we construct a substitution function, which acts as input functions for both the functions *VariableSubstitutionInProcessTerm* and *VariableSubstitutionInValuationList*. Let $\sigma'(a1)$ and $\sigma'(a2)$ be the data valuations of respectively the premises on the left and right. Then the short-hand notation for the substitution function SUBST is defined through

$$
\begin{array}{lll}
\textbf{Subst} = & \mathit{CreateVariableSubstitution}(& \\
& \textbf{Dup}, \mathit{GenFreshVars}(& \\
& \max(\mathit{GetHighestId}(\sigma'(a1)), \mathit{GetHighestId}(\sigma'(a2))) + 1, \textbf{Dup}) & \\
&)\textbf{whr} \quad \textbf{Dup} & = \mathit{DuplicateVariablesInValuation}(\\
& & \mathit{ValuationMinusValuation}(\sigma'(a2), s), \\
& & \mathit{ValuationMinusValuation}(\sigma'(a1), s)) \\
\textbf{end} & &
\end{array}
$$

Basically, we first compute the lowest identifier in the mutable data valuations and identity the valuations that contain duplicate variables. For these valuations we subsequently compute a list of fresh variables. With the list of duplicate variables and the list of fresh generated variables we create the variable rename function. The rename function is then used to rename the variables in the right premise and the right process term. With the help of these functions, we provide the rewrite rule that corresponds to deduction rule par_8:

$$
\begin{array}{lll}
\textbf{eqn} & R_{par_8}(p,s) = \mathit{if}(\mathit{is}_{par}(p), \{a{:}\mathcal{R}_{at} \mid \mathit{Act}^{?}_{\Sigma<}(\mathit{ac}(a)) \wedge \ \mathit{is}_{par}(\pi_t(a)) & \\
& \quad \wedge \exists_{a1,a2:\mathcal{R}_{at}}(\mathit{Act}_{\Sigma<}(\mathit{ac}(a1) +\!\!+ \mathit{ac}(a2)) \approx \mathit{ac}(a) & \\
& \qquad \wedge \neg \mathit{is}_{\checkmark}(\pi_t(a1)) \wedge \ \neg \mathit{is}_{\checkmark}(\pi_t(a2)) \wedge \ a2 \in R(\pi_2(p), s) \wedge \ a1 \in R(\pi_1(p), s) & \\
& \qquad \wedge \pi_t(a1) \approx \pi_1(\pi_t(a)) & \\
& \qquad \wedge \ \mathit{VariableSubstitutionInProcessTerm}(\textsc{Subst}, \pi_t(a2)) \approx \pi_2(\pi_t(a)) & \\
& \qquad \wedge \sigma'(a) \approx \mathcal{S}_<(s +\!\!+ \ \mathit{VariableSubstitutionInValuationList}(\textsc{Subst}, & \\
& \qquad\quad \mathit{ValuationMinusValuation}(\sigma'(a2), s)) & \\
& \qquad) & \\
& \quad \textbf{whr} \quad \textsc{Subst} = & \mathit{CreateVariableSubstitution}(\\
& & \textsc{Dup}, \mathit{GenFreshVars}(\\
& & \max(\mathit{GetHighestId}(\sigma'(a1)), \mathit{GetHighestId}(\sigma'(a2))) + 1, \textsc{Dup}) \\
& &) \ \textbf{whr} \quad \textsc{Dup} \quad = \mathit{DuplicateVariablesInValuation}(\\
& & \qquad \mathit{ValuationMinusValuation}(\sigma'(a2), s), \\
& & \qquad \mathit{ValuationMinusValuation}(\sigma'(a1), s)) \\
& & \ \textbf{end} \\
& \quad \textbf{end} \ )\}, \emptyset); &
\end{array}
$$

5.8 Sync operator

The sync operator $p \mid q$ denotes the synchronized execution of the first action from both process terms p and q, where after the remainder of the process term behaves concurrently. The meta notation corresponds to

$$\mathit{sync}(p, q)$$

The deduction rules $\mathit{sync}_1 \ \ldots \mathit{sync}_4$ in Table 2.4 provide the corresponding semantics. As the semantics nearly describe the deduction rules for the parallel operator, and we already have discussed the design decisions for the unification of the data valuations in Chapter 5.7, we only provide the data equations.

$$
\begin{aligned}
\textbf{eqn}\quad & R_{sync_1}(p,s) = \mathit{if}(\mathit{is}_{sync}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad \exists_{a1,a2:\mathcal{R}_{at}} a1 \in R(\pi_1(p),s) \wedge\ a2 \in R(\pi_2(p),s) \wedge\ \mathit{is}_{\checkmark}(\pi_t(a1)) \\
& \quad\quad \wedge\ \mathit{is}_{\checkmark}(\pi_t(a2)) \wedge\ \mathit{Act}_{\Sigma<}(ac(a1) {+\!\!+} ac(a2)) \approx ac(a) \wedge\ \mathit{is}_{\checkmark}(\pi_t(a)) \\
& \quad\quad \wedge\ \sigma'(a) \approx s \wedge\ \mathit{Act}^{?}_{\Sigma<}(ac(a))\}, \emptyset); \\
& R_{sync_2}(p,s) = \mathit{if}(\mathit{is}_{sync}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad \exists_{a1,a2:\mathcal{R}_{at}} a1 \in R(\pi_1(p),s) \wedge\ a2 \in R(\pi_2(p),s) \wedge\ \neg\mathit{is}_{\checkmark}(\pi_t(a1)) \\
& \quad\quad \wedge\ \mathit{is}_{\checkmark}(\pi_t(a2)) \wedge\ \mathit{Act}_{\Sigma<}(ac(a1) {+\!\!+} ac(a2)) \approx ac(a) \wedge\ \pi_t(a) \approx \pi_t(a1) \\
& \quad\quad \wedge\ \sigma'(a1) \approx \sigma'(a) \wedge\ \mathit{Act}^{?}_{\Sigma<}(ac(a))\}, \emptyset); \\
& R_{sync_3}(p,s) = \mathit{if}(\mathit{is}_{sync}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad \exists_{a1,a2:\mathcal{R}_{at}} a1 \in R(\pi_1(p),s) \wedge\ a2 \in R(\pi_2(p),s) \wedge\ \mathit{is}_{\checkmark}(\pi_t(a1)) \\
& \quad\quad \wedge\ \neg\mathit{is}_{\checkmark}(\pi_t(a2)) \wedge\ \mathit{Act}_{\Sigma<}(ac(a1) {+\!\!+} ac(a2)) \approx ac(a) \wedge\ \pi_t(a) \approx pi_t(a2) \\
& \quad\quad \wedge\ \sigma'(a) \approx \sigma'(a1) \wedge\ \mathit{Act}^{?}_{\Sigma<}(ac(a))\}, \emptyset); \\
& \\
& R_{sync_4}(p,s) = \mathit{if}(\mathit{is}_{sync}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad \exists_{a1,a2:\mathcal{R}_{at}} a1 \in R(\pi_1(p),s) \wedge\ a2 \in R(\pi_2(p),s) \wedge\ \neg\mathit{is}_{\checkmark}(\pi_t(a1)) \\
& \quad\quad \wedge\ \neg\mathit{is}_{\checkmark}(\pi_t(a2)) \wedge\ \mathit{Act}_{\Sigma<}(ac(a1) {+\!\!+} ac(a2)) \approx ac(a) \wedge\ \mathit{is}_{par}(\pi_t(a)) \\
& \quad\quad \wedge\ \pi_1(\pi_t(a)) \approx \pi_t(a1) \wedge\ \mathit{Act}^{?}_{\Sigma<}(ac(a)) \\
& \quad\quad \wedge\ \mathit{VariableSubstitutionInProcessTerm}(\textsc{Subst}, \pi_t(a2)) \approx \pi_2(\pi_t(a)) \\
& \quad\quad \wedge\ \sigma'(a) \approx \mathcal{S}_<(s {+\!\!+}\ \mathit{VariableSubstitutionInValuationList}(\textsc{Subst}, \\
& \quad\quad\quad \mathit{ValuationMinusValuation}(\sigma'(a2), s)) \\
& \quad\quad)\\
& \quad \textbf{whr}\quad \textsc{Subst} = \mathit{CreateVariableSubstitution}(\\
& \quad\quad\quad\quad \textsc{Dup}, \mathit{GenFreshVars}(\\
& \quad\quad\quad\quad \max(\mathit{GetHighestId}(\sigma'(a1)), \mathit{GetHighestId}(\sigma'(a2))) + 1, \textsc{Dup}) \\
& \quad\quad\quad\quad) \textbf{ whr }\quad \textsc{Dup} \quad = \mathit{DuplicateVariablesInValuation}(\\
& \quad\quad\quad\quad\quad\quad\quad\quad \mathit{ValuationMinusValuation}(\sigma'(a2), s), \\
& \quad\quad\quad\quad\quad\quad\quad\quad \mathit{ValuationMinusValuation}(\sigma'(a1), s)) \\
& \quad\quad\quad\quad\quad\quad\quad\quad \textbf{end} \\
& \quad \textbf{end }))\}, \emptyset);
\end{aligned}
$$

5.9 Left merge operator

The left merge operator, denoted as $p \between q$ has two deduction rules for expressing that the behavioral on the left has to perform an action first, before the remainder executes concurrently. The first rule lmerge_1 in Table 2.4 expresses the successful termination of p after which the process behaves as q. The second rule lmerge_2 expresses the continuation of $p' \between q$ after performing an action by p. In the meta notation, we write the operator as

$$\mathit{lmerge}(p, q)$$

assuming that p, q are meta notation process terms. Since we do not make any explicit assumptions, modeling these rules is straightforward.

$$
\begin{aligned}
\textbf{eqn}\quad & R_{lmerge_1}(p,s) = \mathit{if}(\mathit{is}_{lmerge}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad\quad \mathit{at}(ac(a), \checkmark_{\mathrm{p}}, s) \in R(\pi_1(p),s) \wedge\ \pi_t(a) \approx \pi_2(p) \wedge\ \sigma'(a) \approx s\}, \emptyset); \\
& R_{lmerge_2}(p,s) = \mathit{if}(\mathit{is}_{seq}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad\quad \mathit{is}_{par}(\pi_t(a)) \wedge\ \mathit{at}(ac(a), \pi_1(\pi_t(a)), \sigma'(a)) \in R(\pi_1(p),s) \\
& \quad\quad \wedge\ \pi_2(\pi_t(a)) \approx \pi_2(p) \wedge\ \neg\mathit{is}_{\checkmark}(\pi_1(\pi_t(a)))\}, \emptyset);
\end{aligned}
$$

5.10 Allow operator

The allow operator permits within p only those semantic multi-action equivalence classes for which the corresponding action labels are defined in the set of multi-

action labels A. Rules $allow_1$ and $allow_2$ in Table 2.5 describe the semantics of the allow operator.

In the meta notation we write the operator as

$$Allow(A, p)$$

where $A{:}Set(Bag(Act_{Lab}))$ defines the set of permitted multi-action labels (represented by a bag of labels) and p defines the process term in meta notation.

Since the transition in the solution is described by a semantic multi-action equivalence class, we strip the data parameters from the multi-action. The function *actionlabels* removes the data parameters from a semantic multi-action equivalence class. This function corresponds to function $\underline{\alpha_\sim}$ in Definition 2.2.18.

map $labels{:}List(Act_\Sigma) \to Bag(Act_{Lab})$;
var $as{:}List(Act_\Sigma)$;
$a{:}Act_\Sigma$;
eqn $actionlabels([]) = []$;
$actionlabels(a \triangleright as) = \{actionlabel(a){:}1\} \cup actionlabels(as)$;

The operator always allows the internal actions ($\tau_\sim$). Therefore, we extend the set of multi-action labels with the empty list. To ensure that the semantic multi-action equivalence class, represented by ac, occurs in the set of allowed multi-actions labels $V(p) \cup \{\emptyset\}$, we state that the following condition must hold in the set comprehension

$$actionlabels(ac) \in (V(p) \cup \{\emptyset\})$$

The rest of the process term can be translated straightforward resulting in the following data equations

eqn $R_{allow_1}(p, s) = if(is_{allow}(p), \{a{:}\mathcal{R}_{at} \mid is_\checkmark(\pi_t(a)) \wedge\ a \in R(\pi_1(p), s)$
$\wedge\ actionlabels(ac(a)) \in (V(p) \cup \{\emptyset\}) \wedge\ \sigma'(a) \approx s\}, \emptyset)$;
$R_{allow_2}(p, s) = if(is_{allow}(p), \{a{:}\mathcal{R}_{at} \mid is_{allow}(\pi_t(a)) \wedge\ \neg is_\checkmark(\pi_1(\pi_t(a))) \wedge\ V(\pi_t(a)) \approx V(p)$
$\wedge\ at(ac(a), \pi_1(\pi_t(a)), \sigma'(a)) \in R(\pi_1(p), s) \wedge\ actionlabels(ac(a)) \in (V(p) \cup \{\emptyset\})\}, \emptyset)$;

5.11 Block operator

The block operator blocks all (multi)-actions for which an action label is defined in the set of blocking labels. The term representing the block operator consists of two arguments. The first argument represents the set of blocking action labels and the second argument defines the process term to which the encapsulation is applied. The set of deduction rules that go along are given in Table 2.5 by $encap_1$ and $encap_2$.

The block operator is in the meta notation written as

$$Block(bl, p)$$

where $bl{:}Set(Act_{Lab})$ is a set (of blocking) action labels and p is the process term in meta notation.

The set of blocking actions only defines the blocking action labels. To determine if part of a semantic multi-actions equivalence class occurs in the set of blocking labels, we perform an abstraction on the class. The abstraction is provided through the functions *labels* (computing the labels), the function *L2S* (transforming the multi-action label to a set of action labels), and the

intersection of blocking labels. The intersection with blocking labels needs to be empty in order to perform the semantic multi-actions equivalence class. Let p be a blocking process term, and let ac be the semantic multi-actions equivalence class, then the following condition must hold in the comprehension

$$Bag2Set(actionlabels(ac)) \cap (B(p)) \approx \emptyset$$

The function *Bag2Set* is an mCRL2 build-in function that converts a bag into a set. Through this auxiliary function we define the data equations that corresponds to deduction rules $encap_1$ and $encap_2$ in Table 2.5.

$$\begin{array}{ll}
\textbf{eqn} & R_{block_1}(p,s) = if(is_{block}(p), \{a{:}\mathcal{R}_{at} \mid is_{\checkmark}(\pi_t(a)) \\
& \quad \wedge\ a \in R(\pi_1(p),s) \wedge\ Bag2Set(actionlabels(ac(a))) \cap (B(p)) \approx \emptyset \wedge\ \sigma'(a) \approx s\}, \emptyset); \\
& R_{block_2}(p,s) = if(is_{block}(p), \{a{:}\mathcal{R}_{at} \mid is_{block}(\pi_t(a)) \\
& \quad \wedge\ \neg is_{\checkmark}(\pi_1(\pi_t(a))) \wedge\ B(\pi_t(a)) \approx B(p) \wedge\ at(ac(a), \pi_1(\pi_t(a)), \sigma'(a)) \in R(\pi_1(p),s) \\
& \quad \wedge\ Bag2Set(labels(ac(a))) \cap (B(p)) \approx \emptyset\}, \emptyset);
\end{array}$$

5.12 Action rename operator

The rename operator $\rho_R(p)$ renames (multi)-action labels as specified by a rename function R for a process term p. Here, R is the rename function from Definition 2.2.18. Within the meta notation we write the action rename operator as

$$Rename(R,p)$$

where $R{:}Act_{Lab} \to Act_{Lab}$ is the rename function and p is the process term in meta notation.

To model the rename function R we define an identity function and with help of function updates we model the action label renaming for the selective updates. So, if ID denotes the identity function on action labels

$$\begin{array}{ll}
\textbf{map} & ID{:}Act_{Lab} \to Act_{Lab}; \\
\textbf{var} & x{:}Act_{Lab}; \\
\textbf{eqn} & ID(x) = x;
\end{array}$$

then we write a rename the renaming as $[x_1 \mapsto y_1, \ldots x_n \mapsto y_n]$ such that R is defined through $ID[x_1 \mapsto y_1, \ldots x_n \mapsto y_n]$.

To perform the actual rename of the labels in a multi-action, i.e., $R \bullet (a)$, we introduce the function Act_{Rename}. The function requires an "action label-to-action label" (rename) function and a semantic multi-action equivalence class, and produces a semantic multi-action in which the action labels are renamed according to the rename function. The corresponding deduction rules that we introduce for this function are provided by

$$\begin{array}{ll}
\textbf{map} & Act_{Rename}{:}(Act_{Lab} \to Act_{Lab}) \times List(Act_{\Sigma}) \to List(Act_{\Sigma}); \\
\textbf{var} & f{:}Act_{Lab} \to Act_{Lab}; \\
& a{:}Act_{\Sigma}; \\
& as{:}List(Act_{\Sigma}); \\
\textbf{eqn} & Act_{Rename}(f, []) = []; \\
& Act_{Rename}(f, a \triangleright as) = ActSem(f(actionlabel(a)), args(a)) \triangleright Act_{Rename}(f, as);
\end{array}$$

If p is an action rename operator term and ac is a semantic multi-action then $Act_{Rename}(Ren(p), ac)$ returns a semantic multi-action the on which the action rename function $Ren(p)$ as been applied to ac.

