Dogfooding the formal semantics of mCRL2

F.P.M. Stappers Dept. of Mathematics and Computer Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
M.A. Reniers Dept. of Mechanical Engineering, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
S. Weber Dept. for Machine Control and Infrastructure, ASML,
P.O. Box 324, NL-5500 AH Veldhoven, The Netherlands
J.F. Groote Dept. of Mathematics and Computer Science, Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

***Abstract*—The mCRL2 language is a formal specification language that is used to specify, model, analyze and verify behavioral properties for distributed systems and protocols. The semantics of the mCRL2 language is defined formally using Structural Operational Semantics (SOS). In [32] we propose an approach that takes the SOS of a formal language, along with a concrete model, that serves as an initialization, and transforms it to a Linear Process Specification (LPS). In this paper we extend the approach and show that it can be applied to a formal language that is actually used to specify and model discussed systems. Hence, we take mCRL2's own operational semantics and transform it into an mCRL2 specification. In essence, this means that we are feeding the mCRL2 toolset its own formal language definition. This semantic dogfooding approach validates the implemented behavior for the mCRL2 language against its formal definition. By performing this exercise we revealed gaps between the defined and implemented semantics. These gaps have subsequently been resolved.**

I. Introduction

Embedded systems have become ubiquitous and modern society has become – to some extent – dependent on them. This means that we require them to always function 'correctly'. To *guarantee* correctness under *all* possible circumstances, one could formally analyze a model that captures the essential system behavior in an unambiguous way. This requires that a behavioral model is written in a specification language for which the syntax and semantics are defined formally.

In [32] the authors propose a framework that transforms the Structural Operational Semantics (SOS) [26] of a formal specification language into a restricted mCRL2 [14] specification (i.e., Linear Process Specification (LPS) [4], [12]). The mCRL2 language is a formal specification language for modeling the behavior of distributed systems and protocols. Using the underlying mCRL2 toolset [1], mCRL2 specifications can be simulated or model checked (if a modal property is provided).

The initial framework presented in [32], [33] considers only the transformation of SOS deduction rules in the De Simone format as a first step. This work has been used in [35] to model the SOS of a domain specific language (DSL), by which we successfully verified the behavior of domain specific models. Because we wanted to show the applicability and extensibility of the approach, this paper considers the mCRL2 language as another instance of a formal language on top of our framework, see Figure 1. Because the mCRL2 language uses semantic notations that have not been considered so far, we have extended the initial transformation framework. These extensions are detailed in [30], [31] and include the use of data valuations, data parameters in action transitions, multi-actions, action renaming and the generation of fresh variables.

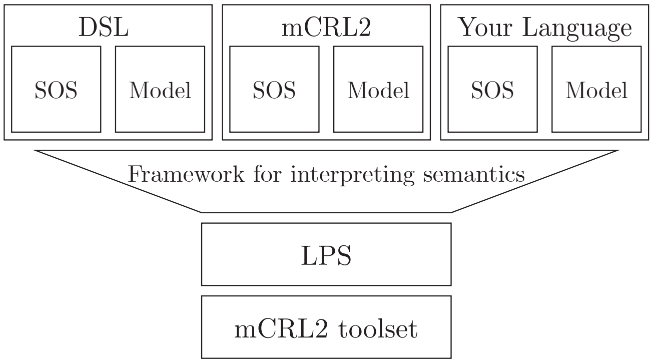

Fig. 1: Schematic overview of our approach

The formal semantics of the mCRL2 language is given in SOS whereas the execution behavior of the supporting mCRL2 toolset is manually written in the C++ language. This means that no automated coupling exists between the formal language definition and its manual implementation. To investigate the correspondence we apply our extended framework to transform mCRL2's SOS, along with a specification – i.e., a model – into an mCRL2 specification. Because we model all transformation rules directly in the mCRL2 language, we force the underlying toolset to eat, rewrite and execute its own formal language definition. This technique is commonly applied in industry [9], [16], [21], [27] and is known as "eating your own dogfood" [17] or *dogfooding* for short.

In this paper we effectively demonstrate that our framework is extensible and mature enough to capture the rich semantics of a language like mCRL2. Moreover, the semantic dogfooding approach reveals that – despite the formal characterization of the mCRL2 language and its thorough study and broad use by many students and researchers – a number of (subtle) differences existed between (i) the intended formal semantics, (ii) the specified semantics, (iii) the implemented semantics and (iv) the modeled semantics. The discovered issues have been addressed and subsequently been resolved.

This paper is structured as follows. Section II describes the preliminaries on SOS and the relevant syntactic and semantic concepts of the mCRL2 language. Section III outlines the modeling of the mCRL2 syntax and semantics in an mCRL2 specification. Section IV presents the results, includes dogfooding examples, and shows the discovered issues. Section V positions our work and Section VI presents our conclusions and elaborates on future work.