With help of the Act_{Rename} function and an additional semantic multi-action ac' (to find a valid substitution), the rename function for deduction rule ren_1 and ren_2 in Table 2.5. The rules that correspond to the deduction rules are provided below

$$
\begin{aligned}
\textbf{eqn}\quad & R_{rename_1}(p,s) = if(is_{rename}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad is_{\checkmark}(\pi_t(a)) \wedge\ \exists_{ac':List(Act_\Sigma)} ac(a) \approx Act_{\Sigma<}(Act_{Rename}(Ren(p), ac')) \\
& \quad \wedge\ at(ac', \pi_t(a), s) \in R(\pi_1(p), s) \wedge\ \sigma'(a) \approx s\}, \emptyset); \\
& R_{rename_2}(p,s) = if(is_{rename}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad Ren(\pi_t(a)) \approx Ren(p) \wedge\ is_{rename}(\pi_t(a)) \wedge\ \neg is_{\checkmark}(\pi_1(\pi_t(a))) \\
& \quad \wedge\ \exists_{ac':List(Act_\Sigma)} ac(a) \approx Act_{\Sigma<}(Act_{Rename}(Ren(p), ac')) \\
& \quad \wedge\ at(ac', \pi_1(\pi_t(a)), \sigma'(a)) \in R(\pi_1(p), s)\}, \emptyset);
\end{aligned}
$$

5.13 Hide operator

The hide operator $\tau_I(p)$ hides all actions in a semantic equivalence class, for which the corresponding label occurs in the set of hiding labels I. A hide operator again has two arguments. The first argument defines action labels and the second argument defines the term to which the operator is applied. Within the meta notation this is expressed as

$$Hide(I, p)$$

where $I{:}Set(Act_{Lab})$ is the set of action labels and p is the process term.

Hiding actions in a semantic multi-action equivalence class is performed by the function Act_{Hide}. This function follows the definition of $\theta_I(\alpha_\sim)$ in Definition 2.2.18. The function requires two arguments, namely a set of action labels I and a semantic multi-action equivalence class $\alpha_\sim$ in which actions are hidden. The function basically removes an action from the semantic multi-action iff the label of that action also occurs in the set of the to be hidden action labels. The function Act_{Hide} is defined as

$$
\begin{aligned}
\textbf{map}\quad & Act_{Hide}{:}Set(Act_{Lab}) \times List(Act_\Sigma) \to List(Act_\Sigma); \\
\textbf{var}\quad & I{:}Set(Act_{Lab}); \\
& as{:}List(Act_\Sigma); \\
& a{:}Act_\Sigma; \\
\textbf{eqn}\quad & Act_{Hide}(I, []) = []; \\
& (actionlabel(a) \in I) \to Act_{Hide}(I, a \triangleright as) = Act_{Hide}(I, as); \\
& \neg(actionlabel(a) \in I) \to Act_{Hide}(I, a \triangleright as) = a \triangleright Act_{Hide}(I, as);
\end{aligned}
$$

Let p' be the hide rename operator term and ac be a semantic multi-action equivalence class then $Act_{Hide}(I(p'), ac)$ returns a semantic multi-action equivalence class in which the matching action labels of $I(p)$ are removed/hidden from ac. Since the semantic actions are provided in linearly ordered list, removing any element form that list will preserve the order. Hence, we do not order the semantic action list after performing this operation.

With help of the Act_{Hide} function and an additional semantic multi-action ac' (to find a valid substitution), the hide function for deduction rule $hide_1$ and $hide_2$ in Table 2.5 are specified by

$$
\begin{aligned}
\textbf{eqn}\quad & R_{hide_1}(p,s) = if(is_{hide}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad is_{\checkmark}(\pi_t(a)) \wedge\ \exists_{ac':List(Act_\Sigma)} \\
& \quad ac(a) \approx Act_{Hide}(I(p), ac') \wedge\ at(ac', \pi_t(a), s) \in R(\pi_1(p), s) \wedge\ \sigma'(a) \approx s\}, \emptyset); \\
& R_{hide_2}(p,s) = if(is_{hide}(p), \{a{:}\mathcal{R}_{at} \mid \\
& \quad I(\pi_t(a)) \approx I(p) \wedge\ is_{hide}(\pi_t(a)) \wedge\ \neg is_{\checkmark}(\pi_1(\pi_t(a))) \wedge\ \exists_{ac':List(Act_\Sigma)} \\
& \quad ac(a) \approx Act_{Hide}(I(p), ac') \wedge\ at(ac', \pi_1(\pi_t(a)), \sigma'(a)) \in R(\pi_1(p), s)\}, \emptyset);
\end{aligned}
$$

5.14 Prehide operator

The prehide operator $\Upsilon_U(p)$ prehides all action labels. That is, it removes all action data parameters from the actions and relabels the action labels to *int*) that are defined in the set of prehiding labels. A term representing the prehide operator has two arguments. The first argument defines the action labels of the semantic actions that are prehidden. The second argument defines the process to which the prehide operator is applied. Within the meta notation we write

$$Prehide(U, p)$$

where $U{:}Set(Act_{Lab})$ is the set of action labels that are prehidden for process term p.

Prehiding actions in a semantic multi-action equivalence class is performed by the function $Act_{Prehide}$. This function follows the definition of $\eta_U(\alpha_\sim)$ in Definition 2.2.18. The function requires two arguments, namely a set of action labels U and a semantic multi-action equivalence class $\alpha_\sim$. The function removes all data parameters and subsequently renames the action label to *int* for all the actions in the semantic multi-action equivalence class, iff the label of an action occurs in the set prehide action labels. The function $Act_{Prehide}$ is defined as

$$\begin{array}{ll}
\textbf{map} & Act_{Prehide}{:}Set(Act_{Lab}) \times List(Act_\Sigma) \to List(Act_\Sigma); \\
\textbf{var} & U{:}Set(Act_{Lab}); \\
 & as{:}List(Act_\Sigma); \\
 & a{:}Act_\Sigma; \\
\textbf{eqn} & Act_{Prehide}(U, []) = []; \\
 & (actionlabel(a) \in U) \to Act_{Prehide}(U, a \triangleright as) = ActSem(int, []) \triangleright Act_{Prehide}(U, as); \\
 & \neg(actionlabel(a) \in U) \to Act_{Prehide}(U, a \triangleright as) = a \triangleright Act_{Prehide}(U, as);
\end{array}$$

$$\begin{array}{ll}
\textbf{eqn} & R_{prehide_1}(p, s) = if(is_{prehide}(p), \{a{:}\mathcal{R}_{at} \mid is_\checkmark(\pi_t(a)) \wedge \ Act_\Sigma \overset{?}{<} (ac(a)) \\
 & \qquad \wedge \exists_{ac':List(Act_\Sigma)} ac(a) \approx Act_{\Sigma<}(Act_{Prehide}(U(p), ac')) \\
 & \qquad \wedge \ at(ac', \pi_t(a), s) \in R(\pi_1(p), s) \wedge \ \sigma'(a) \approx s\}, \emptyset); \\
 & R_{prehide_2}(p, s) = if(is_{prehide}(p), \{a{:}\mathcal{R}_{at} \mid U(\pi_t(a)) \approx U(p) \wedge \ Act_\Sigma \overset{?}{<} (ac(a)) \\
 & \qquad \wedge \ is_{prehide}(\pi_t(a)) \wedge \ \neg is_\checkmark(\pi_1(\pi_t(a))) \wedge \ \exists_{ac':List(Act_\Sigma)} \\
 & \qquad\quad ac(a) \approx Act_{\Sigma<}(Act_{Prehide}(U(p), ac')) \wedge \\
 & \qquad\quad at(ac', \pi_1(\pi_t(a)), \sigma'(a)) \in R(\pi_1(p), s)\}, \emptyset);
\end{array}$$

5.15 Communication operator

The communication operator $\Gamma_C(p)$ renames synchronizing actions C if all action labels occur in the multi-action and the data parameters of the actions all have the same semantic value, when executed by p. Within mCRL2, the communication is given by partial functions where the domain denotes the bag of action labels and the codomain denotes an action label (function γ_C in Definition 2.2.18). For each bag of action labels that occur in the multi-action, the matching multi-action is replaced with an action that has the same data parameter values and the action label of the codomain. To describe the communication we introduce a sort that models the communication

$$\textbf{sort} \quad \mathcal{C} = \textbf{struct}\ communication(C_{dom}{:}List(Act_{Lab}), C_{range}{:}Act_{Lab});$$

The bag of synchronizing action labels are specified with C_{dom}. The resulting action label is specified with C_{range}.

Within the meta notation we model a communication operator as

$$Comm(C^{List}_{comm}, p)$$

where C_{comm}^{List}:$List(\mathcal{C})$ denotes a list of communications in meta notation, and p denotes the process term in meta notation.

To compute the result of the synchronization we introduce the function Act_{Comm}, which implements the description given by the communication γ_C in Definition 2.2.18). The Act_{Comm} function takes a list a list of communications and a list of semantic actions and computes the new list of semantic actions.

map Act_{Comm}:$List(\mathcal{C}) \times List(Act_\Sigma) \to List(Act_\Sigma)$;
var as:$List(Act_\Sigma)$;
C_{comm}:$\mathcal{C}$;
C_{comm}^{List}:$List(\mathcal{C})$;
eqn $Act_{Comm}([], as) = as$;
$Act_{Comm}(C_{comm} \triangleright C_{comm}^{List}, as) =$
$Act_{Comm'}(C_{comm}, C_{comm}^{List}, as, f^{d2a}(as, \lambda x{:}List(\Lambda).\emptyset), L2B(C_{dom}(C_{comm})), [], [])$;

With the function f^{d2a}, we construct from the semantic actions, a mapping that relates the different data parameters values to bags of action labels. The mapping is represented by variable $d2a$.

map f^{d2a}:$List(Act_\Sigma) \times (List(\Lambda) \to Bag(Act_{Lab})) \to (List(\Lambda) \to Bag(Act_{Lab}))$;
var $d2a$:$List(\Lambda) \to Bag(Act_{Lab})$;
as:$List(Act_\Sigma)$;
a:Act_Σ;
eqn $f^{d2a}([], d2a) = d2a$;
$f^{d2a}(a \triangleright as, d2a) = f^{d2a}(as, d2a[args(a) \mapsto d2a(args(a)) \uplus \{actionlabel(a){:}1\}])$;

The bag of communicating action labels is computed with help of function $L2B$.

map $L2B$:$List(Act_{Lab}) \to Bag(Act_{Lab})$;
var a_{lab}:Act_{Lab}
c_{dom}:$List(Act_{Lab})$
eqn $L2B([]) = \emptyset$;
$L2B(a_{lab} \triangleright c_{dom}) = \{a_{lab}{:}1\} \uplus L2B(c_{dom})$;

With the auxiliary function $Act_{Comm'}$we inspect if for a given list of data parameters there exists a bag of labels that corresponds to the bag of communicating action labels.

map $Act_{Comm'}$:$\mathcal{C} \times List(\mathcal{C}) \times List(Act_\Sigma) \times (List(\Lambda) \to Bag(Act_{Lab}))$
$\times Bag(Act_{Lab}) \times List(Act_\Sigma) \times List(Act_\Sigma) \to List(Act_\Sigma)$;
var $d2a$:$List(\Lambda) \to Bag(Act_{Lab})$;
c_{comm}:$Bag(Act_{Lab})$;
$Act_{Result}, Act_{Remain}$:$List(Act_\Sigma)$;
C_{comm}:$\mathcal{C}$;
C_{comm}^{List}:$List(\mathcal{C})$;
as:$List(Act_\Sigma)$;
a:Act_Σ;
eqn $Act_{Comm'}(C_{comm}, C_{comm}^{List}, [], d2a, c_{comm}, Act_{Result}, Act_{Remain}) =$
$Act_{Result} +\!\!+ Act_{Comm}(C_{comm}^{List}, Act_{Remain})$;
$Act_{Comm'}(C_{comm}, C_{comm}^{List}, a \triangleright as, d2a, c_{comm}, Act_{Result}, Act_{Remain}) =$
$if(c_{comm} \subseteq d2a(args(a)),$
$Act_{Comm'}($ $C_{comm}, C_{comm}^{List}, Actions^-(C_{dom}(C_{comm}), a \triangleright as, args(a)),$
$f^{d2a}(Actions^-(C_{dom}(C_{comm}), a \triangleright as, args(a)), \lambda x{:}List(\Lambda).\emptyset),$
$c_{comm}, ActSem(C_{range}(C_{comm}), args(a)) \triangleright Act_{Result}, Act_{Remain})$
$, Act_{Comm'}(C_{comm}, C_{comm}^{List}, as, d2a, c_{comm}, Act_{Result}, a \triangleright Act_{Remain})$
$)$;

If such a subset of action labels exists we replace all semantic actions with a label in the bag of actions by the communicating action. This is accomplished by first removing all semantic actions from the list with the help of $Actions^{-}$.

map $Actions^{-}$:$List(Act_{Lab}) \times List(Act_{\Sigma}) \times List(\Lambda) \to List(Act_{\Sigma})$;
var a_{lab}:Act_{Lab};
c_{dom}:$List(Act_{Lab})$;
as:$List(Act_{\Sigma})$;
$args$:$List(\Lambda)$;
eqn $Actions^{-}([], as, args) = as$;
$Actions^{-}(a_{lab} \triangleright c_{dom}, as, args) = Actions^{-}(c_{dom}, Action^{-}(ActSem(a_{lab}, args), as), args)$;

The actual removal of an action is performed by the function $Action^{-}$.

map $Action^{-}$:$Act_{\Sigma} \times List(Act_{\Sigma}) \to List(Act_{\Sigma})$;
var a, b:Act_{Σ};
as:$List(Act_{\Sigma})$;
eqn $Action^{-}(a, []) = []$;
$Action^{-}(a, b \triangleright as) = if(a \approx b, as, b \triangleright Action^{-}(a, as))$;

With the help of these auxiliary functions, we provide the data equations that correspond to the deduction rules $comm_1$ and $comm_2$:

eqn $R_{comm_1}(p, s) = if(is_{comm}(p), \{a{:}\mathcal{R}_{at} \mid is_{\checkmark}(\pi_t(a))$
$\wedge\ \exists_{ac':List(Act_{\Sigma})} at(ac', \pi_t(a), s) \in R(\pi_1(p), s) \wedge\ \sigma'(a) \approx s$
$\wedge\ ac(a) \approx Act_{\Sigma<}(Act_{Comm}(CL(p), ac'))$
$\wedge\ Act_{\Sigma<}^{?}(ac')\}, \emptyset)$;
$R_{comm_2}(p, s) = if(is_{comm}(p), \{a{:}\mathcal{R}_{at} \mid CL(\pi_t(a)) \approx CL(p)$
$\wedge\ is_{comm}(\pi_t(a)) \wedge\ \neg is_{\checkmark}(\pi_1(\pi_t(a))) \wedge\ \exists_{ac':List(Act_{\Sigma})}$
$ac(a) \approx Act_{\Sigma<}(Act_{Comm}(CL(p), ac'))$
$\wedge\ at(ac', \pi_1(\pi_t(a)), \sigma'(a)) \in R(\pi_1(p), s) \wedge\ Act_{\Sigma<}^{?}(ac')\}, \emptyset)$;

5.16 Process definition

A process definition in the mCRL2 language is described as a system of process equations that consists of a process label, a list of process parameters and a process expression. Within mCRL2 the system of process equations is defined by PE, a process definition is specified as $X(\overrightarrow{v}) = p$ where $X \in PE$, and a process reference is written as $X(\overrightarrow{v{=}e})$.

To model the process equations we introduce a mapping PES that specifies PE. The function maps process labels to process terms. To model process labels, we introduce sort $\mathcal{X}$ that defines a separate label for each of the equations in PE. Let the process definitions $\{X_1(\overrightarrow{v_1}), \ldots, X_n(\overrightarrow{v_n})\}$ be PE, where $X_1, \ldots, X_n$ are the process labels, $\overrightarrow{v_1}, \ldots, \overrightarrow{v_n}$ its corresponding process parameters, and $p_1, \ldots, p_n$ the associated process terms. Then we define the sort $\mathcal{X}$ as

sort $\mathcal{X} = \textbf{struct}\ X_1 \mid \ldots \mid X_n$;

We omit the list of process parameters since we write all process references as $X(v_1 = e_1, \ldots, v_n = e_n)$.

Example 5.16.1. Let the PE of an mCRL2 specification be defined by "$X_1 = a$" and "$X_2(v{:}\mathbb{B}) = b(v)$". Then we model PE in the meta notation as

sort $\mathcal{X} = \textbf{struct}\ p_1 \mid p_2$;
map PES:$\mathcal{X} \to \mathcal{P}$
eqn $\text{PES}(X_1) = alpha([Act(a, [])])$;
$\text{PES}(X_2) = alpha([Act(a, [\mathcal{E}_\mathcal{V}(\mathbb{B}_\mathcal{V}(v))])])$;

A single process reference $X(v_1 = e_1, \ldots, v_n = e_n)$ is written in the meta notation as:

$$Def(X, [(v_1, e_1), \ldots, (v_n, e_n)])$$

where $X \in \mathcal{X}$, v_i is a variable label and e_i is a data expression, for $i \in [1, n]$.

By modeling a process equation in this way, we preserve the option to provide the process parameter updates in random order. Since the direct use of tuples (nameless constructors) is prohibited in the mCRL2 language, we represent process parameter updates (v, e) through sort $\mathcal{Q}$

struct $\mathcal{Q} = \textbf{struct}\ pp(variable{:}\mathcal{V}, dataexpression{:}\mathcal{E})$;

By using *Def* in conjunction with PES we specify a mechanism that is used to assign data expressions to local variables and the substitution of variables in process terms and valuations.

To evaluate a process parameter we require the following two functions:

- *ComputePPunder*:$List(\mathcal{Q}) \times \mathcal{S} \to \mathcal{S}$ This function takes a list of process parameters and, with the help of a variable-to-value function, return the corresponding values for the variables of the process parameter. That is, the evaluation of $\{\!|\overrightarrow{e}|\!\}^\sigma$ in deduction rule def_1 and def_2 in Table 2.6.
- *ToInternalValuation*:$\mathcal{S} \to \mathcal{S}_f$ This function takes a mutable data valuation and returns the corresponding functional data valuation. All variables that are not defined within the functional data valuation are mapped to the $\perp$ value.

To replace variables in a process term ($[\overrightarrow{d}/\overrightarrow{d'}]$ in def_2 of Table 2.6) we use the function *VariableSubstitutionInProcessTerm*. This is the same function that has been discussed for the parallel operator in Chapter 5.7. It requires a variable substitution function and a process term to replace all variables in the process term.