II. Preliminaries

A. Structural Operational Semantics

Structural Operational Semantics (SOS) specifies the actions a piece of syntax can perform, typically represented by a Transition System Specification (TSS) [5], [13]. A signature defines the syntax for which the semantics is defined. A signature fixes the composition operators and their corresponding arities, for which we assume a set of variables $\mathcal{V}_{\text{SOS}}$ and a set of action labels $\mathcal{A}_{\text{SOS}}$.

A *signature* Σ_{SOS} consists of (i) a collection $\mathcal{S}_{\text{SOS}}$ of sorts represented by $S, S_1, \ldots, S_n$, and (ii) a collection of function symbols together with their arities. The collection of *terms* $\mathcal{T}(\Sigma_{\text{SOS}})$ over signature Σ_{SOS} is the smallest set such that (i) a variable $x \in \mathcal{V}^{S}_{\text{SOS}}$ (set of variables restricted to sort S) is a term of sort S, and (ii) $f(t_1, \ldots, t_n)$ is a term of sort S, if $t_1, \ldots, t_n$ are terms, where t_i is a term of sort S_i and $f \in \Sigma_{\text{SOS}}$ is an n-ary function symbol of sort $S_1 \times \cdots \times S_n \rightarrow S$.

A *valuation* $\sigma : \mathcal{V}^{S}_{\text{SOS}} \rightarrow M_S$ is a sort preserving function from variables to a collection of modeled values of sort S, represented by M_S. Let $p, p' \in \mathcal{T}(\Sigma_{\text{SOS}})$ be terms, let $l \in \mathcal{A}_{\text{SOS}}$, and let $\sigma, \sigma' : \mathcal{V}^{S}_{\text{SOS}} \rightarrow M_S$, then a *transition formula* is of the form $(p, \sigma) \xrightarrow{l} (p', \sigma')$.

A *Transition System Specification* (TSS) [5], [13] denotes a set of deduction rules. It is defined by a tuple $(\Sigma_{\text{SOS}}, \mathcal{D}_{\text{SOS}})$, where Σ_{SOS} is a signature and $\mathcal{D}_{\text{SOS}}$ is a set of deduction rules. A deduction rule is of the form $\frac{H}{C}$ where H is a set of transition formulas or Boolean expressions, called the set of *premises*, and C is a transition formula, called the *conclusion*. To derive the conclusion, and perform an action, all premises need to be satisfied.

B. The mCRL2 Language

The mCRL2 language is action based and geared towards the specification and behavioral verification of distributed systems, parallel computer programs and protocols. To describe the specifications the mCRL2 language uses a formal syntax and semantics that are rich. Although we have dogfed the entire untimed part of the mCRL2 language, we only introduce those syntactic and semantic notions that are needed for showing the discovered issues by the (entire) dogfooding approach. The full syntax and semantics can be found in [15], [30]. The entire dogfooding approach can be found in [30], [31].

1) Syntactic Concepts: An mCRL2 *process specification* is a five tuple $PS = (\mathcal{D}^{\text{mCRL2}}, AD, PE, p, \mathcal{X}^{\text{mCRL2}})$ where $\mathcal{D}^{\text{mCRL2}}$ is a data specification, AD is an action declaration, PE is a set of process equations, p is a process expression and $\mathcal{X}^{\text{mCRL2}}$ is a set of global variables. Here $\mathcal{D}^{\text{mCRL2}} = (\Sigma^{\text{mCRL2}}, E)$ consists of a *signature* Σ^{mCRL2} and a set of *data equations* E. A signature $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ consists of a set of sorts names $\mathcal{S}^{\text{mCRL2}}$, a set of constructor functions $\mathcal{C}^{\text{mCRL2}}$ and a set of mappings $\mathcal{M}^{\text{mCRL2}}$.

a) Sorts: The mCRL2 data specification incorporates built-in (container) sorts, like $\mathbb{B}, \mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathit{Set}, \mathit{List}$ along with their conventional functions for manipulation, but also allows the specification of user defined sorts and user defined functions to manipulate data. Within this paper we use the standard available (container) sorts, mappings and constructor functions to model the syntax and semantics. In addition, we also use structured sorts that are of the form:

$$\begin{aligned}\textbf{sort}\quad S = \textbf{struct}\quad & c_1(\pi^1_1{:}D^1_1, \ldots, \pi^1_n{:}D^1_n)?is_{c_1} \mid \ldots \mid \\ & c_n(\pi^n_1{:}D^n_1, \ldots, \pi^n_n{:}D^n_n)?is_{c_n};\end{aligned}$$

Here S denotes the sort name for the structured sort. Then for $1 \leq i, j \leq n$, $c_i \in \mathcal{C}^{\text{mCRL2}}$ denotes a constructor function, π^j_i denotes the j^{th} projection function $S \rightarrow D^i_j$ for the i^{th} element, and is_{c_i} denotes the recognizer function that only evaluates to *true* if the term is headed by the constructor function c_i.

Example A sort *Tree* for storing natural numbers in the leaves can be modeled as:

$$\begin{aligned}\textbf{sort}\quad \mathit{Tree} \;=\; \textbf{struct}\; & \mathit{leaf}(\mathit{val}{:}\mathbb{N})?is_{\mathit{leaf}} \mid \\ & \mathit{node}(\mathit{left}{:}\mathit{Tree}, \mathit{right}{:}\mathit{Tree})?is_{\mathit{node}};\end{aligned}$$

A tree only containing the leaves 1 and 2 can be modeled as $\mathit{node}(\mathit{leaf}(1), \mathit{leaf}(2))$. If $t \in \mathit{Tree}$, then we can test if t is a node by $is_{\mathit{node}}(t)$ using the recognizer function, or obtain t's left branch by using the projection function $\mathit{left}(t)$.

b) Data: Data expressions in mCRL2 specifications are expressions, that are inductively constructed from variables and function symbols. Such data expressions have to be well-typed, see [20]. They are used to influence the behavior of processes. Data expressions can be manipulated using data equations. Data equations are of the form:

$$\begin{aligned}&\textbf{map}\quad f : D_1 \times \cdots \times D_n \rightarrow D;\\ &\textbf{var}\quad v_1 : D_1, \ldots, v_n : D_n;\\ &\textbf{eqn}\quad f(v_1, \ldots, v_n) = M;\end{aligned}$$

where $f \in \mathcal{M}^{\text{mCRL2}}$ is a mapping , $D_1, \ldots, D_n, D \in \mathcal{S}^{\text{mCRL2}}$ are sorts, $v_1, \ldots, v_n \in \mathcal{X}^{\text{mCRL2}}$ are variables of respectively the sorts $D_1, \ldots, D_n$, and M is the resulting data expression, either a constant or a function that is defined by a subset of variables $v_1, \ldots, v_n$ and (possibly other) mappings and/or constructor functions.

Example Assume that we take the sort *Tree* from our previous example. Now we define a function SUM that takes the sum over all leaves in the tree. This function can be modeled as:

$$\begin{array}{ll}
\textbf{map} & \mathit{SUM} : \mathit{Tree} \to \mathbb{N}; \\
\textbf{var} & t, u : \mathit{Tree}; \\
 & a : \mathbb{N}; \\
\textbf{eqn} & \mathit{SUM}(\mathit{leaf}(a)) = a; \\
 & \mathit{SUM}(\mathit{node}(t, u)) = \mathit{SUM}(t) + \mathit{SUM}(u);
\end{array}$$

If we take $\mathit{node}(\mathit{leaf}(1), \mathit{leaf}(2))$ and apply SUM to it, we get $\mathit{SUM}(\mathit{node}(\mathit{leaf}(1), \mathit{leaf}(2))) = \mathit{SUM}(\mathit{leaf}(1)) + \mathit{SUM}(\mathit{leaf}(2)) = 3$

The mCRL2 language allows many mathematical notations that follow their mathematical counterparts as closely as possible. Next to the basic operations on, e.g., natural numbers and Booleans, the language also allows more complex operations like set comprehensions or the union of sets. To illustrate a set comprehension of all natural numbers up to (not including) the value 5, however including the value 12, we write:

$$\{n : \mathbb{N} \mid n < 5 \vee \ n \approx 12\}$$

To unify sets S and T, assuming that both are of the same type, we simply write $S \cup T$.

c) Actions: For every process specification we assume that a set of action labels $\mathcal{A}^{\text{mCRL2}}$ is available. An *action declaration* is a subset of $\mathcal{A}^{\text{mCRL2}}$. All the actions that are specified inside a *process specification* have to be declared by an *action declaration*. *Syntactic multi-actions* are of the form:

$$\alpha ::= a(\vec{d}) \ \mid \ \alpha|\alpha,$$

where $a \in \mathcal{A}^{\text{mCRL2}}$ denotes an action label and $\vec{d}$ denotes a vector of data parameters. The syntactic multi-action $\alpha|\alpha'$ consists of the actions from both the syntactic multi-actions α and α'.

d) Processes: To describe behavior we use *Process expressions*. Process expressions are of the form:

$$\begin{array}{rl}
p ::= & \alpha \mid c{\to}p \mid \sum_{v:D} p \mid p{\cdot}p \mid p\|p \mid \\
 & \nabla_V(p) \mid X(v_1{=}d_1, \ldots, v_n{=}d_n)
\end{array}$$