As we fist compute the corresponding valuations for the process parameter updates they are still of the form $[\overrightarrow{d} \mapsto \{\!|\overrightarrow{e}|\!\}^\sigma]$. Since we require $[\overrightarrow{d}' \mapsto \{\!|\overrightarrow{e}|\!\}^\sigma]$, we need to rename the variables in the valuations as well. For that we will use the function *VariableSubstitutionInValuationList* that performs the same task as *VariableSubstitutionInProcessTerm* but for a mutable data valuations instead.

For the first deduction rule def_1 in Table 2.6, we compute the process parameter updates. This is accomplished by applying the function *ComputePPunder* to the process parameters defined by p, i.e., $ppl(p)$ and the data valuation s. While this computation requires a functional data validation function, we convert s with help of *ToInternalValuation*. To update the variables in the mutable data valuation we subsequently remove duplicate variables with the function *RemoveDuplicateVariablesFromValuations*:$List(\mathcal{Q}) \times \mathcal{S} \to \mathcal{S}$, concatenate the lists and order the valuations. As such we define the first deduction rule:

eqn $R_{Def_1}(p, s) = if(is_{def}(p), \{a{:}\mathcal{R}_{at} \mid at(ac(a), \pi_t(a), \text{S}) \in R(\text{PES}(P(p)), \text{S})$
$\wedge\ \sigma'(a) \approx s \wedge\ Act^?_{\Sigma_<}(ac(a)) \wedge\ is_\checkmark(\pi_t(a))\}, \emptyset)$
whr $\text{S} = \mathcal{S}_<(ComputePPunder(ppl(p), ToInternalValuation(s)) +\!\!+$
$RemoveDuplicateVariablesFromValuations(ppl(p), s))$
end ;

For the second deduction rule def_1 in Table 2.6, we first compute the values assigned to the variables by the process parameters under the current data valuation. This is performed by the function *ComputePPunder*. Then we substitute the defined variables by fresh variables, such that we can specify a variable substitution that is used in the mutable data valuation. The application of these functions is described as REN. To substitute old variables by freshly generated variables in a process term the we define SUBST. This function performs the substitution for the defined variables by freshly generate variables. The function is applied to the input process term. With these two functions we obtain the following data equation:

$$
\begin{array}{lll}
\textbf{eqn} & R_{Def_2}(p,s) = \mathit{if}(\mathit{is}_{def}(p), \{a{:}\mathcal{R}_{at} \mid a \in R(\textsc{Subst}, \textsc{Ren}{+\!\!+}s) & \\
 & \quad \wedge\ \mathit{Act}^{?}_{\Sigma<}(\mathit{ac}(a)) \wedge\ \neg \mathit{is}_{\checkmark}(\pi_t(a))\}, \emptyset) & \\
 & \textbf{whr}\ \ \textsc{Ren} = & \mathit{VariableSubstitutionInValuationList}(\\
 & & \quad \mathit{CreateVariableSubstitution}(\\
 & & \quad\quad \mathit{GetVarLabelsFromPP}(\mathit{ppl}(p)), \\
 & & \quad\quad \mathit{GenFreshVars}(\mathit{GetHighestId}(s)+1, \\
 & & \quad\quad\quad \mathit{GetVarLabelsFromPP}(\mathit{ppl}(p)))), \\
 & & \quad\quad \mathit{ComputePPunder}(\mathit{ppl}(p), \mathit{ToInternalValuation}(s))) \\
 & \quad , \textsc{Subst} = & \mathit{VariableSubstitutionInProcessTerm}(\\
 & & \quad \mathit{CreateVariableSubstitution}(\\
 & & \quad\quad \mathit{GetVarLabelsFromPP}(\mathit{ppl}(p)), \\
 & & \quad\quad \mathit{GenFreshVars}(\mathit{GetHighestId}(s)+1, \\
 & & \quad\quad\quad \mathit{GetVarLabelsFromPP}(\mathit{ppl}(p)))), \\
 & & \quad \mathrm{PES}(P(p))) \\
 & \textbf{end}\ ; &
\end{array}
$$

Chapter 6

Examples

In this section we illustrate some of models that have been used as test cases to validate the modeled semantics. The model of the semantics, which applies to the entire language is given in Appendix A.1. Semantics that are specific for the models are given in Appendix A.2. All the models that served as test cases are given in Appendix A.3, where each initialization denotes a separate test case.

Figure 6.1 illustrates these examples through six graphs that were generated using the mCRL2 toolset (development-svn revision:9701+). Each illustration shows the LTS for a model in the meta notation. In an LTS, an arrow depicts a transition and a node depicts a state. The initial state is a double lined node. A white colored node with outgoing transitions marks a non-terminating state whereas a white colored state without outgoing transitions marks a terminating state. A gray colored state without outgoing transitions marks a deadlock state. The characterization of the state space is given in each caption. The tools that have been used to generate the pictures are subsequently txt2lps, lps2lts and ltsgraph. The first tool reads a textual LPS and stores it in the binary LPS format. The second tool unfolds an LPS into a labeled transition system. The third tool has been used to position the states and export the figures.

Figure 6.1a Figure 6.1a shows the state space for the meta notation of the mCRL2 term "$\tau + a1 \cdot \delta$". The meta notation that corresponds to the mCRL2 term is:

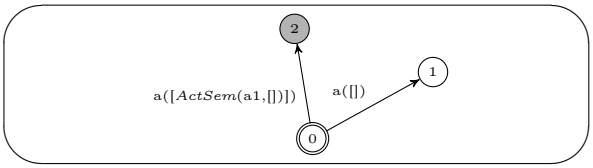
$$alt(alpha([tau]), seq(alpha([Act(a1, [])]), deadlock))$$

This meta notation is used at the initial term to generate the state space. Furthermore, we specify that the data valuation is empty, i.e., "$s = []$".

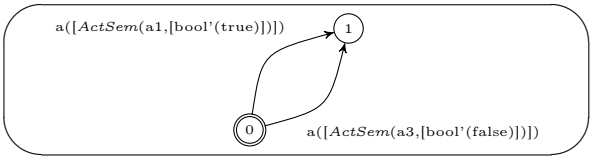
Figure 6.1b In Figure 6.1b we illustrate the state space for the mCRL2 process term "$\sum_{v1:\mathbb{B}} v1 \rightarrow a1(v1) \diamond (a3(v1)$". The meta notation that corresponds to the mCRL2 process term is given by:

$$Sum([\mathbb{B}_{\mathcal{V}}(v1)], cond2(\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v1)),\\ alpha([Act(a1, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v1))])]), alpha([Act(a3, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v1))])])))$$

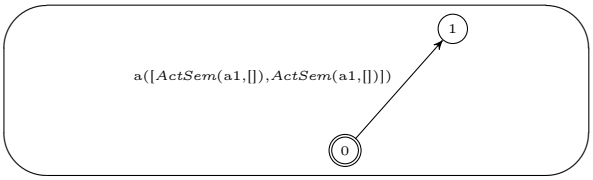
where the data valuation is initially empty.



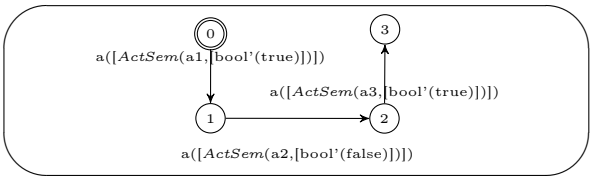
(a) $+$, $\cdot$, α, τ, δ



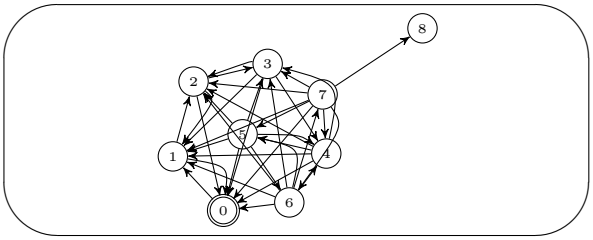
(b) $\sum$, $\rightarrow$, $\diamond$, α



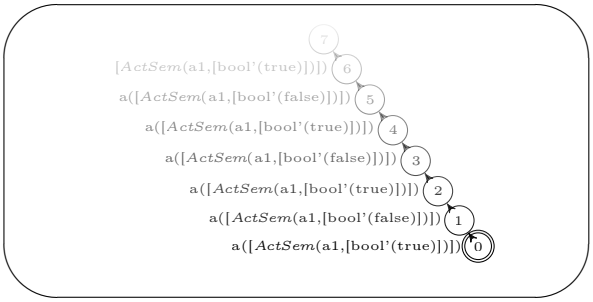
(c) Γ, σ, α



(d) X, σ



(e) X, $\|$, α



(f) X,σ,$\not\cong$

Figure 6.1: Six different mCRL2 specifications, generated with the implemented semantics of the mCRL2 language in the mCRL2 toolset

Figure 6.1c Figure 6.1c shows the LTS that results from the effect of a communication. The communication that we define is that two $a2|a2$ actions synchronize to an $a1$ action. To demonstrate, we consider the mCRL2 process term "$\Gamma_{a2|a2\to a1}(a2|a2)$". In the meta notation we write the mCRL2 process term as:

$$Comm(communication([a2, a2], a1), alpha([Act(a2, []), Act(a1, []), Act(a2, [])]))$$

For generating the state space, we assume the initial data valuation to be empty.

Figure 6.1d The effect of local variables (i.e., the assignment of values to process parameters) is illustrated in Figure 6.1d. Here, we model the following mCRL2 process term

proc $P2(v1{:}\mathbb{B}) = a1(v1) \cdot (P3(v1 = false) \cdot a3(v1));$
$P3(v1{:}\mathbb{B}) = a2(v1);$

The associated meta notation that goes with this term is defined as

eqn $\mathrm{PES}(p2) = seq(alpha([Act(a1, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v1))])]),$
$seq(Def(p3, [pp(\mathbb{B}_{\mathcal{V}}(v1), \mathcal{E}_{\Lambda}(\mathbb{B}_{\Lambda}(false)))]),$
$alpha([Act(a3, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v1))])])));$
$\mathrm{PES}(p3) = alpha([Act(a2, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v1))])]);$

The model is initialized by "$P2(v1 = true)$". To reflect the initialization in the meta notation, we specify "$p2([field(\mathbb{B}_{\mathcal{V}}(v1), \mathbb{B}_{\Lambda}(true))])$" as our input model. The data valuation will be empty. Observe in the LTS the value changes of the Boolean variable $v1$ in the data parameters of actions $a1$, $a2$, and $a3$.

Figure 6.1e Figure 6.1e shows the result of the semantics that for a recursive process definition that corresponds to the mCRL2 process term "$X = a1 \cdot (a1 \parallel X)$". Each time the recursion is unfolded, the process allows more concurrent behavior. The corresponding meta notation for this term is:

eqn $\mathrm{PES}(X) = seq(alpha([Act(a1, [])]), par(alpha([Act(a2, [])]), Def(X, [])));$

The initialization is provided through the mCRL2 process term "X". This term is specified in the meta notation as "$Def(X, [])$". The above specification is somewhat funny. To linearize an mCRL2 specification, a specification must be in the pCRL2 format [33]. Since the process introduces concurrency within a recursive process, the specification does not fit this format. Alternatively, the specification is not in a linear format so it cannot be parsed ans stored as an LPS. Since these are the only two mCRL2 input formats, we cannot generate the corresponding state space for this native mCRL2 process term. However, by first translating the process term into the meta notation, and use the semantic framework, we are able to generate a truncated state space. We show a truncated state space, since every unfolding of "X" introduces additional concurrency, which results in exponential growth of computation time (and memory) to calculate the transitions. Since the generated LTS is ever expanding, we only show the outgoing transitions system for the first eight states. The outgoing transitions from the ninth state and onwards are omitted.

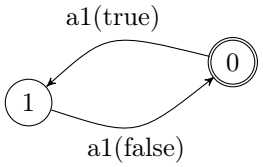


Figure 6.2: Recursion in mCRL2

Figure 6.1f Finally, in Figure 6.1f we show the behavior for another process recursion. The mCRL2 specification that we consider is:

$$P7(v1{:}\mathbb{B}) = a1(v1) \cdot X(\neg v1)$$

Here the recursion performs action $a1$ thereby showing the value of the Boolean data parameter $v1$. After performing the action, we negate the value of this variable and perform the recursion again. The state space that belongs to this mCRL2 specification is depicted in Figure 6.2 if we assume that $v1$ is initially *true*.

Now if we transform this process term into the meta notation we write:

$$\begin{aligned}\textbf{eqn}\quad & \mathrm{PES}(p7) = seq(alpha([Act(a1, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v1))])]),\\ & \quad Def(p7, [pp(\mathbb{B}_{\mathcal{V}}(v1), \mathcal{E}^1_{expr}(\mathbb{B}_{\mathcal{O}}(neg), \mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v1))))]));\end{aligned}$$

If we generate the state space for the meta notation model, we witness a non-terminating path in which the value of $v1$ alternates. To illustrate the non-terminating path, we add a fade to represent the everlasting continuation of the alternating pattern. If we compare the generated state spaces, we see that the semantics between the models deviates. Investigation shows that the difference is caused by the generation and subsequent rename of the fresh variables in a process definition. The mCRL2 semantics state that every time a process definition is expanded, it introduces new (fresh) variables. Since these variables are added to the data valuation, we never reach a previously visited state for which both the process term and data valuation have the same value. Note that this only holds for process definitions that have process parameters.

Chapter 7

Modeling compliance

To dogfood the Structural Operational Semantics of the mCRL2 language, we modeled the deduction rules and all related concepts as data equations and sorts. As we depend heavily on the underlying rewriting technology of the mCRL2 toolset, the modeling has to be done in a particular way to ensure computational feasibility. To ensure this we have to (i) create a specification that is a *mCRL2-restrictive* TSS [37], (ii) use deduction rules (along with auxiliary and supporting functions) that can be expressed in sorts and data equations, (iii) use data equations that are terminating, and (iv) avoid enumerations over dense domains (e.g. $\mathbb{R}$) and functions.

To meet (i), we observe that the used process term signature contains finitely many symbols and that the set of action labels $\mathcal{A}$ is finite, i.e., they are provided by the instantiated model. The TSS specifies a finite set of deduction rules and all deduction rules have a (strict) stratification. Therefore it is possible to compute all the deduction rules.

To fulfill (ii), we express all concepts of the mCRL2 language as sorts, data expressions and data equations that can be computed by the mCRL2 toolset. Therefore, some of the notations may deviate from typical mathematical notations. Examples of this can be found in the way set comprehension is denoted or the way in which a tuple is specified.

To comply to (iii), all evaluations of the data equations need to be finite. Especially when we want to compute a set comprehension that is recursively defined, we can define evaluations that will not terminate. Assuming that function g defines a recursive set comprehension, $f{:}A \to \mathbb{B}$ is a Boolean function on the input provided by $g{:}A \to 2^A$, and $h{:}A \to A$ denotes some function on the provided input. Now we define g as

$$g(p) = \{a{:}A \mid f(p) \wedge\ a \in g(h(p))\}$$

Since the current rewrite strategies in the mCRL2 toolset assume no order, it can be that $a \in g(h(p))$ is computed prior to $f(p)$. As g is defined recursively this results in an infinite recursion. To avoid this, we recommend to compute finite functions prior to (possible) infinite ones. As function f can be computed separately from the body, i.e., the computation is performed in a *if* construction, we alternatively write

$$g(p) = \mathit{if}(f(p), \{a{:}A \mid a \in g(h(p))\}, \emptyset)$$

The application of this technique can be found in the way in which the signature check is performed.

Restriction (iv) implies that we only analyze meta notated models for the untimed fragment of the mCRL2 language. Time has a dense domain (i.e., an uncountable number of solutions between any two different time values). This means that we would have introduced an uncountable number of solutions, which renders a meaningful analysis impossible. For that reason we have eliminated time from the semantics.

For the same reason, we also advise to avoid the use of sort $\mathbb{R}$ in the meta notation. Note that we do not state that we cannot transform these concepts. On the contrary, the transformation of the time concept, as well as other dense domains, poses no problem. To illustrate this, we sketch the way in which the deduction rules for the timed semantics of mCRL2 (e.g. [16]) can be transformed. First, observe that the semantics has a $\rightsquigarrow$ predicate. For that we introduce a new transition relation. To model $\xrightarrow{a}_t$ we extend the transition relation of $\xrightarrow{a}$: So we redefine $\mathcal{R}_{at}$ as

sort $\quad \mathcal{R}_{at} = \textbf{struct}\ at(ac{:}List(Act_\Sigma), \pi_{time}{:}\mathbb{R}^{>0}, \pi_t{:}\mathcal{P}, \sigma'{:}\mathcal{S});$

where $\pi_{time}{:}\mathbb{R}$ denotes the time-extension. Since time is reflected by the time 'at' operator ($@$), the *initialization* operator ($\gg$), and the before operator ($\ll$), we need to extend the signature of the process terms. Furthermore, we incorporate the data equations that describe the semantics of the operator, but these are almost straightforward. The crux of handling dense domains comes when we want to compute the possible outcome for these rules. In many cases they would simply provide an infinite number of solutions.

Chapter 8

Evaluation

With our approach we capture the untimed semantics of mCRL2 in (roughly) 1000 lines of mCRL2 code. To generate state spaces for native mCRL2 specifications we need to linearize the processes first. The resulting models preserve a bisimilation w.r.t. to the native specifications. So, with help of the tools, we can validate that the state spaces that are generated using the semantics are bisimilar (modulo the lifting of transitions and data) to the state spaces that are generated by the manual implementation of the language. However, if we would have had an (exhaustive) simulator that would act on the native specification prior to the linearization, we could validate that the relation between the state spaces describes an isomorphic relation. This provides confidence that the (intended) semantics is indeed implemented in the underlying source code.

Since our approach results in a large number of non-trivial data equations, the underlying rewriter has been tested extensively. Especially the use and solveability of quantifiers has been tested thoroughly. For this report we have validated the behavior of almost one hundred concepts (including small models for initialization). From this we can conclude that our method can be used to both prototype and evaluate the behavior of formal languages.