Here, p denotes a process term, α denotes a syntactic multi-action, $c \in \mathbb{B}$ denotes a Boolean data-expression, v, $v_1, \ldots, v_n \in \mathcal{X}^{\text{mCRL2}}$ $(n{\geq}0)$ are variables, V denotes a set of multi-action labels, and $d_1, \ldots, d_n$ denote data expressions. For processes, $c{\to}p$ denotes the conditional if-then execution, $\sum\limits_{v:D} p$ denotes the non-deterministic choice over the domain of D by selecting a value for variable v, $p{\cdot}p$ denotes the sequential composition, and $p\|p$ denotes the parallel composition. The operator $\nabla_V(p)$ allows only the multi-actions from the set V of multi-action labels. X is a recursion variable, $X(v_1{=}d_1, \ldots, v_n{=}d_n)$ is a process reference to a *process equation*. For a *process equation* $X(v_1{:}D_1, \ldots, v_n{:}D_n) = p \in \mathit{PE}$ holds that p is a process expression, and for $(1 \leq i \leq n)$, $v_i \in \mathcal{X}^{\text{mCRL2}}$ are variables of sort $D_i \in \mathcal{S}^{\text{mCRL2}}$.

2) Semantical Concepts: The semantics for the mCRL2 language is rich [15], [30], [31]. As it is too lengthy to be described in full, we only describe the semantic notations that are relevant for the dogfooding approach. So we restrict the sorts of the input data specifications to only the Boolean sort. Hence, $\mathcal{S}^{\text{mCRL2}}$ corresponds to only the sort name $\mathbb{B}$ with the constructor functions *true* and *false*, presuming that *true* and *false* are different. We assume no mappings, i.e., $\mathcal{M}^{\text{mCRL2}} = \emptyset$.

Let $M_{\mathbb{B}}$ be the $\mathcal{D}_{\mathbb{B}}^{\text{mCRL2}}$-model where the domain of $M_{\mathbb{B}}$ is the sort $\mathbb{B}$, i.e., $\{\mathbf{true}, \mathbf{false}\}$. Then we write $\sigma{:}\mathcal{X}^{\text{mCRL2}} \to M_{\mathbb{B}}$ to describe a (partial) *valuation* function. We write $\sigma[v \mapsto w]$ for the valuation σ' with domain $\mathit{dom}(\sigma) \cup \{v\}$, such that $\sigma'(v) = w$ and $\forall_{v' \in \mathit{dom}(\sigma) \setminus \{v\}} \sigma'(v') = \sigma(v')$ hold.

The semantic interpretation function $\{[\cdot]\}^{\sigma}$ on a data expression is defined as:

- $\{[v]\}^{\sigma} = \sigma(v)$ for every variable $v \in \mathcal{X}^{\text{mCRL2}}$.
- $\{[\mathit{true}]\}^{\sigma} = \mathbf{true}$.
- $\{[\mathit{false}]\}^{\sigma} = \mathbf{false}$.

Let $a \in \mathcal{A}^{\text{mCRL2}}$, let $\vec{w} \in \vec{M_{\mathbb{B}}}$ and let α, β be syntactic multi-actions. Then the interpretation of a syntactic multi-action under σ is inductively defined by:

$$[\![a(\vec{w})]\!]^{\sigma} = a(\{[\vec{w}]\}^{\sigma}) \qquad\qquad [\![\alpha|\beta]\!] = [\![\alpha]\!]^{\sigma}|[\![\beta]\!]^{\sigma}$$

Note, that for *presentation purposes* the syntactic and the semantic action labels are both represented by $\mathcal{A}^{\text{mCRL2}}$.

Let α, β be semantic multi-actions, then the semantic multi-action equivalence relation is defined as the smallest equivalence relation that satisfies:

$$\alpha \mid \beta \sim \beta \mid \alpha \qquad\qquad (\alpha \mid \beta) \mid \gamma \sim \alpha \mid (\beta \mid \gamma)$$

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

$$\alpha_{\sim} = \{\beta :: \beta \sim \alpha\}$$

Let $a \in \mathcal{A}^{\text{mCRL2}}$ and let $\vec{w} \in \vec{M_{\mathbb{B}}}$, and let $\alpha_{\sim}$, $\beta_{\sim}$ be multi-action equivalence classes, then we define a function that combines separate equivalence classes into a new equivalence class by:

$$(a(\vec{w})_{\sim})_{\sim} = a(\vec{w})_{\sim} \qquad\qquad (\alpha_{\sim} \mid \beta_{\sim})_{\sim} = (\alpha \mid \beta)_{\sim}$$

Given a process specification, 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.

Let PS be a process specification and let σ_0 be a data valuation. The semantics for a process specification PS given $\mathcal{A}^{\text{mCRL2}}$ and σ_0 is defined by a transition system $(S, \mathit{Act}, \longrightarrow, s_0, T)$:

- The states S contain all pairs (p', σ') for process expressions p' and valuations σ'. The state where a process successfully terminates is denoted by $\checkmark$.
- The set of labels Act is the set of semantic multi-action equivalence classes.
- The transitions are inductively defined by the deduction rules in Table I. These describe the semantics for the restricted mCRL2 language. The transition relation is denoted by $(p', \sigma) \xrightarrow{m} (p'', \sigma'), (p', \sigma) \xrightarrow{m} \checkmark \in S \times \mathit{Act} \times S$.
- The initial state s_0 corresponds to (p, σ_0).
- $T \subseteq S$ is the set of *terminating states*. The set of terminating states corresponds to the states where $\checkmark$ holds.

$$(Alpha)\,\frac{}{(\alpha,\sigma)\xrightarrow{\llbracket\alpha\rrbracket^{\sigma}_{\sim}}\checkmark}$$

$$(Seq_1)\,\frac{(p,\sigma)\xrightarrow{m}\checkmark}{(p{\cdot}q,\sigma)\xrightarrow{m}(q,\sigma)} \qquad (Seq_2)\,\frac{(p,\sigma)\xrightarrow{m}(p',\sigma')}{(p{\cdot}q,\sigma)\xrightarrow{m}(p'{\cdot}q,\sigma')}$$

$$(Cond_1)\,\frac{\{\![b]\!\}^{\sigma}{=}\mathbf{true} \quad (p,\sigma)\xrightarrow{m}\checkmark}{(b{\rightarrow}p,\sigma)\xrightarrow{m}\checkmark} \qquad (Cond_2)\,\frac{\{\![b]\!\}^{\sigma}{=}\mathbf{true}, \quad (p,\sigma)\xrightarrow{m}(p',\sigma')}{(b{\rightarrow}p,\sigma)\xrightarrow{m}(p',\sigma')}$$

$$(Sum_1)\,\frac{(p,\sigma[v\mapsto w])\xrightarrow{m}\checkmark, w{\in}M_{\mathbb{B}}}{(\sum\limits_{v:\mathbb{B}}p,\sigma)\xrightarrow{m}\checkmark} \qquad (Sum_2)\,\frac{(p[v\mapsto v'],\sigma[v'\mapsto w])\xrightarrow{m}(p',\sigma'),\ w{\in}M_{\mathbb{B}}}{(\sum\limits_{v:\mathbb{B}}p,\sigma)\xrightarrow{m}(p',\sigma')}$$

where v' is a fresh variable of sort $\mathbb{B}$, i.e., $v' \notin dom(\sigma)$

$$(Par_1)\,\frac{(p,\sigma)\xrightarrow{m}\checkmark}{(p\|q,\sigma)\xrightarrow{m}(q,\sigma)} \qquad (Par_2)\,\frac{(p,\sigma)\xrightarrow{m}(p',\sigma')}{(p\|q,\sigma)\xrightarrow{m}(p'\|q,\sigma')}$$

$$(Par_3)\,\frac{(q,\sigma)\xrightarrow{m}\checkmark}{(p\|q,\sigma)\xrightarrow{m}(p,\sigma)} \qquad (Par_4)\,\frac{(q,\sigma)\xrightarrow{m}(q',\sigma')}{(p\|q,\sigma)\xrightarrow{m}(p\|q',\sigma')}$$

$$(Par_5)\,\frac{(p,\sigma)\xrightarrow{m}\checkmark, (q,\sigma)\xrightarrow{n}\checkmark}{(p\|q,\sigma)\xrightarrow{(m|n)_{\sim}}\checkmark} \qquad (Par_6)\,\frac{(p,\sigma)\xrightarrow{m}(p',\sigma'), (q,\sigma)\xrightarrow{n}\checkmark}{(p\|q,\sigma)\xrightarrow{(m|n)_{\sim}}(p',\sigma')}$$

$$(Par_7)\,\frac{(p,\sigma)\xrightarrow{m}\checkmark, (q,\sigma)\xrightarrow{n}(q',\sigma')}{(p\|q,\sigma)\xrightarrow{(m|n)_{\sim}}(q',\sigma')} \qquad (Par_8)\,\frac{(p,\sigma)\xrightarrow{m}(p',\sigma'), (q,\sigma)\xrightarrow{n}(q',\sigma'')}{(p\|q,\sigma)\xrightarrow{(m|n)_{\sim}}(p'\|q',\sigma'{\cup}\sigma'')}$$

$$(Allow_1)\,\frac{(p,\sigma)\xrightarrow{m}\checkmark, m\cap V\neq\emptyset}{(\nabla_V(p),\sigma)\xrightarrow{m}\checkmark} \qquad (Allow_2)\,\frac{(p,\sigma)\xrightarrow{m}(p',\sigma'), m\cap V\neq\emptyset}{(\nabla_V(p),\sigma)\xrightarrow{m}(\nabla_V(p'),\sigma')}$$

$$(Def_1)\,\frac{(q,\sigma[\vec{v}\mapsto\{\![\vec{d}]\!\}^{\sigma}])\xrightarrow{m}\checkmark}{(X(\vec{v}{=}\vec{d}),\sigma)\xrightarrow{m}\checkmark} \qquad (Def_2)\,\frac{(q[\vec{v}\mapsto\vec{v'}],\sigma[\vec{v'}\mapsto\{\![\vec{d}]\!\}^{\sigma}])\xrightarrow{m}(q',\sigma')}{(X(\vec{v}{=}\vec{d}),\sigma)\xrightarrow{m}(q',\sigma')}$$

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

TABLE I: SOS deduction rules for the restricted mCRL2 language

III. Modeling the semantics of the mCRL2 language in an mCRL2 specification

An mCRL2 specification consists of two parts, namely a data specification part and a process specification part. Computations that are performed by the data specification are executed by a higher order rewriter. The behavior associated with process operators is manually implemented in code.