For larger models, such as the ABP, we were unable to compute the state space. This is caused by a number of issues. First, the models have not been optimized. That is, many of the computations for the deduction rules are performed several times (no caching). For example, the parallel operator defines eight deduction rules for which the premises share a number of computations that are individually (re)computed. By rearranging these computations we could easily increase the performance. However, this would compromise both the readability and traceability with respect to the original deduction rules. Second, the implementation of the mCRL2 semantics contains rather complex deduction rules. Resolving (and substituting) duplicate variables is extremely expensive. Removing, for example rule par_8 in Table 2.3, from the deduction rules, showed a difference of generating over 700 states par minute rather than just 50 states. To resolve the complex evaluation, we could model the fresh variable function differently, i.e., we could generated fresh variables based on the position (in terms of depth and 'left' or 'right' deduction rules) of the derivation tree and a number that counts the number of already freshly generated variables. Based on this information, we could generate unique fresh variables in a more efficient way. However, this requires that the deduction rules should incorporate the way in

which the variables are generated. Based on these observations, we see that the scalability stands with the complexity and number of deduction rules, the size of an instantiated model as well as the implementation of the underlying rewriter.

Dogfooding the formal language also forced us to reconsider the existing formal semantics. Although the language is formal, it still contained ambiguous behavior. An example can be found in e.g. the original specification for "let there be a fresh variable d' ". Does it mean that d' is a unique fresh variable or do infinitely many fresh variables correspond to d'? We assumed the first since it provides an analyzable model. If we would have modeled the second, it would create infinite branching behavior each time we require a fresh variable. Furthermore, we have also adapted the definition of a semantic multi-action. In the original work, the multi-action is a collection of semantic actions. It assumed that such a semantic multi-action is an equivalence class, however this is never explicitly stated. Since, we actually need to apply functions on these semantic multi-actions, we have explicitly stated that there indeed exists such an equivalence class and also included the representation of such a class.

As we expected, dogfooding of formal semantics reveals semantic issues and implementation bugs. Although the semantics has been considered finished since September 2009 [1], we were still able to uncover errata. These errata include simple oversights in the documentation such as duplicate deduction rules (e.g. for $\between$) and a missing deduction rule for the parallel operator. However, we also stumbled upon seven missing auxiliary operators that have been accidentally removed after performing a transition to a non-terminating term. To illustrate, deduction rule $allow_2$ was written as $\frac{(p,\sigma) \xrightarrow{m} (p',\sigma')}{(\nabla_V(p),\sigma) \xrightarrow{m} (p',\sigma')}$ while it should have been written as $\frac{(p,\sigma) \xrightarrow{m} (p',\sigma')}{(\nabla_V(p),\sigma) \xrightarrow{m} (\nabla_V(p'),\sigma')}$ instead.

Apart from this, we also uncovered two semantic errors. The first one we have already discussed in Chapter 6 where we show that a recursive process definition with process parameters corresponds to an infinite state space, since every iteration introduces fresh variables. The second semantic mismatch that we uncovered is illustrated by the following example. Consider the mCRL2 process

$$\textbf{proc} \quad P() = \textstyle\sum_{d:Bool} a(d) \cdot b(d);$$

The process picks a Boolean value and assigns it to the variable d, and performs action $a(d)$ followed by $b(d)$. Now, we also define a process $Q()$

$$\begin{aligned}\textbf{proc} \quad Q() &= a(\mathit{true}) \cdot b(\mathit{true}) \\ &\quad + a(\mathit{false}) \cdot b(\mathit{false});\end{aligned}$$

We see that $P()$ and $Q()$ are bisimilar. Now assume that $P() \parallel P() \cong Q() \parallel Q()$. We specify their specifications in the meta notation and generate their state spaces [2]. Since we assume that the process $P() \parallel P()$ and the process $Q() \parallel Q()$ are congruent, the state spaces must be (strongly) bisimilar. However, when we perform the bisimilation check[3] we observe that the state spaces are *not* strongly bisimilar. Obviously, something is wrong. When we observe the counter example traces, we see that $P() \parallel P()$ can perform the following actions

[1] Wed Sep 9 13:27:26 2009 UTC, rev 1575, svn repository: oas_repos

[2] tool: *lps2lts*

[3] tool: *ltsconvert -ebsim*

$a(false) \cdot a(true) \cdot b(true) \cdot b(true)$, but $Q() \parallel Q()$ cannot. The root cause for this problem lies in deduction rule sum_2 in Table 2.2 that assigns a value to the binder variable. If a valuation for the binder variable already exists, the selected value overwrites the value in the valuation. Due to the interleaving behavior of $P()$ and the sum operator, local variables are overwritten resulting in the observed undesirable behavior. Hence, it means that for the current set of deduction rules $P() \parallel P() \not\cong Q() \parallel Q()$ holds.

On the implementation side we also uncovered bugs and ill performing code. Many of the ill performing code related to data expressions that could not be resolved. As a result, parts of the underlying code have been altered and rewrite rules have been added to the data specification to evaluate the data expressions. In turn, this has led to the refactoring and unification of code for the underlying rewriters. This included the removal of obsolete rewrite strategies and simplification of the code. The refactoring has led to a significant improvement of the performance when evaluating quantifiers (a factor of 5), as shown in Figure 8.1. Here, the horizontal axis depicts the revision of the toolset whereas the logarithmic vertical axis depicts the time needed to generate the state space. Since these experiments have been performed in the development branch, some revisions could not generate a state space and are omitted from the plot (which results in apparent gaps).

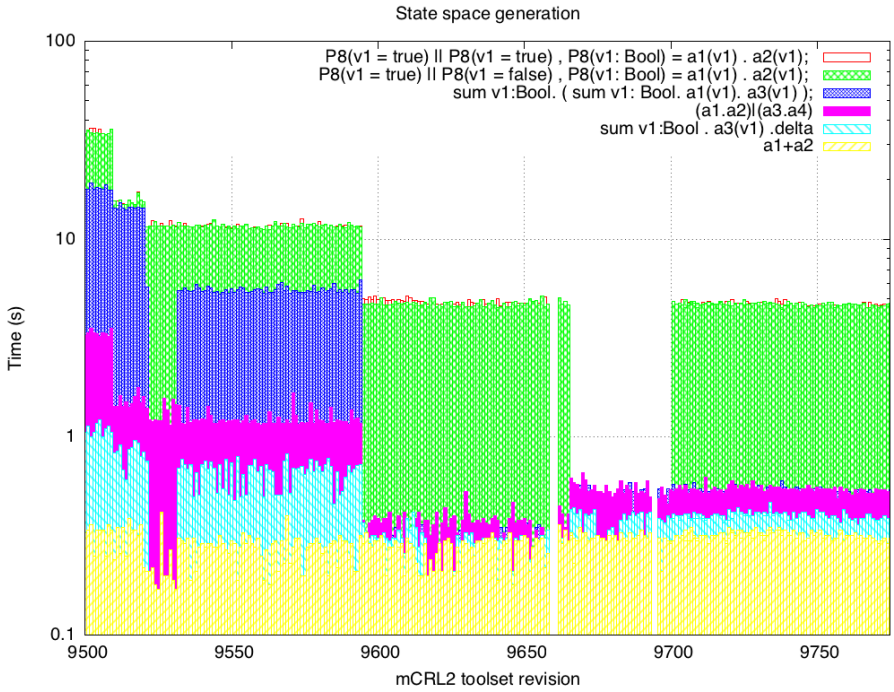


Figure 8.1: Performance measurements for state space generation

Furthermore, our approach sparked a discussion on the external use of finite sets and bags by users. Normal sets and bags cannot be enumerated since they are characterized by functions and exception lists. As they have no constructor

functions, they have no smallest nor largest element. Hence we can potentially have infinitely many solutions. Since our approach, at some points, requires storage for a finite number of elements (e.g. to represent a data valuation or a semantic multi-action equivalence class) the use of finite sorts and bags would be rather convenient and helpful. Finite sets are currently already available in the underlying toolset but cannot be accessed by users. Finally, the internal representation of finite sets and finite bags could be reconsidered. These container sorts use an ordered-list to represent the containing sort. Consequently, lists are traversed in linear time to e.g. find or store an element in the ordered lists. As tree-like data structures are known to perform better, we should consider them here as well.

Chapter 9

Related work

Dogfooding is applied in (ordinary) software development for the development of new, or to extend existing, products. Examples can be found in the field of compilers [39], where dogfooding (or in this case bootstrapping) is applied to construct compilers. Other examples include the Eclipse framework [19] that is used to developed plug-ins for, and extension of, the Eclipse framework. Another example that could be considered is the use of Emacs/Vi. Here, these editors are used to write customizations for the editors themselves. Wolfram Research states [26] that (parts) of their web sites, applications, documentations, and test and build processes are all driven by the *Mathematica* Language.

Practicing these kind of techniques in formal software engineering, especially in the area of verifying formal languages and model checkers is not so common. In our case, we have a model checker eat and interpret the formal semantics of its own language which is rather rare. In fact, we believe that our dogfooding approach is unique and the first of its kind. However, it still leaves the question, could we have used other approaches? To the best of our knowledge, Maude would be the only candidate that could directly express the Structural Operational Semantics of mCRL2. Maude is a high-level language and high-performance system supporting both equational and rewriting logic and is used for, and applied to, a wide range of applications. It's simple and expressive logic allows the representation of many models of concurrent and distributed systems, including forms of SOS.

Among many work, in [9] the authors outline a translation from Modular SOS (MSOS) [27, 28] to the Maude rewriting logic and prove the transformation correct. In [8] the authors show that they are able to model GSOS/OSOS rules in the Maude system, which allows them to execute Ordered SOS specifications. The work of [21] shows the implementation of Eden (the parallel extension of the functional language Haskell) in Maude. More recent work [34] presents the implementation of the semantics for the πCRWL calculus and the formalization of AADL in [30]. Based on these, and many other successful experiments, we believe that it is possible to implement the semantics of the mCRL2 language with Maude. As this route is still open, the implementation of the mCRL2 semantics in Maude could be considered as future work. Note that with this approach we would have to use *separate* toolsets. As a result, a successful implementation in Maude would 'only' be able to uncover the identified semantic issues but *not* the uncovered implementation issues.

Chapter 10

Conclusion and future work

In [20] it is stated that "Engineers who use their own company's tools exclusively, tend to propagate the bad aspects of their tools because they might not even realize an alternative approach exists. They often fail to either understand or appreciate the good points of other companies' tools. Furthermore, it also encourages the Not Invented Here syndrome." In this report we show how dogfooding can be used as a valuable technique for the identification of 'bad aspects' not only in the tools but also in their governing formal semantics.

Using our semantic dogfooding approach, we show that the mCRL2 toolset can actually eat *and* verify the Structural Operational Semantics of its own language. To achieve this, the SOS has been captured using an mCRL2 specification that is an *mCRL2-restrictive* TSS and contains deduction rules (along with auxiliary and supporting functions) that can all be expressed as sorts and data equations. To ensure computational feasibility, these data equations have to be are terminating and enumerations across dense domains and functions should be avoided. The transformation of all language concepts and their formal semantics is a non-trivial task for which we outline and motivate all underlying design and implementation decisions. To illustrate the feasibility and application of our approach, we included some examples and their generated state spaces.

This report shows that the mCRL2 toolset is capable to formally analyze the operational semantics associated with a language like mCRL2 itself and (exhaustively) simulate the behavior using the underlying toolset. Our experiments and results show that, even though the mCRL2 semantics have been defined formally and are considered stable for over two years, we were still able to detect and pinpoint mismatches between the formally defined intended semantics and its manual implementation in the toolset. Next to the expected small errata in the accompanying documentation, we also uncovered more severe issues like violating congruence and an infinite state space caused by non terminating recursion. These results not only provide clear merits for the automated analysis of formal semantics but also show that the mCRL2 toolset can potentially be used for prototyping and exploring the formal semantics of a language. To actually use the formal semantics, in conjunction with large models is currently too ambitious. For that we need to improve the performance of the rewriters (and its underlying data structures), and optimize the deduction rules such that e.g. performing the same computations is kept at a minimum. The use of axioms can be embedded to define equivalence classes on process terms. By addressing

these in the reduction of process terms to a normal form we could also potentially improve performance.

Since the mCRL2 language is used as our target specification language, all computations are performed in the mCRL2 toolset. Therefore it could be that an implementation given directly in a native programming language (i.e., the manual implementation) has better performance as no additional overhead is introduced. That is, the underlying toolset was not implemented with our framework in mind. Based on our work we sparked several ideas to further improve and unify the underlying rewrite strategies. The work in this report can be seen as a first step in a line of research towards a generalized model checker that can accept a wide variety of SOS-based (domain specific) languages. The semantic validation of other formal languages such as CSP, CIF, ASML and POOSL can be considered as future work. Also, the framework can be used for the formalization and validation process of informal languages like SLCO and various domain specific languages is considered future work. To illustrate the latter we have formalized an informal domain specific language in [38] and used an early version of our framework to generate state spaces and analyze behavior of domain models. In such a case, the formalization process can not only be used to detect ambiguities in the language semantics but also help to 'tune' deduction rules for computational performance.

The integration into language workbenches is considered future work as well. That is, a language workbench could support the definition of signatures and associated Structural Operational Semantics including an automated transformation from these deduction rules to data equations. The latter nearly constitutes a one to one mapping when considering the language specific part of our mCRL2 implementation. Such a tool could be useful, since the manual implementation of the rules is tedious, time consuming and error prone. Also the conversion from the syntactic instance of a model to its syntactic meta notation could easily be automated. Since the semantic interpretation is not bound to a single language, it theoretically allows the study of compound concepts in different formal languages within a single mCRL2 specification. Another direction for future work would be to use a heterogeneous composition of native mCRL2 models and semantic models defined in other languages.

Apart from this, we would like to consider the timed fragment of the mCRL2 language. As a direct interpretation of the dense time domain would pose all kind of problems, it might be worthwhile to consider an approach that partitions the dense domain into a discrete/non-dense domain. We would start by considering such partitioning rules as part of the formal semantic definition. Finally, we want consider a method for the validation of congruence relations as future work. That is, with the help of axioms and a generated set of models that together cover all deduction rules, one could verify that an axiom holds and whether the behavior related by an axiom is strongly bisimilar.

In conclusion, we would like to emphasize that our approach can be applied to, and implemented in, other languages and toolsets. That is, if a language and toolset support the definition and computation of set comprehensions, can deal with quantifiers and support a mechanism to systematically perform transitions (i.e. can implement an LPS), it could be used to implement and most likely benefit from our approach. As such we encourage others to conduct experiments similar to ours.