By dogfooding the semantics of the mCRL2 language, we lift the specification of the operators from code to the higher order rewriter. This is accomplished by modeling process terms as data expressions (i.e., the signature of the process terms is described by a structured sort), by modeling deduction rules as mappings with corresponding rewrite rules (i.e., deduction rules are represented by set comprehensions), and by expressing transition relations as process expressions. Data (sorts, mappings, constructors, . . .) can be modeled by the data itself. Syntactic and semantic concepts that are expressed in their lifted form are referred to as notions in the *meta notation*.

Transition relations for the deduction rules are modeled by an mCRL2 specification of a particular form, namely a Linear Process Specification (LPS). An LPS can be viewed as a symbolic representation for a (possible infinite) labeled transition system. A formal definition of an LPS and its components can be found in [4], [12]. An LPS consists of an mCRL2 data specification, a Linear Process Equation (LPE), and an initialization. The (simplified) LPE that we use in this paper is:

$$X(\vec{d}{:}\vec{D}) \quad = \quad \sum_{e:E} c(\vec{d},e){\rightarrow}a(f(\vec{d},e)){\cdot}X(g(\vec{d},e))$$

where e and E respectively denote a variable name and a sort expression, $c(\vec{d},e)$ is a data expression of sort $\mathbb{B}$ that serves as a guard to allow actions, $a(f(\vec{d},e)) : \mathcal{A}^{\mathrm{mCRL2}} \times \vec{D}$ is an action with data parameters and $g(\vec{d},e)$ is a term of sort $\vec{D}$ that denotes the next state.

To capture mCRL2's syntax and semantics in an mCRL2 specification, we take design and modeling decisions such that the resulting models are proper mCRL2 specifications and they can be subjected to exhaustive simulation. In the context of mCRL2, exhaustive simulation corresponds to the exploration of all possible behavior. So if one models a system with infinite branching or specifies a non-terminating process, the exhaustive simulation runs infinitely and will be only limited by the available time and memory a machine or user has.

This section explains the modeling decisions for (i) the modeling of the signature of a process term, (ii) the representation of data expressions, (iii) the computation of syntactic multi-actions into semantic multi-action equivalence classes, (iv) the representation of the transition relations and (v) the SOS deduction rules.

A. Process Terms

The signature of a process term is translated into a structured sort, represented by $\mathcal{P}$. Process terms in the mCRL2 language are defined by a single sort. For every mCRL2 process term, we introduce a constructor that carries the textual characterization of the function symbol. Arguments of these terms can be represented by different sorts. They describe, e.g., the condition in the conditional choice operator, the actions that are allowed to execute within the scope of an allow operator, or a sub-process term. To model the sorts for syntactic multi-actions, Boolean data expressions, variables, sets of syntactic multi-actions, process labels, and lists of process parameter updates, we respectively introduce and use the sorts $List(Act_{\Xi})$[1], $\mathcal{E}_{\mathbb{B}}, \mathcal{V}, Set(Bag(Act_{Lab})), \mathcal{X}$ and $List(\mathcal{Q})$. The arguments of the terms are modeled by their mathematical counterparts.

To access the operands of a term, we add projection functions. Here, π_n denotes the n^{th} operand of a process term of sort $\mathcal{P}$. To access other operands of other sorts, like c in $c \to p$, we add projection functions with specially selected names, e.g., π_c. For the conciseness of the model, the $\checkmark$ predicate is modeled via the $\checkmark_{\mathbf{p}}$ constructor. Based on these decisions, we model the signature of the process expressions as:

$$
\begin{aligned}
\textbf{sort}\quad \mathcal{P} = \mathbf{struct}\quad & \\
& \checkmark_{\mathbf{p}}?is_{\checkmark} \\
& \mid Alpha(\pi_{multiaction}{:}List(Act_{\Xi}))?is_{\alpha} \\
& \mid Seq(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{Seq} \\
& \mid Cond(\pi_c{:}\mathcal{E}_{\mathbb{B}}, \pi_1{:}\mathcal{P})?is_{Cond} \\
& \mid Sum(\pi_d{:}\mathcal{V}, \pi_1{:}\mathcal{P})?is_{Sum} \\
& \mid Par(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{Par} \\
& \mid Allow(\pi_V{:}Set(Bag(Act_{Lab})), \pi_1{:}\mathcal{P})?is_{Allow} \\
& \mid Def(\pi_{PE_{lab}}{:}\mathcal{X}, \pi_{ProcParAsss}{:}List(\mathcal{Q}))?is_{Def};
\end{aligned}
$$

B. Data

Data expressions are either values or variables. A data expression is modeled by a structured sort. Because we restrict the language to the Boolean sort, we only have to represent this sort in the meta notation. So, the sort for the Boolean data expression is modeled by $\mathcal{E}_{\mathbb{B}}$ (condition's guard):

$$
\begin{aligned}
&\textbf{sort}\quad \mathcal{V} = \mathbf{struct}\ v_1 \mid \ldots \mid v_n; \\
&\textbf{sort}\quad \mathcal{E}_{\mathbb{B}} = \mathbf{struct}\ \mathcal{E}_{\mathcal{V}}(dvr{:}\mathcal{V})?is^{\mathcal{V}}_{\mathcal{E}} \mid \mathcal{E}_{\Lambda}(dvl{:}\mathbb{B})?is^{\Lambda}_{\mathcal{E}}
\end{aligned}
$$

where sort $\mathcal{V}$ models the set of variables, $\mathcal{E}_{\mathcal{V}}$ models a data expression being a variable and $\mathcal{E}_{\Lambda}$ models a data expression being a value. The signature of valuation σ is modeled via sort $\mathcal{S}$. The implementation of $\mathcal{S}$ can be found in [30], [31].

C. Multi-actions

The mCRL2 language has two kinds of multi-actions, namely the syntactic actions and the semantic actions. The syntactic actions are the ones specified in an mCRL2 specification. Semantic actions are observed during the execution of a process. For both kinds we introduce sorts such that we can represent them. So, we represent *syntactic action* by the sort:

$$
\begin{aligned}
\textbf{sort}\quad & Act_{\Xi} = \mathbf{struct} \\
& \quad Act(actionlabel{:}Act_{Lab}, args{:}List(\mathcal{E}_{\mathbb{B}}));
\end{aligned}
$$

A *syntactic multi-action* consists of a list of syntactic actions, i.e., $List(Act_{\Xi})$. A *semantic action* is represented by the sort:

$$
\begin{aligned}
\textbf{sort}\quad & Act_{\Sigma} = \mathbf{struct} \\
& \quad ActSem(actionlabel{:}Act_{Lab}, args{:}List(M_{\mathbb{B}}));
\end{aligned}
$$

A *semantic multi-action equivalence class* consists of an (ordered) list of semantic actions, i.e., $List(Act_{\Sigma})$. To transform syntactic multi-actions into semantic multi-action equivalence classes, we provide an interpretation function $sem^{List}_{Act_{\Xi}}$: $List(Act_{\Xi}) \times \mathcal{S} \to List(Act_{\Sigma})$ that models $[\![\cdot]\!]^{\sigma}$. Its full definition is described in [30], [31].

D. Transition Relation Representation

Deduction rules describe transition relations, that are presented as $(p, \sigma) \xrightarrow{a} (p', \sigma')$. Such a transition relation is composed from (i) a modeled state, (ii) the signature of a modeled transition, and (iii) a transition with the corresponding updated state (p', σ'). All of these components are modeled accordingly.

To model (i), we first observe that a state of an mCRL2 transition relation consists of a process term p and a data valuation σ. The meta notation introduces process X with a process parameter that stores the meta notation of process term p and the meta notation for valuation σ. Hence, process X describes the transition relation for a state (p, σ).

Data expressions cannot be directly used as transitions. Thus the semantic multi-action equivalence class is modeled as a data parameter with a dedicated action label $\mathbb{A}$. The action label denotes the kind of relation, i.e. the transition relation. To model (ii), the signature of a transition for $\xrightarrow{a}$, we declare (and use) the following mCRL2 action declaration. Note that semantic multi-actions are expressed through notions that are also used for the syntactic actions.

$$\textbf{act}\quad \mathbb{A}{:}List(Act_{\Sigma});$$

To model (iii) we define a function that, given a state, computes the transitions and the corresponding state updates for a transition relation. Hence, we introduce sort $\mathcal{R}_{at}$ to represent a solution that consists of a multi-action, a process term and a data valuation:

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

Here, at is the constructor function for a solution, argument π_{α} denotes the multi-action equivalence class, argument $\pi_{p'}$ denotes the updated process term and argument $\pi_{\sigma'}$ denotes the updated valuation. Then for every deduction rule ($d \in \mathcal{D}_{\mathrm{SOS}}$), we introduce $R_d{:}\mathcal{P} \times \mathcal{S} \to Set(\mathcal{R}_{at})$ that provides the modeled deduction rule for d. The deduction rule itself is encoded as a set comprehension. The set of solutions is computed by $R{:}\mathcal{P} \times \mathcal{S} \to Set(\mathcal{R}_{at})$, which unifies the solution sets of the individual deduction rules.