Appendix A

Models

A.1 Language semantics

The language semantics describes the implementation of the semantics that hold for all untimed mCRL2 models. To create a valid LPS, this semantics is extended with the model specific (static) semantics (Appendix A.2) and a specific model (Appendix A.3).

```
% Internal Valuation
sort InternalValuation = Variable -> Value;

% Mutable Valuation
sort Valuation = List(Field);

% A field for in a mutable valuation
sort Field = struct field( variable: Variable , valvalue: Value );

% Data expression
sort DataExpression = struct
                        de_var(dvr:Variable)?is_de_var
                    | de_val(dvl:Value)?is_de_val
                    | de_expr_1(f: Func, expr1: DataExpression )?is_de_expr_1
                    | de_expr_2(f: Func, expr1: DataExpression,
                                         expr2: DataExpression )?is_de_expr_2;
% Syntactic action
sort ActionSyntax  =
       struct Act(  actionlabel: ActionLabel, args: List(DataExpression))
            | ActionTau;

% Semantic action
sort ActionSemantic =
       struct ActSem( actionlabel: ActionLabel, args: List(Value));

% The signature of an mCRL2 process terms
sort ProcessTerm = struct
                   checkmark?is_checkmark
                 | deadlock?is_deadlock
                 | alpha( multiaction: List(ActionSyntax) )?is_alpha
                 | alt( pi_1: ProcessTerm, pi_2: ProcessTerm)?is_alt
                 | seq( pi_1: ProcessTerm, pi_2: ProcessTerm)?is_seq
                 | cond1( C: DataExpression, pi_1: ProcessTerm)?is_cond1
                 | cond2( C: DataExpression,
                     pi_1: ProcessTerm, pi_2: ProcessTerm)?is_cond2
                 | Sum( d: Variable, pi_1: ProcessTerm)?is_sum
                 | par( pi_1: ProcessTerm, pi_2: ProcessTerm)?is_par
                 | lmerge( pi_1: ProcessTerm, pi_2: ProcessTerm)?is_lmerge
                 | sync( pi_1: ProcessTerm, pi_2: ProcessTerm)?is_sync
                 | Allow( V:Set(Bag(ActionLabel)), pi_1: ProcessTerm )?is_allow
                 | Block( B:Set(ActionLabel), pi_1: ProcessTerm )?is_block
                 | Rename( Ren:ActionLabel->ActionLabel,
```

```
                    pi_1: ProcessTerm )?is_rename
                  | Hide( I:Set(ActionLabel), pi_1: ProcessTerm )?is_hide
                  | Prehide( U:Set(ActionLabel), pi_1: ProcessTerm )?is_prehide
                  | Comm( CL:List(Communication), pi_1: ProcessTerm )?is_comm
                  | Def( P:ProcessLabel, ppl: List(PP) )?is_def
                  ;

% The process parameter sort
sort PP  = struct pp ( variable: Variable , dataexpression: DataExpression);

% The communication sort
sort Communication =
       struct communication( CmI: List(ActionLabel), CmR: ActionLabel );


sort ActionTransition =
       struct at( ac:   List(ActionSemantic),
                  pi_t: ProcessTerm,
                  sigma': Valuation );

% Semantic interpretation
map sem_ActList: List(ActionSyntax)#InternalValuation   -> List(ActionSemantic);
    sem_Act:     ActionSyntax# InternalValuation        -> ActionSemantic ;
    sem_Var:     Variable#InternalValuation             -> Value;
    sem_DexList: List(DataExpression)#InternalValuation -> List(Value);
var a:  ActionSyntax;
    as: List(ActionSyntax);
    sigma: InternalValuation;
    d:  Variable;
    des: List(DataExpression);
    de: DataExpression;
    expr1: DataExpression;
    expr2: DataExpression;
    l: ActionLabel;
eqn sem_ActList( [] , sigma )       = [] ;
    sem_ActList( a |> as , sigma )  =
      if( a == ActionTau,
        sem_ActList( as, sigma),
        sem_Act(a , sigma ) |>  sem_ActList( as, sigma));
    sem_Act( Act(l,des) , sigma)    = ActSem( l, sem_DexList( des, sigma) );
    sem_Var( d, sigma)              = sigma(d);
    sem_DexList( [] , sigma )       = [];
    sem_DexList( de |> des, sigma ) =
      sem_Dex(de, sigma) |> sem_DexList( des, sigma);

% Label identity function
map ID: ActionLabel -> ActionLabel;
var x: ActionLabel;
eqn ID(x)=x;

% Action rename function
map ActRename: (ActionLabel -> ActionLabel)#List(ActionSemantic)
                  -> List(ActionSemantic);
var f: ActionLabel->ActionLabel;
    a: ActionSemantic;
    as: List(ActionSemantic);
eqn ActRename(f, [] ) = [];
    ActRename(f, a |> as ) =
      ActSem(f(actionlabel(a)), args(a)) |> ActRename(f, as);

% Action hide function
map ActHide: Set(ActionLabel)#List(ActionSemantic) -> List(ActionSemantic);
var I: Set(ActionLabel);
    as: List(ActionSemantic);
    a: ActionSemantic;
eqn ActHide(I, [])       = [];
     (actionlabel(a) in I) -> ActHide(I, a |> as ) =      ActHide(I, as);
    !(actionlabel(a) in I) -> ActHide(I, a |> as ) = a |> ActHide(I, as);

% Action prehide function
map ActPrehide: Set(ActionLabel)#List(ActionSemantic) -> List(ActionSemantic);
var U: Set(ActionLabel);
    as: List(ActionSemantic);
    a: ActionSemantic;
```

```
eqn ActPrehide(U, [])       = [];
    (actionlabel(a) in U) -> ActPrehide(U, a |> as ) =
                          ActSem( int, []) |>  ActPrehide(U, as);
    !(actionlabel(a) in U) -> ActPrehide(U, a |> as ) =
                      a |> ActPrehide(U, as);

% Action communication function
map ActComm: List(Communication)#List(ActionSemantic) -> List(ActionSemantic);
var as: List(ActionSemantic);
    C: Communication;
    CL: List(Communication);
eqn ActComm([]      , as) = as ;
    ActComm(C |> CL , as) =
      ActCommAux( C , CL, as,
       ArgumentsToActionLabelMap( as, lambda x: List(Value). {} ),
       ActionLabelsToBag( CmI(C)) , [] , [] );


% Auxiliary function required by the action communication function
% that maps action data parameters to a bag of action labels
map ArgumentsToActionLabelMap: List(ActionSemantic)#
      (List(Value) -> Bag(ActionLabel)) -> (List(Value) -> Bag(ActionLabel));
var ActionParametersToActionLabels: List(Value)->Bag(ActionLabel);
    as: List(ActionSemantic);
    a: ActionSemantic;
eqn ArgumentsToActionLabelMap( [] , ActionParametersToActionLabels ) =
      ActionParametersToActionLabels;
    ArgumentsToActionLabelMap( a |> as , ActionParametersToActionLabels ) =
      ArgumentsToActionLabelMap( as,
       ActionParametersToActionLabels[
        args(a) -> ActionParametersToActionLabels(args(a)) +
                   {actionlabel(a):1}] );

% Auxiliary function required by the action communication function
% that transforms a list of action labels to a bag of action labels.
map ActionLabelsToBag: List(ActionLabel) -> Bag(ActionLabel);
var ActLab: ActionLabel;
    CommActLabels: List(ActionLabel);
eqn ActionLabelsToBag( [] ) = {};
    ActionLabelsToBag( ActLab |> CommActLabels ) =
      {ActLab:1} + ActionLabelsToBag(CommActLabels);

% Auxiliary function required by the action communication function
% that computes the synchronizing actions in the remaining multi-action.
map ActCommAux: Communication#List(Communication)#List(ActionSemantic)#
      (List(Value)->Bag(ActionLabel))#Bag(ActionLabel)#List(ActionSemantic)#
      List(ActionSemantic) -> List(ActionSemantic);
var ActionParametersToActionLabels: List(Value)->Bag(ActionLabel);
    CommActBag: Bag(ActionLabel);
    ResultActions, RemainingActions: List(ActionSemantic);
    C: Communication;
    CL: List(Communication);
    as: List(ActionSemantic);
    a: ActionSemantic;
eqn ActCommAux( C, CL, [], ActionParametersToActionLabels,
      CommActBag, ResultActions, RemainingActions ) =
        ResultActions ++ ActComm( CL, RemainingActions);
    ActCommAux( C, CL, a |> as , ActionParametersToActionLabels,
      CommActBag, ResultActions, RemainingActions ) =
        if( CommActBag <= ActionParametersToActionLabels(args(a)),
          %Condition holds
          ActCommAux( C, CL,
            EliminateMatchingActions( CmI(C) , a |> as, args(a) ),
            ArgumentsToActionLabelMap(
              EliminateMatchingActions( CmI(C) , a |> as, args(a) ),
              lambda x: List(Value). {} ),
            CommActBag, ActSem(CmR(C), args(a)) |> ResultActions,
            RemainingActions),
         %Condition does not hold
          ActCommAux( C, CL, as, ActionParametersToActionLabels, CommActBag,
            ResultActions, a |> RemainingActions)
         );

% Auxiliary function required by the action communication function
```

```
% to remove an occurrence of a synchonizing action in the remaining multi-action
map EliminateMatchingActions: List(ActionLabel)#List(ActionSemantic)#List(Value)
-> List(ActionSemantic);
var ActLab: ActionLabel;
    CommActLabels: List(ActionLabel);
    as: List(ActionSemantic);
    args: List(Value);
eqn EliminateMatchingActions( [] , as, args ) = as ;
    EliminateMatchingActions( ActLab |> CommActLabels , as, args ) =
      EliminateMatchingActions( CommActLabels ,
        RemoveAction( ActSem( ActLab ,args), as), args );

% Auxiliary function required by the action communication function
% to remove an action in the remaining multi-action
map RemoveAction: ActionSemantic#List(ActionSemantic)->List(ActionSemantic);
var a, b: ActionSemantic;
    as: List(ActionSemantic);
eqn RemoveAction( a, [] ) = [];
    RemoveAction( a, b |> as ) = if(a == b , as,  b |> RemoveAction( a, as));

% Determine the last generated fresh variable in a data valuation
map GetHighestId: Valuation -> Nat;
    GetVarId : Field -> Nat;
var las: Valuation;
    a: Field;
eqn GetHighestId( [] ) = 0;
    GetHighestId( a |> las ) = max(GetVarId(a), GetHighestId(las));
    GetVarId( a ) =
      if( is_d'(variablelabel(variable(a))),
        id(variablelabel(variable(a))), 0) ;

% Conversion of mutable valuation into internal valuation
map ToInternalValuation: Valuation -> InternalValuation;
    ToInternalValuation: Valuation#InternalValuation -> InternalValuation;
var as: Field;
    lass: Valuation;
    sigma: InternalValuation;
eqn ToInternalValuation( lass ) =
      ToInternalValuation( lass, lambda v: Variable. bot );
    ToInternalValuation( [] , sigma ) = sigma;
    ToInternalValuation( as |> lass, sigma ) =
      ToInternalValuation( lass, sigma[ variable(as) -> valvalue( as ) ] );

% Conversion of a list of semantics action to a bag of action labels
map actionlabels: List(ActionSemantic) -> Bag(ActionLabel);
var as: List(ActionSemantic);
    a: ActionSemantic;
eqn actionlabels( [] ) = {};
    actionlabels( a |> as ) = {actionlabel(a):1} + actionlabels(as);

% Update a mutable valuation with a given field
map UpdateValuation: Field#Valuation -> Valuation;
var a,b: Field;
    bl: Valuation;
    v: Variable;
    w: Value;
    vs: List(Variable);
    ws: List(Value);
eqn UpdateValuation( a, [] ) = [a];
    UpdateValuation( a, b |> bl) =
      if(variable(b) == variable(a), a |> bl,  b |> UpdateValuation( a, bl)) ;

% The solutions functions that compute the transition relations
map R,
    R_alpha,
    R_alt_1, R_alt_2, R_alt_3, R_alt_4,
    R_seq_1, R_seq_2,
    R_cond1_1, R_cond1_2,
    R_cond2_1, R_cond2_2, R_cond2_3, R_cond2_4,
    R_sum_1, R_sum_2,
    R_par_1, R_par_2, R_par_3, R_par_4, R_par_5, R_par_6, R_par_7, R_par_8,
    R_sync_1, R_sync_2, R_sync_3, R_sync_4,
    R_lmerge_1, R_lmerge_2,
    R_allow_1, R_allow_2,
```

```
264       R_block_1, R_block_2,
265       R_rename_1, R_rename_2,
266       R_hide_1, R_hide_2,
267       R_prehide_1, R_prehide_2,
268       R_comm_1, R_comm_2,
269       R_Def_1, R_Def_2
270       : ProcessTerm#Valuation -> Set( ActionTransition );
271   var p: ProcessTerm;
272       s: Valuation;
273   eqn R(p,s)   = R_alpha(p,s)
274                + R_alt_1(p,s) + R_alt_2(p,s) + R_alt_3(p,s) + R_alt_4(p,s)
275                + R_seq_1(p,s) + R_seq_2(p,s)
276                + R_cond1_1(p,s) + R_cond1_2(p,s)
277                + R_cond2_1(p,s) + R_cond2_2(p,s) + R_cond2_3(p,s) + R_cond2_4(p,s)
278                + R_sum_1(p,s)  + R_sum_2(p,s)
279                + R_par_1(p,s) + R_par_2(p,s) + R_par_3(p,s) + R_par_4(p,s)
280                + R_par_5(p,s)
281                + R_par_6(p,s) + R_par_7(p,s) + R_par_8(p,s)
282                + R_sync_1(p,s) + R_sync_2(p,s) + R_sync_3(p,s) + R_sync_4(p,s)
283                + R_lmerge_1(p,s) + R_lmerge_2(p,s)
284                + R_allow_1(p,s) + R_allow_2(p,s)
285                + R_block_1(p,s) + R_block_2(p,s)
286                + R_rename_1(p,s) + R_rename_2(p,s)
287                + R_hide_1(p,s) + R_hide_2(p,s)
288                + R_prehide_1(p,s) + R_prehide_2(p,s)
289                + R_comm_1(p,s) + R_comm_2(p,s)
290                + R_Def_1(p,s)  + R_Def_2(p,s)
291                ;
292
293       %%
294       %% -------------------------------------
295       %% (alpha,s) --[[alpha]](s)--> checkmark
296       R_alpha(p,s) = if( is_alpha(p) ,
297         { at(OrderAction(sem_ActList(multiaction(p), ToInternalValuation( s))),
298              checkmark,  s)},
299         {} );
300
301       %   (p,s) --m--> checkmark
302       % --------------------------
303       % (p + q,s) --m--> checkmark
304       R_alt_1(p,s) = if( is_alt(p), { a: ActionTransition |
305                          a in R(pi_1(p),s)
306                       && checkmark == (pi_t(a))
307                       && sigma'(a) == s
308                       }, {} );
309
310       %   (p,s) --m--> (p',s')
311       % -------------------------
312       % (p + q, s) --m--> (p',s')
313       R_alt_2(p,s) = if( is_alt(p), { a: ActionTransition |
314                          a in R(pi_1(p),s)
315                       && !is_checkmark(pi_t(a))
316                         }, {} );
317
318       %   (q,s) --m--> checkmark
319       % --------------------------
320       % (p + q,s) --m--> checkmark
321       R_alt_3(p,s) = if( is_alt(p), { a: ActionTransition |
322                          a in R(pi_2(p),s)
323                       && is_checkmark(pi_t(a))
324                       && sigma'(a) == s
325                       }, {} );
326
327       %   (p,s) --m--> (q',s')
328       % -------------------------
329       % (p + q, s) --m--> (q',s')
330       R_alt_4(p,s) = if( is_alt(p), { a: ActionTransition |
331                          a in R(pi_2(p),s)
332                       && !is_checkmark(pi_t(a))
333                       }, {} );
334
335       % (p,s) --m--> checkmark
336       % ----------------------
337       % (p . q,s) --m--> (q,s)
```

```
338     R_seq_1(p,s) = if( is_seq(p), { a: ActionTransition |
339                     at(ac(a), checkmark, sigma'(a) ) in R(pi_1(p),s)
340                 && pi_t(a) == pi_2(p)
341                 && sigma'(a) == s
342                 }, {} );
343
344     %     (p,s) --m--> (p',s')
345     % ----------------------------
346     % (p . q,s) --m--> (p'. q, s')
347     R_seq_2(p,s) = if( is_seq(p),
348                  { a: ActionTransition |
349                    is_seq(pi_t(a))
350                 && at(ac(a), pi_1(pi_t(a)), sigma'(a)) in R(pi_1(p),s)
351                 && pi_2(pi_t(a)) == pi_2(p)
352                 && pi_1(pi_t(a)) != checkmark
353                 }, {} );
354
355     % (p,s) --m--> checkmark && [[b]](s)==true
356     % ----------------------------------------
357     % ( b -> p,s) --m--> checkmark
358     R_cond1_1(p,s)= if( is_cond1(p)
359        && Cast2InternalBool( sem_Dex(C(p), ToInternalValuation(s)) )
360                   , { a: ActionTransition |
361                         a in R( pi_1(p), s)
362                      && is_checkmark(pi_t(a))
363                      && sigma'(a) == s
364                    }, {});
365
366     % (p,s) --m--> (p',s') && [[b]](s)==true
367     % --------------------------------------
368     % ( b -> p,s) --m--> (p',s')
369     R_cond1_2(p,s)= if( is_cond1(p)
370         && Cast2InternalBool( sem_Dex(C(p), ToInternalValuation(s)) )
371                   , { a: ActionTransition |
372                         a in R( pi_1(p), s)
373                      && !is_checkmark(pi_t(a))
374                    }, {});
375
376     % (p,s) --m--> checkmark && [[b]](s)==true
377     % ----------------------------------------
378     % ( b -> p <> q,s) --m--> checkmark
379     R_cond2_1(p,s)= if( is_cond2(p)
380         && Cast2InternalBool( sem_Dex(C(p), ToInternalValuation(s)) )
381                    , { a: ActionTransition |
382                         a in R( pi_1(p), s)
383                      && is_checkmark(pi_t(a))
384                      && sigma'(a) == s
385                    }, {});
386
387     % (p,s) --m--> (p',s') && [[b]](s)==true
388     % --------------------------------------
389     % ( b -> p <> q ,s) --m--> (p',s')
390     R_cond2_2(p,s)= if( is_cond2(p)
391         && Cast2InternalBool( sem_Dex(C(p), ToInternalValuation(s)) )
392                   , { a: ActionTransition |
393                         a in R( pi_1(p), s)
394                      && !is_checkmark(pi_t(a))
395                    }, {});
396
397     % (q,s) --m--> checkmark && [[b]](s)==false
398     % -----------------------------------------
399     % ( b -> p <> q,s) --m--> checkmark
400     R_cond2_3(p,s)= if( is_cond2(p)
401         && !Cast2InternalBool( sem_Dex(C(p), ToInternalValuation(s)) )
402                   ,{ a: ActionTransition |
403                         a in R( pi_2(p), s)
404                      && is_checkmark(pi_t(a))
405                      && sigma'(a) == s
406                    }, {});
407
408     % (q,s) --m--> (q',s') && [[b]](s)==false
409     % --------------------------------------
410     % ( b -> p <> q ,s) --m--> (q',s')
411     R_cond2_4(p,s)= if( is_cond2(p)
```

```
412           && !Cast2InternalBool( sem_Dex(C(p), ToInternalValuation(s)) )
413                       , { a: ActionTransition |
414                              a in R( pi_2(p), s)
415                           && !is_checkmark(pi_t(a))
416                        }, {});
417
418       % (p,s[d:=e]) --m--> checkmark
419       % ------------------------------- e in M_D
420       % (sum d:D . p, s) --m--> checkmark
421       R_sum_1(p,s) = if( is_sum(p)   , { a: ActionTransition |
422                              sigma'(a) == s
423                           && is_checkmark(pi_t(a))
424                           && (exists v: Value .
425                                RestrictDomain( d(p), v )
426                            && (at(ac(a), pi_t(a), Z) in R( pi_1(p), Z)
427                                whr Z = OrderValuation(
428                                  UpdateValuation( field(d(p), v), s)) end)
429                              )}, {});
430
431       % (p,s[d:=e]) --m--> (p',s')
432       % ------------------------------- e in M_D
433       % (sum d:D . p, s) --m--> (p',s')
434       R_sum_2(p,s) = if( is_sum(p)   , { a: ActionTransition |
435                              !is_checkmark(pi_t(a))
436                           && (exists v: Value .
437                              RestrictDomain( d(p), v )
438                           && a in R( pi_1(p),
439                              OrderValuation(UpdateValuation(field(d(p) ,v), s)))
440                           ) }, {});
441
442       %  (p,s) --m--> checkmark
443       % ------------------------
444       % (p || q, s) --m--> (q, s)
445       R_par_1(p,s) = if( is_par(p)   , { a: ActionTransition |
446                              at(ac(a), checkmark, s) in R( pi_1(p), s)
447                           && pi_t(a) ==pi_2(p)
448                           && sigma'(a) == s
449                           }, {});
450
451       %  (p,s) --m--> (p',s')
452       % ------------------------------
453       % (p || q, s) --m--> (p'|| q, s')
454       R_par_2(p,s) = if( is_par(p)   , { a: ActionTransition |
455                              is_par(pi_t(a))
456                           && at(ac(a), pi_1(pi_t(a)), sigma'(a)) in R( pi_1(p), s)
457                           && !is_checkmark(pi_1(pi_t(a)))
458                           && pi_2(pi_t(a)) ==pi_2(p)  }, {});
459
460       %  (q,s) --m--> checkmark
461       % ------------------------
462       % (p || q, s) --m--> (p, s)
463       R_par_3(p,s) = if( is_par(p)   , { a: ActionTransition |
464                              at(ac(a), checkmark, s) in R( pi_2(p), s)
465                           && pi_t(a) ==pi_1(p)
466                           && sigma'(a) == s
467                           }, {});
468
469       %  (p,s) --m--> (q',s')
470       % -------------------------------
471       % (p || q, s) --m--> (p || q', s')
472       R_par_4(p,s) = if( is_par(p)   , { a: ActionTransition |
473                              is_par(pi_t(a))
474                           && at(ac(a), pi_2(pi_t(a)), sigma'(a)) in R( pi_2(p), s)
475                           && !is_checkmark(pi_2(pi_t(a)))
476                           && pi_1(pi_t(a)) ==pi_1(p)  }, {});
477
478       % (q,s) --m--> checkmark,  (q,s) --n--> checkmark
479       % ------------------------------------------------
480       % (p || q, s) --m|n--> checkmark
481       R_par_5(p,s) = if( is_par(p)   , { a: ActionTransition |
482                              IsOrderedActionList(ac(a))
483                           && is_checkmark(pi_t(a))
484                           && sigma'(a) == s
485                           && exists t1, t2: List(ActionSemantic).
```

```
                        at(t1, checkmark, s ) in R( pi_1(p), s)
                     && at(t2, checkmark, s ) in R( pi_2(p), s)
                     && OrderAction(t1 ++ t2) == ac(a)
                     }, {});

    % (q,s) --m--> (p', s'),  (q,s) --n--> checkmark
    % ----------------------------------------------
    % (p || q, s) --m|n--> (p',s')
    R_par_6(p,s) = if( is_par(p)   , { a: ActionTransition |
                        IsOrderedActionList(ac(a))
                     &&
                        exists a1, a2: ActionTransition.
                        a1 in R( pi_1(p), s)
                     && pi_t(a1) == checkmark
                     && a2 in R( pi_2(p), s)
                     && pi_t(a2) != checkmark
                     && OrderAction(ac(a1) ++ ac(a2)) == ac(a)
                     && pi_t(a2) == (pi_t(a))
                     && sigma'(a) == sigma'(a2)
                     }, {});

    % (q,s) --m--> checkmark ,  (q,s) --n--> (q',s')
    % ----------------------------------------------
    % (p || q, s) --m|n--> (q',s')
    R_par_7(p,s) = if( is_par(p)   , { a: ActionTransition |
                        IsOrderedActionList(ac(a))
                     && exists a1, a2: ActionTransition.
                        a1 in R( pi_1(p), s)
                     && pi_t(a1) != checkmark
                     && a2 in R( pi_2(p), s)
                     && pi_t(a2) == checkmark
                     && OrderAction(ac(a1) ++ ac(a2)) == ac(a)
                     && pi_t(a1) == (pi_t(a))
                     && sigma'(a) == sigma'(a1)
                     }, {});

    % (q,s) --m--> (p',s') ,  (q,s) --n--> (q',s'')
    % ----------------------------------------------
    % (p || q, s) --m|n--> (p'||q', s' ++ s'')
    R_par_8(p,s) = if( is_par(p)   , { a: ActionTransition |
                        IsOrderedActionList(ac(a))
                    &&
                        is_par(pi_t(a))
                    && exists a1, a2: ActionTransition.(
                        OrderAction(ac(a1) ++ ac(a2)) == ac(a)
                    && pi_t(a1) != checkmark
                    && pi_t(a2) != checkmark
                    && a2 in R( pi_2(p), s)
                    && a1 in R( pi_1(p), s)
                    && pi_t(a1) == pi_1(pi_t(a))
                    && VariableSubstitutionInProcessTerm( SUBST, pi_t(a2) )
                        == pi_2(pi_t(a))
                    && sigma'(a) ==  OrderValuation( s
                        ++ VariableSubstitutionInValuation( SUBST ,
                            ValuationMinusValuation( sigma'(a2),s ))
                        )
                    whr SUBST =
                      CreateVariableSubstitution(
                        DUP , GenFreshVars(
                        max(GetHighestId( sigma'(a1)),
                            GetHighestId(sigma'(a2)))+ 1, DUP)
                      ) whr DUP = DuplicateVariablesInValuation(
                         ValuationMinusValuation( sigma'(a2) ,s),
                         ValuationMinusValuation( sigma'(a1) ,s)) end
                    end )
                    }, {});

    % (p,s) --m--> checkmark
    % ----------------------
    % (p ||_ q, s) --m--> (q, s)
    R_lmerge_1(p,s) = if( is_lmerge(p), { a: ActionTransition |
                        at(ac(a), checkmark, s ) in R(pi_1(p),s)
                    && pi_t(a) == pi_2(p)
                    && sigma'(a) == s
```

```
560                       }, {});
561
562       % (p,s) --m--> (p',s')
563       % -------------------------------
564       % (p ||_ q, s) --m--> (p' || q, s')
565       R_lmerge_2(p,s) = if( is_lmerge(p), { a: ActionTransition |
566                             is_par(pi_t(a))
567                          && at(ac(a), pi_1(pi_t(a)), sigma'(a)) in R(pi_1(p),s)
568                          && pi_2(pi_t(a)) == pi_2(p)
569                          && pi_1(pi_t(a)) != checkmark
570                          }, {} );
571
572       % (p,s) --m--> checkmark , (q,s) --n--> checkmark
573       % ----------------------------------------------
574       % (p|q, s) --m|n--> checkmark
575       R_sync_1(p,s) = if( is_sync(p), { a: ActionTransition |
576                             IsOrderedActionList(ac(a))
577                          && sigma'(a) == s
578                          && is_checkmark(pi_t(a))
579                          && exists a1, a2: ActionTransition.
580                             a1 in R( pi_1(p), s)
581                          && a2 in R( pi_2(p), s)
582                          && pi_t(a1) == checkmark
583                          && pi_t(a2) == checkmark
584                          && OrderAction(ac(a1) ++ ac(a2)) == ac(a)
585                          }, {});
586
587       % (p,s) --m--> (p',s') , (q,s) --n--> checkmark
588       % ---------------------------------------------
589       % (p|q, s) --m|n--> (p',s')
590       R_sync_2(p,s) = if( is_sync(p), { a: ActionTransition |
591                             IsOrderedActionList(ac(a))
592                          && exists a1, a2: ActionTransition.
593                             a1 in R( pi_1(p), s)
594                          && a2 in R( pi_2(p), s)
595                          && pi_t(a1) != checkmark
596                          && pi_t(a2) == checkmark
597                          && OrderAction(ac(a1) ++ ac(a2)) == ac(a)
598                          && pi_t(a) == pi_t(a1)
599                          && sigma'(a1) == sigma'(a)
600                          }, {});
601
602       % (p,s) --m--> checkmark , (q,s) --n--> (q',s')
603       % ---------------------------------------------
604       % (p|q, s) --m|n--> (q',s')
605       R_sync_3(p,s) = if( is_sync(p), { a: ActionTransition |
606                             IsOrderedActionList(ac(a))
607                          && exists a1, a2: ActionTransition.
608                             a1 in R( pi_1(p), s)
609                          && a2 in R( pi_2(p), s)
610                          && pi_t(a1) == checkmark
611                          && pi_t(a2) != checkmark
612                          && OrderAction(ac(a1) ++ ac(a2)) == ac(a)
613                          && pi_t(a) == pi_t(a2)
614                          && sigma'(a) == sigma'(a1)
615                          }, {});
616
617       % (p,s) --m--> (p,s') , (q,s) --n--> (q',s'')
618       % --------------------------------------------
619       % (p|q, s) --m|n--> (p'||q',s'++s'')
620       R_sync_4(p,s) = if( is_sync(p), { a: ActionTransition |
621                             IsOrderedActionList(ac(a))
622                          && exists a1, a2: ActionTransition. (
623                             a1 in R( pi_1(p), s)
624                          && a2 in R( pi_2(p), s)
625                          && pi_t(a1) != checkmark
626                          && pi_t(a2) != checkmark
627                          && OrderAction(ac(a1) ++ ac(a2)) == ac(a)
628                          && is_par(pi_t(a))
629                          && pi_1(pi_t(a)) == pi_t(a1)
630                          && VariableSubstitutionInProcessTerm( SUBST, pi_t(a2) )
631                             == pi_2(pi_t(a))
632                          && sigma'(a) ==  OrderValuation( s
633                             ++ VariableSubstitutionInValuation( SUBST ,
```

```
                              ValuationMinusValuation( sigma'(a2),s ))
                          )
                      whr SUBST =
                          CreateVariableSubstitution(
                            DUP , GenFreshVars(
                              max(GetHighestId( sigma'(a1)),
                                  GetHighestId(sigma'(a2)))+ 1, DUP )
                          ) whr DUP = DuplicateVariablesInValuation(
                                ValuationMinusValuation( sigma'(a2) ,s),
                                ValuationMinusValuation( sigma'(a1) ,s)) end
                      end)
                      }, {});

    % (p,s) --m--> checkmark,  (m _in_ V +{tau})
    % ------------------------------------------
    % (allow(A,p),s) --m--> checkmark
    R_allow_1(p,s) = if( is_allow(p), { a: ActionTransition |
                          pi_t(a)  == checkmark
                      && a in R( pi_1(p), s)
                      && actionlabels(ac(a)) in (V(p) + {{}})
                      && sigma'(a) == s
                      }, {});

    % (p,s) --m--> (p',s'),  (m _in_ V +{tau})
    % ------------------------------------------
    % (allow(A,p),s) --m--> (allow(A,p'),s')
    R_allow_2(p,s) = if( is_allow(p), { a: ActionTransition |
                          is_allow(pi_t(a))
                      && pi_1(pi_t(a))  != checkmark
                      &&    V(pi_t(a)) == V(p)
                      && at(ac(a),pi_1(pi_t(a)), sigma'(a)) in R( pi_1(p), s)
                      && actionlabels(ac(a)) in (V(p) + {{}})
                      }, {});

    % (p,s) --m--> checkmark,  (m{} * B == {})
    % ------------------------------------------
    % (block(B,p),s) --m--> checkmark
    R_block_1(p,s) = if( is_block(p), { a: ActionTransition |
                          pi_t(a)  == checkmark
                      && a in R( pi_1(p), s)
                      && Bag2Set(actionlabels(ac(a))) * (B(p)) == {}
                      && sigma'(a) == s
                      }, {});

    % (p,s) --m--> checkmark,  (m{} * B == {})
    % ------------------------------------------
    % (block(B,p),s) --m--> (allow(B,p'),s')
    R_block_2(p,s) = if( is_block(p), { a: ActionTransition |
                          is_block(pi_t(a))
                      && pi_1(pi_t(a))  != checkmark
                      &&    B(pi_t(a)) == B(p)
                      && at(ac(a),pi_1(pi_t(a)), sigma'(a)) in R( pi_1(p), s)
                      && Bag2Set(actionlabels(ac(a))) * (B(p)) == {}
                      }, {});

    % (p,s) --m--> checkmark
    % ----------------------------------
    % (rename(R,p),s) --R(m)--> checkmark
    R_rename_1(p,s) = if( is_rename(p), { a: ActionTransition |
                          is_checkmark(pi_t(a))
                      && IsOrderedActionList(ac(a))
                      &&   exists ac': List(ActionSemantic).
                          IsOrderedActionList(ac')
                      && ac(a) == OrderAction(ActRename( Ren(p), ac'))
                      && at(ac', pi_t(a), s) in R( pi_1(p), s)
                      && sigma'(a) == s
                      }, {});

    % (p,s) --m--> (p',s')
    % ------------------------------------------
    % (rename(R,p),s) --Ren(R,m)--> (rename(R,p'),s')
    R_rename_2(p,s) = if( is_rename(p), { a: ActionTransition |
                          Ren(pi_t(a)) == Ren(p)
                      && IsOrderedActionList(ac(a))
```

```
                        && is_rename(pi_t(a))
                        && pi_1(pi_t(a)) != checkmark
                        && exists ac': List(ActionSemantic).
                           IsOrderedActionList(ac')
                        && ac(a) == OrderAction(ActRename( Ren(p), ac'))
                        && at(ac',pi_1(pi_t(a)), sigma'(a)) in R( pi_1(p), s)
                        }, {});

    % (p,s) --m--> checkmark
    % ----------------------------------------
    % (hide(I,p),s) --h(I,m)--> checkmark
    R_hide_1(p,s) = if( is_hide(p), { a: ActionTransition |
                           is_checkmark(pi_t(a))
                        && IsOrderedActionList(ac(a))
                        && exists ac': List(ActionSemantic).
                           ac(a) == ActHide( I(p), ac')
                        && at(ac', pi_t(a), s) in R( pi_1(p), s)
                        && sigma'(a) == s
                        }, {});

    % (p,s) --m--> (p',s')
    % ----------------------------------------
    % (hide(I,p),s) --h(I,m)--> (hide(I,p'),s')
    R_hide_2(p,s) = if( is_hide(p), { a: ActionTransition |
                           I(pi_t(a)) == I(p)
                        && IsOrderedActionList(ac(a))
                        && is_hide(pi_t(a))
                        && pi_1(pi_t(a)) != checkmark
                        && exists ac': List(ActionSemantic).
                           ac(a) == ActHide( I(p), ac')
                        && at(ac',pi_1(pi_t(a)), sigma'(a)) in R( pi_1(p), s)
                        }, {});

    % (p,s) --m--> checkmark
    % ---------------------------------------
    % (prehide(U,p),s) --ph(U,m)--> checkmark
    R_prehide_1(p,s) = if( is_prehide(p), { a: ActionTransition |
                           is_checkmark(pi_t(a))
                        && IsOrderedActionList(ac(a))
                        && exists ac': List(ActionSemantic).
                           ac(a) == ActPrehide( U(p), ac')
                        && at(ac', pi_t(a), s) in R( pi_1(p), s)
                        && sigma'(a) == s
                        }, {});

    % (p,s) --m--> (p',s')
    % -------------------------------------------------
    % (prehide(U,p),s) --ph(U,m)--> (prehide(U,p'),s')
    R_prehide_2(p,s) = if( is_prehide(p), { a: ActionTransition |
                           U(pi_t(a)) == U(p)
                        && IsOrderedActionList(ac(a))
                        && is_prehide(pi_t(a))
                        && pi_1(pi_t(a)) != checkmark
                        && exists ac': List(ActionSemantic).
                           ac(a) == ActPrehide( U(p), ac')
                        && at(ac',pi_1(pi_t(a)), sigma'(a)) in R( pi_1(p), s)
                        }, {});

    % (p,s) --m--> checkmark
    % ----------------------------------
    % (comm(C,p),s) --cm(C,m)--> checkmark
    R_comm_1(p,s) = if( is_comm(p), { a: ActionTransition |
                           is_checkmark(pi_t(a))
                        && IsOrderedActionList(ac(a))
                        &&   exists ac': List(ActionSemantic).
                           at(ac', pi_t(a), s ) in R( pi_1(p), s)
                        && sigma'(a) == s
                        && ac(a) ==  OrderAction(ActComm( CL(p), ac'))
                        }, {});

    % (p,s) --m--> (p',s')
    % ----------------------------------------
    % (comm(C,p),s) --cm(C,m)--> (comm(C,p'),s')
    R_comm_2(p,s) = if( is_comm(p), { a: ActionTransition |
```

```
782                          CL(pi_t(a)) == CL(p)
783                       && IsOrderedActionList(ac(a))
784                       && is_comm(pi_t(a))
785                       && pi_1(pi_t(a)) != checkmark
786                       && exists ac': List(ActionSemantic).
787                          ac(a) == OrderAction(ActComm( CL(p), ac'))
788                       && at(ac',pi_1(pi_t(a)), sigma'(a)) in R( pi_1(p), s)
789                       }, {});
790
791     % (q,s(d:=[[e]](s)) --m--> checkmark
792     % ----------------------------------
793     % (P(e),s) --m--> checkmark
794     R_Def_1(p, s) = if( is_def(p), { a: ActionTransition |
795                          at(ac(a), pi_t(a) , Z ) in R( def( P(p)), Z )
796                       && sigma'(a) == s
797                       && IsOrderedActionList(ac(a))
798                       && is_checkmark(pi_t(a))}, {})
799                       whr Z = OrderValuation(
800                           ComputePPunderInternalValuation( ppl(p),
801                             ToInternalValuation(s) ) ++
802                             RemoveDuplicateVariablesFromFields(ppl(p), s) )
803                       end ;
804
805
806     % (q[d:=d'],s(d':=[[e]](s)) --m--> (q',s')
807     % ----------------------------------
808     % (P(e),s) --m--> (q',s')
809     R_Def_2(p, s) = if( is_def(p), { a: ActionTransition |
810                          a in R( SUBST, REN++s)
811                       && IsOrderedActionList(ac(a))
812                       && !is_checkmark(pi_t(a)) }, {})
813                       whr REN = VariableSubstitutionInValuation(
814                             CreateVariableSubstitution(
815                               GetVarLabelsFromPP( ppl(p) ),
816                               GenFreshVars(GetHighestId(s) + 1,
817                                 GetVarLabelsFromPP(ppl(p)))),
818                             ComputePPunderInternalValuation( ppl(p),
819                               ToInternalValuation(s)))
820                         , SUBST = VariableSubstitutionInProcessTerm(
821                             CreateVariableSubstitution(
822                               GetVarLabelsFromPP( ppl(p) ),
823                               GenFreshVars(GetHighestId(s) + 1,
824                                 GetVarLabelsFromPP(ppl(p)))),
825                             def( P(p) ) )
826                       end;
827  % Function to retrieve the variables from a process parameters
828  map GetVarLabelsFromPP: List(PP) -> List(Variable);
829  var ppl: List(PP);
830      pp : PP;
831  eqn GetVarLabelsFromPP( [] ) = [];
832      GetVarLabelsFromPP( pp |> ppl ) = variable(pp) |> GetVarLabelsFromPP(ppl);
833
834  % Function for generating new variables for a vector of variables.
835  % Identifier starts at value of 'n'
836  map GenFreshVars: Nat#List(Variable)->List(Variable);
837  var vs: List(Variable);
838      v : Variable;
839      n: Nat;
840  eqn GenFreshVars(n,[])        =  [];
841      GenFreshVars(n, v |> vs ) =  GenFreshVar( v,n ) |> GenFreshVars( n, vs );
842
843  % Function to create a variable substitution
844  map CreateVariableSubstitution:
845        List(Variable)#List(Variable) -> (Variable -> Variable);
846      CreateVariableSubstitution:
847        List(Variable)#List(Variable)#(Variable -> Variable) ->
848          (Variable -> Variable);
849  var OldVar: Variable;
850      NewVar: Variable;
851      OldVars: List(Variable);
852      NewVars: List(Variable);
853      VarRename: Variable -> Variable;
854  eqn CreateVariableSubstitution(OldVars, NewVars ) =
855        CreateVariableSubstitution( OldVars, NewVars, lambda v: Variable. (v) );
```

```
    CreateVariableSubstitution([], [], VarRename )  = VarRename;
    CreateVariableSubstitution(OldVar|>OldVars, NewVar|>NewVars, VarRename)=
      CreateVariableSubstitution(OldVars, NewVars, VarRename[OldVar -> NewVar]);

% Function to substitution variables in a mutable valuation
map VariableSubstitutionInValuation: (Variable->Variable)#Valuation-> Valuation;
var VarRename: Variable -> Variable;
    as: Valuation;
    a : Field;
eqn VariableSubstitutionInValuation(VarRename, [] ) = [];
    VariableSubstitutionInValuation(VarRename, a |> as ) =
      field(VarRename(variable(a)),
        valvalue(a)) |> VariableSubstitutionInValuation(VarRename,as);


% Function to substitution variables in a process term
map VariableSubstitutionInProcesParameters:
      (Variable->Variable)#List(PP) -> List(PP);
    VariableSubstitutionInProcessTerm:
      (Variable->Variable)#ProcessTerm -> ProcessTerm;
    VariableSubstitutionInActionList:
      (Variable->Variable)#List(ActionSyntax) -> List(ActionSyntax);
    VariableSubstitutionInAction:
      (Variable->Variable)#ActionSyntax -> ActionSyntax;
    VariableSubstitutionInVariableList:
      (Variable->Variable)#List(Variable) -> List(Variable);
    VariableSubstitutionInDataExpressionList:
      (Variable->Variable)#List(DataExpression) -> List(DataExpression);
var VarRename: Variable -> Variable;
    v: Variable;
    pt1, pt2: ProcessTerm;
    al: List(ActionSyntax);
    a: ActionSyntax;
    vl: List(Variable);
    dl: List(DataExpression);
    d: DataExpression;
    ppl: List(PP);
    pp: PP;
    V:Set(Bag(ActionLabel));
    B:Set(ActionLabel);
    Ren:ActionLabel->ActionLabel;
    I,U:Set(ActionLabel);
    CmI: List(ActionLabel);
    CmR: ActionLabel;
    P:ProcessLabel;
    l: VariableLabel;
    CL: List(Communication);
eqn VariableSubstitutionInVariableList(VarRename, [] ) = [];
    VariableSubstitutionInVariableList(VarRename, v |> vl ) =
      VarRename(v) |> VariableSubstitutionInVariableList(VarRename, vl);
    VariableSubstitutionInDataExpressionList(VarRename, [] ) = [];
    VariableSubstitutionInDataExpressionList(VarRename, d |> dl ) =
      VariableSubstitutionInDataExpression(VarRename, d ) |>
        VariableSubstitutionInDataExpressionList( VarRename, dl );
    VariableSubstitutionInAction( VarRename, a ) =
      Act(actionlabel(a),
        VariableSubstitutionInDataExpressionList(VarRename, args(a) ) );
    VariableSubstitutionInActionList( VarRename, [] ) = [];
    VariableSubstitutionInActionList( VarRename, a |> al) =
      VariableSubstitutionInAction( VarRename, a) |>
        VariableSubstitutionInActionList( VarRename, al);
    VariableSubstitutionInProcessTerm( VarRename, deadlock ) = deadlock;
    VariableSubstitutionInProcessTerm( VarRename, checkmark ) = checkmark;
    VariableSubstitutionInProcessTerm( VarRename, alpha(al) ) =
      alpha( VariableSubstitutionInActionList(VarRename, al ) );
    VariableSubstitutionInProcessTerm( VarRename, seq(pt1, pt2) ) =
      seq( VariableSubstitutionInProcessTerm(VarRename, pt1),
        VariableSubstitutionInProcessTerm(VarRename, pt2) );
    VariableSubstitutionInProcessTerm( VarRename, alt(pt1, pt2) ) =
      alt( VariableSubstitutionInProcessTerm(VarRename, pt1),
        VariableSubstitutionInProcessTerm(VarRename, pt2) );
    VariableSubstitutionInProcessTerm( VarRename, cond1(d, pt1) ) =
      cond1( VariableSubstitutionInDataExpression(VarRename, d),
        VariableSubstitutionInProcessTerm(VarRename, pt1));
```

```
    VariableSubstitutionInProcessTerm( VarRename, cond2(d, pt1, pt2) ) =
      cond2(VariableSubstitutionInDataExpression(VarRename, d),
        VariableSubstitutionInProcessTerm(VarRename, pt1),
        VariableSubstitutionInProcessTerm(VarRename, pt2) );
    VariableSubstitutionInProcessTerm( VarRename, Sum(v, pt1) ) =
      Sum( VarRename(v), VariableSubstitutionInProcessTerm(VarRename, pt1));
    VariableSubstitutionInProcessTerm( VarRename, par(pt1, pt2) ) =
      par( VariableSubstitutionInProcessTerm(VarRename, pt1),
        VariableSubstitutionInProcessTerm(VarRename, pt2) );
    VariableSubstitutionInProcessTerm( VarRename, sync(pt1, pt2) ) =
      sync( VariableSubstitutionInProcessTerm(VarRename, pt1),
        VariableSubstitutionInProcessTerm(VarRename, pt2) );
    VariableSubstitutionInProcessTerm( VarRename, Allow(V, pt1) ) =
      Allow( V, VariableSubstitutionInProcessTerm(VarRename, pt1) );
    VariableSubstitutionInProcessTerm( VarRename, Block(B, pt1) ) =
      Block( B, VariableSubstitutionInProcessTerm(VarRename, pt1) );
    VariableSubstitutionInProcessTerm( VarRename, Hide(I, pt1) ) =
      Hide( I, VariableSubstitutionInProcessTerm( VarRename, pt1 ));
    VariableSubstitutionInProcessTerm( VarRename, Prehide(U, pt1) ) =
      Prehide( U, VariableSubstitutionInProcessTerm( VarRename, pt1 ));
    VariableSubstitutionInProcessTerm( VarRename, Comm( CL, pt1))=
      Comm( CL , VariableSubstitutionInProcessTerm( VarRename, pt1 ));
    VariableSubstitutionInProcessTerm( VarRename, Def( P, ppl)) =
      Def( P, VariableSubstitutionInProcesParameters( VarRename, ppl));
    VariableSubstitutionInProcessTerm( VarRename, Rename(Ren, pt1) ) =
      Rename( Ren, VariableSubstitutionInProcessTerm(VarRename, pt1) );
    VariableSubstitutionInProcesParameters(VarRename, [] ) = [];
    VariableSubstitutionInProcesParameters(VarRename, pp |> ppl ) =
      pp(variable(pp),
        VariableSubstitutionInDataExpression(VarRename, dataexpression(pp))) |>
          VariableSubstitutionInProcesParameters( VarRename, ppl);

% Function to retrieve duplicate variables from a mutable valuation
map DuplicateVariablesInValuation: Valuation#Valuation->List(Variable);
    DuplicateVariablesInValuation: Valuation#Set(Variable)->List(Variable);
    GetVariablesInValuation: Valuation -> Set(Variable);
var fas: Valuation;
    as: Valuation;
    a: Field;
    vs: Set(Variable);
eqn GetVariablesInValuation([])={};
    GetVariablesInValuation(a|> as)={variable(a)}+GetVariablesInValuation(as);
    DuplicateVariablesInValuation( as , fas ) =
      DuplicateVariablesInValuation(as, GetVariablesInValuation( fas) );
    DuplicateVariablesInValuation( [] , vs ) = [] ;
    DuplicateVariablesInValuation( a |> as , vs ) =
      if(variable(a) in vs,
        [variable(a)] , [] ) ++ DuplicateVariablesInValuation( as, vs);

% Function to subtract a mutable valuation from another mutable valuation
map ValuationMinusValuation: Valuation#Valuation->Valuation;
var x: Field;
    xs: Valuation;
    ys: Valuation;
eqn ValuationMinusValuation( [] , ys ) = [];
    ValuationMinusValuation( x |> xs, ys) =
      if(x in ys, ValuationMinusValuation( xs, ys ),
        x |> ValuationMinusValuation( xs, ys ));

% Function to order a mutable valuation
map OrderValuation: Valuation -> Valuation;
    OrderValuation: (Field#Field -> Bool)#Valuation -> List( Field);
var x:Field;
    xs:Valuation;
    pred: Field#Field -> Bool;
eqn OrderValuation ( [] ) = [];
    OrderValuation ( x |> xs) =
      OrderValuation( lambda i,j:Field. (i < j) , x |> xs);
    OrderValuation (pred, [])     = [];
    OrderValuation (pred, x |> xs) =
      InsertField(pred, x, OrderValuation(pred, xs));

% Function to insert a field in a mutable valuation
map InsertField : (Field#Field -> Bool)#Field# Valuation -> Valuation;
```

```
var x,y: Field;
    ys: Valuation;
    pred: Field#Field -> Bool;
eqn InsertField (pred, x, []) = [x];
    pred(x, y)    -> InsertField(pred,x, y|> ys)= x|> y|> ys;
    (!pred(x, y)) -> InsertField(pred,x, y|> ys)= y|> InsertField( pred, x, ys);

% Function to determine that a semantic action list is ordered
map IsOrderedActionList: List(ActionSemantic) -> Bool;
var x,y:ActionSemantic;
    xs:List(ActionSemantic);
eqn IsOrderedActionList([])       = true;
    IsOrderedActionList(x |> [] ) = true;
    IsOrderedActionList(x |> y |> xs)= (x<=y) && IsOrderedActionList( y |> xs );

% Function to order a list of semantic actions
map OrderAction: List(ActionSemantic) -> List(ActionSemantic);
    OrderAction: (ActionSemantic#ActionSemantic -> Bool)#List(ActionSemantic) ->
      List( ActionSemantic);
var x:ActionSemantic;
    xs:List(ActionSemantic);
    pred: ActionSemantic#ActionSemantic -> Bool;
eqn OrderAction (xs)  = OrderAction( lambda i,j:ActionSemantic. (i <= j) ,xs);
    OrderAction (pred, [])     = [];
    OrderAction (pred, x |> xs) = InsertAction(pred, x, OrderAction(pred, xs));

% Function to insert a semantic action into a list of semantic actions
map InsertAction: (ActionSemantic#ActionSemantic -> Bool)#
      ActionSemantic# List(ActionSemantic) -> List(ActionSemantic);
var x,y: ActionSemantic;
    ys: List(ActionSemantic);
    pred: ActionSemantic#ActionSemantic -> Bool;
eqn InsertAction (pred, x, []) = [x];
    pred(x, y)   ->InsertAction(pred, x, y|> ys)=x |> y |> ys;
    (!pred(x, y))->InsertAction(pred, x, y|> ys)=y |> InsertAction(pred, x, ys);

% Function to transform a list of process parameters
map ComputePPunderInternalValuation: PP#InternalValuation->Field;
    ComputePPunderInternalValuation: List(PP)#InternalValuation->Valuation;
var p: PP;
    pl: List(PP);
    s: InternalValuation;
eqn ComputePPunderInternalValuation(p,s) =
      field( variable(p), sem_Dex( dataexpression(p), s));
    ComputePPunderInternalValuation([], s) = [];
    ComputePPunderInternalValuation(p |> pl, s) =
      ComputePPunderInternalValuation(p,s)|>
        ComputePPunderInternalValuation(pl,s);

% Function that preserves all fields in a valuation, for which the
% variables do not occur as a left hand side variable in a list
% of process parameters
map  RemoveDuplicateVariablesFromFields: List(PP)#Valuation -> Valuation;
     AddIfNonDup: List(PP)#Field -> Valuation;
var  pl: List(PP);
     p : PP;
     lass: Valuation;
     ass: Field;
eqn  RemoveDuplicateVariablesFromFields( pl , [] ) = [];
     RemoveDuplicateVariablesFromFields( pl , ass |> lass ) =
       AddIfNonDup(pl, ass) ++ RemoveDuplicateVariablesFromFields(pl , lass);
     AddIfNonDup( []      , ass  ) = [ass];
     AddIfNonDup( p |> pl , ass  ) =
       if( variable(p) == variable(ass) , [] , AddIfNonDup( pl, ass));

% Transition relation function
act a: List(ActionSemantic);

% Linear Process Equation
proc X(p: ProcessTerm, s: Valuation ) =
       sum r: ActionTransition. ( r in R(p, s) )-> a( ac(r) ).
         X( pi_t(r), sigma'(r) );
```

A.2 Model specific semantics

The model specific semantics describe the semantics that are specific for a set of models. Within this semantics we describe the allowed actions, variables, and (user defined) sorts, and the system of process equations.

```
% Sort for variables
sort Variable = struct bool( variablelabel:VariableLabel )?is_bool
                     | nat( variablelabel:VariableLabel)?is_nat;

% Sort for values
sort Value   = struct bot | bool'( b:Bool )?is_bool    | nat'( n:Nat)?is_nat;

% Sort for process equation labels
sort ProcessLabel = struct p0 | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 | P | Q;

% Sort for action labels
sort ActionLabel = struct a | b | a1 | a2 | a3 | a4 | int;

% Sort for variable labels.
% Note that d'(Nat) may only be used to generate fresh variables
sort VariableLabel = struct v| v1 | v2 | v3 | d'(id:Nat)?is_d';

% Process Equations
map def: ProcessLabel -> ProcessTerm;
eqn def( p0 ) = alpha([Act( a1, [de_var(bool(v1))] )]);
    def( p1 ) = seq( alpha([Act( a1, [] )]),
                  par( alpha( [Act( a2, [] )]), Def( p1, [] )));
    def( p2 ) = seq( alpha([Act( a1, [de_var(bool(v1))] )]),
                  seq( Def( p3, [pp(bool(v1), de_val(bool'(false)))]),
                    alpha( [Act( a1, [de_var(bool(v1))] )] )));
    def( p3 ) = alpha([Act( a2, [de_var(bool(v1))] )]);
    def( p4 ) = seq( alpha([Act( a2, [de_var(bool(v1))] )]),
                  alpha([Act( a2, [de_var(bool(v1))] )]));
    def( p5 ) = seq( alpha([Act( a1, [de_var(bool(v1))] )]),
                  seq( Def( p4, [pp(bool(v1),de_val(bool'(false)))]),
                    alpha( [Act( a1, [de_var(bool(v1))] )] ) ) );
    def( p6 ) = cond1( de_var(bool(v1)) , alpha([Act(a3, [de_var(bool(v1))])]));
    def( p7 ) = seq( alpha([Act( a1, [de_var(bool(v1))] )]),
                  Def( p7,
                    [pp(bool(v1), de_expr_1( bool_op(neg), de_var(bool(v1))))]));
    def( p8 ) = seq( alpha([Act( a1, [de_var(bool(v1))] )]),
                  alpha([Act( a2, [de_var(bool(v1))] )]) );
    def( P ) = Sum( bool(v),  seq(alpha([ Act(a, [de_var(bool(v))])]),
                  alpha([Act(b, [ de_var(bool(v)) ] )] )));
    def( Q ) = alt(seq(alpha([ Act(a, [de_val(bool'(true))])]),
                   alpha([Act(b, [ de_val(bool'(true)) ] )] ))
                 , seq(alpha([ Act(a, [de_val(bool'(false))])]),
                   alpha([Act(b, [ de_val(bool'(false)) ] )])));

% Restrict the selection of a value to a particular domain
map RestrictDomain: Variable#Value   -> Bool;
var f: Func;
    v1,v2: Value;
    v: Variable;
    w: Value;
    vs: List(Variable);
    ws: List(Value);
eqn RestrictDomain( v, w ) =
      (is_bool(v) && is_bool(w)) || (is_nat(v) && is_nat(w));

%% Generate a fresh variable with an appropriate sort
map GenFreshVar:Variable#Nat-> Variable;
var l: VariableLabel;
    vid: Nat;
eqn GenFreshVar( bool(l), vid ) = bool(d'(vid));
    GenFreshVar( nat(l), vid )  = nat(d'(vid));


% Sorts for operators
sort Operator = struct neg | and | or | eq ;
sort Func     = struct bool_op( op: Operator )?is_bool
                     | nat_op( op: Operator )?is_nat
```

```
68                      ;
69
70  % Function to interpret meta notation data expression into values
71  map sem_Dex: DataExpression#InternalValuation -> Value;
72  var d: Variable;
73      dvl: Value;
74      expr1: DataExpression;
75      expr2: DataExpression;
76      sigma: InternalValuation;
77  eqn sem_Dex( de_var(d), sigma) = sigma(d);
78      sem_Dex( de_val(dvl), sigma ) = dvl;
79      sem_Dex( de_expr_1( bool_op(neg), expr1), sigma ) =
80        bool'(!Cast2InternalBool(sem_Dex(expr1, sigma)));
81      sem_Dex( de_expr_2( bool_op(and), expr1, expr2), sigma ) =
82        bool'(Cast2InternalBool(sem_Dex(expr1, sigma)) &&
83          Cast2InternalBool(sem_Dex(expr2, sigma)));
84      sem_Dex( de_expr_2( bool_op(or), expr1, expr2), sigma ) =
85        bool'(Cast2InternalBool(sem_Dex(expr1, sigma)) ||
86          Cast2InternalBool(sem_Dex(expr2, sigma)));
87      sem_Dex( de_expr_2( bool_op(eq), expr1, expr2), sigma ) =
88        bool'(Cast2InternalBool(sem_Dex(expr1, sigma)) ==
89          Cast2InternalBool(sem_Dex(expr2, sigma)));
90
91  % Function to cast meta data expressions to mcrl2 data expressions
92  map Cast2InternalBool: Value->Bool;
93      Cast2InternalNat: Value->Nat;
94  var b: Bool;
95      n: Nat;
96  eqn Cast2InternalBool(bool'(b)) = b;
97      Cast2InternalNat(nat'(n))   = n;
98
99  % Function to substitute variables in data expressions
100 map VariableSubstitutionInDataExpression:
101       (Variable->Variable)#(DataExpression) -> (DataExpression);
102 var VarRename: Variable -> Variable;
103     value: Value;
104     v: Variable;
105     f: Func;
106     expr1, expr2: DataExpression;
107 eqn VariableSubstitutionInDataExpression(VarRename, de_val(value))=
108       de_val(value);
109     VariableSubstitutionInDataExpression(VarRename, de_var(v)) =
110       de_var(VarRename(v));
111     VariableSubstitutionInDataExpression(VarRename, de_expr_1(f, expr1))=
112       de_expr_1(f, VariableSubstitutionInDataExpression( VarRename, expr1));
113     VariableSubstitutionInDataExpression(VarRename, de_expr_2(f, expr1, expr2))=
de_expr_2(f, VariableSubstitutionInDataExpression( VarRename, expr1),
114         VariableSubstitutionInDataExpression( VarRename, expr2));
```

A.3 Models used for input

The models that have served for input are described here. Each $i$nit represents a different mCRL2 model in meta notation. The LPE that is used to generate the transitions carries the process label X. The LPE contains two arguments. The first argument denotes the mCRL2 model written in the meta notation. The second argument denotes the initial valuation for the model.

```
1  % Multi action tests
2  init X(alpha([Act(a1,[])]),[]);
3  init X(alpha([]),[]);
4  init X(alpha([ActionTau]),[]);
5  init X(alpha([Act(a1,[]),ActionTau]),[]);
6  init X(alpha([Act(a1,[de_var(bool(v1))])]),[field(bool(v1),bool'(true))]);
7
8  % Alternative composition tests
9  init X(alt(alpha([Act(a1,[])]),deadlock),[]);
10 init X(alt(alpha([Act(a1,[])]),alpha([Act(a2,[])])),[]);
11
12 % Sequential composition tests
```

```
13  init X(seq(alpha([Act(a1,[])]),alpha([Act(a2,[])])),[]);
14  init X(seq(alpha([Act(a1,[])]),
15         eq(alpha([Act(a2,[])]),alpha([Act(a2,[])]))),[]);
16  init X(seq(seq(alpha([Act(a1,[de_var(bool(v1))])]),
17                alpha([Act(a2,[de_var(bool(v1))])])),
18           alpha([Act(a3,[de_var(bool(v1))])])),
19           [field(bool(v1),bool'(true))]);
20  init X(seq(alpha([Act(a1,[])]),seq(alpha([Act(a2,[])]),
21           alpha([Act(a3,[])]))),[field(bool(v1),bool'(true))]);
22  init X(alt(alpha([]),seq(alpha([Act(a1,[])]),deadlock)),[]);
23
24  % Sum tests
25  init X(Sum(bool(v1),alpha([Act(a3,[de_var(bool(v1))])])),[]);
26  init X(Sum(bool(v1),seq(alpha([Act(a3,[de_var(bool(v1))])]),deadlock)),[]);
27  init X(Sum(bool(v1),seq(alpha([Act(a3,[de_var(bool(v1))])]),checkmark)),[]);
28  init X(Sum(bool(v1),seq(alpha([Act(a3,[de_var(bool(v1))])]),
29         alpha([Act(a3,[])]))),[]);
30  init X(Sum(bool(v1),seq(alpha([Act(a3,[de_var(bool(v1))])]),
31         alpha([Act(a3,[de_var(bool(v1))])]))),[]);
32  init X(Sum(bool(v1),seq(Sum(bool(v1),
33         alpha([Act(a1,[de_var(bool(v1))])]),
34         alpha([Act(a3,[de_var(bool(v1))])]))),[]);
35
36  % Condition tests
37  init X(cond1(de_var(bool(v1)),alpha([Act(a3,[de_var(bool(v1))])])),
38         [field(bool(v1),bool'(true))]);
39  init X(cond1(de_var(bool(v1)),alpha([Act(a3,[de_var(bool(v1))])])),
40         [field(bool(v1),bool'(true))]);
41  init X(cond1(de_var(bool(v1)),alpha([Act(a3,[de_var(bool(v1))])])),
42         [field(bool(v1),bool'(false))]);
43  init X(cond1(de_expr_1(bool_op(neg),de_var(bool(v1))),
44         alpha([Act(a3,[de_var(bool(v1))])])),
45           [field(bool(v1),bool'(false))]);
46  init X(cond2(de_var(bool(v1)),alpha([Act(a1,[de_var(bool(v1))])]),
47         alpha([Act(a2,[de_var(bool(v1))])])),
48           [field(bool(v1),bool'(true))]);
49  init X(cond2(de_var(bool(v1)),alpha([Act(a1,[de_var(bool(v1))])]),
50         alpha([Act(a2,[de_var(bool(v1))])])),
51           [field(bool(v1),bool'(false))]);
52  init X(Sum(bool(v1),cond1(de_var(bool(v1)),
53         alpha([Act(a3,[de_var(bool(v1))])]))),[]);
54  init X(Sum(bool(v1),cond2(de_var(bool(v1)),
55         alpha([Act(a1,[de_var(bool(v1))])]),
56           alpha([Act(a3,[de_var(bool(v1))])]))),[]);
57  init X(Sum(bool(v1),cond2(de_expr_1(bool_op(neg),de_var(bool(v1))),
58         alpha([Act(a1,[de_var(bool(v1))])]),
59           alpha([Act(a3,[de_var(bool(v1))])]))),[]);
60
61  % Parallel tests
62  init X(par(alpha([Act(a1,[])]),alpha([Act(a2,[])])),[]);
63  init X(par(seq(alpha([Act(a1,[])]),alpha([Act(a2,[])])),
64         seq(alpha([Act(a3,[])]),alpha([Act(a4,[])]))),[]);
65
66  % Sync tests
67  init X(sync(alpha([Act(a1,[])]),alpha([Act(a2,[])])),[]);
68  init X(sync(seq(alpha([Act(a1,[])]),alpha([Act(a2,[])])),alpha([Act(a3,[])])),[]);
69  init X(sync(alpha([Act(a1,[])]),seq(alpha([Act(a2,[])]),alpha([Act(a3,[])]))),[]);
70  init X(sync(seq(alpha([Act(a1,[])]),alpha([Act(a2,[])])),
71          seq(alpha([Act(a3,[])]),alpha([Act(a4,[])]))),[]);
72
73  % Left merge tests
74  init X(lmerge(alpha([Act(a1,[])]),alpha([Act(a2,[])])),[]);
75  init X(lmerge(seq(alpha([Act(a1,[])]),alpha([Act(a2,[])])),alpha([Act(a3,[])])),[]);
76  init X(lmerge(alpha([Act(a1,[])]),seq(alpha([Act(a2,[])]),alpha([Act(a3,[])]))),[]);
77  init X(lmerge(seq(alpha([Act(a1,[])]),alpha([Act(a2,[])])),
78          seq(alpha([Act(a3,[])]),alpha([Act(a4,[])]))),[]);
79
80  % Allow tests
81  init X(Allow({},alpha([Act(a1,[])])),[]);
82  init X(Allow({{a2:1}},alpha([Act(a1,[])])),[]);
83  init X(Allow({{a1:1}},alpha([Act(a1,[])])),[]);
84  init X(Allow({{a1:1}},seq(alpha([Act(a1,[])]),alpha([Act(a1,[])]))),[]);
85  init X(Allow({{a2:1}},seq(alpha([Act(a1,[])]),alpha([Act(a2,[])]))),[]);
86  init X(Allow({{a1:1}},seq(alpha([Act(a1,[])]),alpha([Act(a2,[])]))),[]);
```

```
87
88  % Block tests
89  init X(Block({a1},alpha([Act(a1,[])])),[]);
90  init X(Block({a2},alpha([Act(a1,[])])),[]);
91  init X(Block({a2},seq(alpha([Act(a1,[])]),alpha([Act(a2,[])]))),[]);
92  init X(Block({a2},alpha([Act(a1,[]),Act(a2,[])])),[]);
93
94  % Action rename tests
95  init X(Rename(ID[a2->a1],alpha([Act(a2,[])])),[]);
96  init X(Rename(ID[a2->a1],seq(alpha([Act(a2,[])]),alpha([Act(a2,[])]))),[]);
97
98  % Prehide tests
99  init X(Prehide({a2},alpha([Act(a2,[])])),[]);
100 init X(Prehide({a2},seq(alpha([Act(a2,[])]),alpha([Act(a1,[])]))),[]);
101
102 % Hide tests
103 init X(Hide({a2},alpha([Act(a2,[])])),[]);
104 init X(Hide({a2},seq(alpha([Act(a2,[])]),alpha([Act(a1,[])]))),[]);
105
106 % Communication tests
107 init X(Comm([communication([a2,a2],a1)],
108         alpha([Act(a2,[]),Act(a1,[]),Act(a2,[])])),[]);
109 init X(Comm([communication([a2,a2],a1)],
110         alpha([Act(a2,[]),Act(a1,[]),Act(a2,[de_var(bool(v1))])])),
111       [field(bool(v1),bool'(true))]);
112
113 % Process Equation tests
114 init X(Def(p0,[pp(bool(v1),de_val(bool'(false)))]),[field(bool(v1),bool'(true))]);
115 init X(Def(p0,[pp(bool(v1),de_var(bool(v2)))]),[field(bool(v1),bool'(true))]);
116 init X(Def(p0,[pp(bool(v2),de_var(bool(v1)))]),[field(bool(v1),bool'(true))]);
117 init X(Def(p2,[]),[field(bool(v1),bool'(true))]);
118 init X(Def(p5,[]),[field(bool(v1),bool'(true))]);
119 init X(Def(p4,[pp(bool(v1),de_val(bool'(false)))]),[field(bool(v1),bool'(true))]);
120 init X(seq(Def(p4,[pp(bool(v1),de_val(bool'(false)))]),
121        alpha([Act(a2,[de_var(bool(v1))])])),[field(bool(v1),bool'(true))]);
122 init X(Def(p6,[pp(bool(v1),de_val(bool'(true)))]),
123        [field(bool(v1),bool'(false))]);
124 init X(Sum(bool(v1),Sum(bool(v2),
125        alpha([Act(a1,[de_var(bool(v1)),de_var(bool(v2))])]))),[]);
126 init X(par(Def(p0,[pp(bool(v1),de_val(bool'(false)))]),
127         Def(p0,[pp(bool(v1),de_val(bool'(true)))])),[]);
128 init X(par(Def(p8,[pp(bool(v1),de_val(bool'(true)))]),
129         Def(p8,[pp(bool(v1),de_val(bool'(true)))])),[]);
130 init X(par(Def(p8,[pp(bool(v1),de_val(bool'(false)))]),
131        Def(p8,[pp(bool(v1),de_val(bool'(true)))])),[]);
132 init X(Def(P,[]),[]);
133 init X(Def(Q,[]),[]);
134 init X(par(Def(P,[]),Def(P,[])),[]);
135 init X(par(Def(Q,[]),Def(Q,[])),[]);
```

Bibliography

[1] L. Aceto and M. Hennessy. Termination, Deadlock and Divergence. In M.G. Main, A. Melton, M.W. Mislove, and D.A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 301–318. Springer, 1989.

[2] D.A. van Beek, K.L. Man, , J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *J. Log. Algebr. Program.*, 68(1-2):129–210, 2006.

[3] D.A. van Beek, M.A. Reniers, R.R.H. Schiffelers, and J.E. Rooda. Foundations of a Compositional Interchange Format for Hybrid Systems. In A. Bemporad, A. Bicchi, and G.C. Buttazzo, editors, *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 587–600. Springer, 2007.

[4] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[5] J.A. Bergstra, A. Ponse, and J. van Wamel. Process Algebra with Backtracking. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *REX School/Symposium*, volume 803 of *Lecture Notes in Computer Science*, pages 46–91. Springer, 1993.

[6] M. Bezem, R.N. Bol, and J.F. Groote. Formalizing Process Algebraic Verifications in the Calculus of Constructions. *Formal Asp. Comput.*, 9(1):1–48, 1997.

[7] R.N. Bol and J.F. Groote. The Meaning of Negative Premises in Transition System Specifications. *J. ACM*, 43(5):863–914, 1996.

[8] C. Braga and A. Verdejo. Modular Structural Operational Semantics with Strategies. *Electr. Notes Theor. Comput. Sci.*, 175(1):3–17, 2007.

[9] C.O. Braga, E.H. Haeusler, J. Meseguer, and P.D. Mosses. Mapping Modular SOS to Rewriting Logic. In M. Leuschel, editor, *LOPSTR*, volume 2664 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002.

[10] J. Carroll. Microsoft, eat your own dog food.

[11] R. Cleaveland and M. Hennessy. Priorities in Process Algebras. *Inf. Comput.*, 87(1/2):58–77, 1990.

[12] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP - A Protocol Validation and Verification Toolbox. In R. Alur and T.A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer, 1996.

[13] W. Fokkink. *Modelling Distributed Systems.* Springer Berlin Heidelberg, 2007.

[14] M. Goldsmith and I. Zakiuddin. Critical Systems Validation and Verification with CSP and FDR. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *FM-Trends*, volume 1641 of *Lecture Notes in Computer Science*, pages 243–250. Springer, 1998.

[15] J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, W. Wesselink, and T.A.C. Willemse. Experiences in developing the mCRL2 toolset. *Softw., Pract. Exper.*, 41(2):143–153, 2011.

[16] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M. van Weerdenburg. The Formal Specification Language mCRL2. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[17] J.F. Groote, A.H.J. Mathijssen, M. van Weerdenburg, and Y.S. Usenko. From μCRL to mCRL2: Motivation and Outline. *Electr. Notes Theor. Comput. Sci.*, 162:191–196, 2006.

[18] J.F. Groote and F.W. Vaandrager. Structured Operational Semantics and Bisimulation as a Congruence. *Inf. Comput.*, 100(2):202–260, 1992.

[19] K. Haaland. JIT Software Development-Inside the Eclipse Software Development Process.

[20] W. Harrison. Eating Your Own Dog Food. *IEEE Software*, 23(3):5–7, 2006.

[21] M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Using Maude and Its Strategies for Defining a Framework for Analyzing Eden Semantics. *Electr. Notes Theor. Comput. Sci.*, 174(10):119–137, 2007.

[22] G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[23] J.J.A. Keiren and M.A. Reniers. Type checking mCRL2. Technical Report 11, Technische Universiteit Eindhoven, 2011.

[24] K.G. Larsen and A. Skou. Compositional Verification of Probabilistic Processes. In R. Cleaveland, editor, *CONCUR*, volume 630 of *Lecture Notes in Computer Science*, pages 456–471. Springer, 1992.

[25] R. Mariani. My History of Visual Studio (Part 10, final) - https://blogs.msdn.com/ricom/archive/2009/10/19/my-history-of-visual-studio-part-10-final.aspx.

[26] J. McLoone. Eating Your Own Dogfood, May 2007.

[27] P.D. Mosses. Exploiting labels in Structural Operational Semantics. In H. Haddad, A. Omicini, R.L. Wainwright, and L.M. Liebrock, editors, *SAC*, pages 1476–1481. ACM, 2004.

[28] P.D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[29] M.R. Mousavi, M.A. Reniers, and J.F. Groote. SOS formats and meta-theory: 20 years after. *Theor. Comput. Sci.*, 373(3):238–272, 2007.

[30] P.C Ölveczky, A. Boronat, and J. Meseguer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time maude. In J. Hatcliff and E. Zucca, editors, *FMOODS/FORTE*, volume 6117 of *Lecture Notes in Computer Science*, pages 47–62. Springer, 2010.

[31] G.D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[32] M. Queiroz. An Android dogfood diet for the holidays.

[33] M.A. Reniers, J.F. Groote, M. van der Zwaag, and J. van Wamel. Completeness of Timed mCRL. *Fundam. Inform.*, 50(3-4):361–402, 2002.

[34] A. Riesco and J. Rodríguez-Hortalá. A Natural Implementation of Plural Semantics in Maude. *Electr. Notes Theor. Comput. Sci.*, 253(7):165–175, 2010.

[35] R. de Simone. Higher-Level Synchronising Devices in Meije-SCCS. *Theor. Comput. Sci.*, 37:245–267, 1985.

[36] F.P.M. Stappers, M.A. Reniers, and S. Weber. Transforming SOS specifications to linear processes. Technical Report 11-07, Technische Universiteit Eindhoven, 2011.

[37] F.P.M. Stappers, M.A. Reniers, and Sven Weber. Transforming SOS Specifications to Linear Processes. In G. Salaün and B. Schätz, editors, *FMICS*, volume 6959 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 2011.

[38] F.P.M. Stappers, S. Weber, M.A. Reniers, S. Andova, and I. Nagy. Formalizing a domain specific language using SOS: An industrial case study. In Uwe Aßmann and Anthony Sloane, editors, *Post-proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, volume 6940 of *LNCS*, page ? Springer, Heidelberg, August 2011.

[39] P.D. Terry. *Compilers and Compiler Generators: An Introduction with C++*. Coriolis Group, March 1997.

[40] the mCRL2 toolset. http://www.mcrl2.org.

[41] Y.S. Usenko. *Linearization in μCRL*. PhD thesis, Eindhoven University of Technology, December 2002.

[42] J.P.M. Voeten, P.H.A. van der Putten, M.C.W. Geilen, and M.P.J. Stevens. Formal Modelling of Reactive Hardware/software Systems. In *in J.P. Veen, Ed., Proceedings of ProRISC/IEEE'97, Utrecht : STW, Technology Foundation*, pages 663–670, 1997.