[1] For convenience and presentation purposes we present an multi-action as a list of actions.

$$\begin{array}{ll}
\textbf{map} & R, R_{\mathit{Alpha}}, R_{\mathit{Alt_1}}, \ldots, R_{\mathit{Def}_1}, R_{\mathit{Def}_2}{:}\mathcal{P} \times \mathcal{S} \to \mathit{Set}(\mathcal{R}_{at}); \\
\textbf{var} & p{:}\mathcal{P}; \\
 & s{:}\mathcal{S}; \\
\textbf{eqn} & R(p,s) = \; R_{\mathit{Alpha}}(p,s) \cup R_{\mathit{Alt_1}}(p,s) \cup \ldots \cup \\
 & \qquad\qquad R_{\mathit{Def}_1}(p,s) \cup R_{\mathit{Def}_2}(p,s);
\end{array}$$

Since only the applicable functions return a non-empty set, only the actual solutions remain. So we model the transition relation with help of the LPE as:

$$\begin{array}{ll}
\textbf{proc} & X(p{:}\mathcal{P}, s{:}\mathcal{S}) = \\
 & \sum_{r:\mathcal{R}_{at}} (r \in R(p,s)) \to \mathbb{A}(\pi_\alpha(r)) \cdot X(\pi_{p'}(r), \pi_{\sigma'}(r))
\end{array}$$

If (p_0, σ_0) denotes the initial state for the mCRL2 model in the meta notation, then the LPS is initialized by:

$$\textbf{init} \quad X(p_0, \sigma_0);$$

E. Deduction Rules

The original deduction rules in [15] describe the semantics for dense domains, e.g., the idle time transition and the (possible) enumeration over sort $\mathbb{R}$. To avoid computational problems during the analysis we prohibit the use of dense sorts in the translation. Note that incorporating these dense domains into the translation, does not hinder the approach, as shown in [30], [31]. In this paper we present a limited set of deduction rules that are relevant for the discovered issues.

To illustrate the modeling of deduction rules, we describe how the deduction rules for multi-actions (*Alpha*), sequential composition (*Seq1*, *Seq2*) and the conditional choice (*Cond1*, *Cond2*) are modeled. The complete set of modeled deduction rules can be found in [30], [31].

a) Multi-actions: Let $a_1, \ldots, a_n \in Act_\Xi$, then a syntactic multi-action is written in the meta notation as:

$$\mathit{Alpha}([a_1, \ldots, a_n])$$

The function R_{Alpha} computes the solutions that correspond to the deduction rule of a multi-action by means of a set comprehension. We allow a solution r in the set iff (i) input term p is an action process term ($is_\alpha(p)$), (ii) the semantic multi-action corresponds to the syntactic multi-action, evaluated under the data valuation ($\pi_\alpha(r) \approx sem^{List}_{Act_\Xi}(\pi_{multiaction}(p), s)$), (iii) the process term denotes a successful termination relation ($is_{\checkmark}(\pi_{p'}(r))$), and (iv) the data valuation remains unchanged ($\pi_{\sigma'}(r) \approx s$). Hence, we express R_{Alpha}:

$$\begin{array}{ll}
\textbf{eqn} & R_{\mathit{Alpha}}(p,s) = \\
 & \quad \mathit{if}(is_\alpha(p), \\
 & \qquad \{r{:}\mathcal{R}_{at} \mid \pi_\alpha(r) \approx sem^{List}_{Act_\Xi}(\pi_{multiaction}(p)), s) \\
 & \qquad\quad \wedge\ is_{\checkmark}(\pi_{p'}(r)) \ \wedge\ \pi_{\sigma'}(r) \approx s\}, \emptyset);
\end{array}$$

b) Sequential Composition: The sequential composition operator is denoted by $p \cdot q$. The meta notation expresses the sequential composition as:

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

where $p, q \in \mathcal{P}$ are process terms in meta notation.

Deduction rule Seq_1 in Table I expresses the successful termination of p. Deduction rule Seq_2 in Table I expresses the continuation as $p' \cdot q$ after p has performed an action and does not successfully terminate. The solutions that belong to deduction rule Seq_1 and Seq_2 are computed by $R_{\mathit{Seq_1}}$ and $R_{\mathit{Seq_2}}$, respectively. Note that $R_{\mathit{Seq_2}}$ demands that the signature of the resulting process term is again a sequential composition ($is_{\mathit{Seq}}(\pi_{p'}(r))$). Hence, we specify the following two data equations:

$$\begin{array}{ll}
\textbf{eqn} & R_{\mathit{Seq_1}}(p,s) = \mathit{if}(is_{\mathit{Seq}}(p), \\
 & \quad \{r{:}\mathcal{R}_{at} \mid at(\pi_\alpha(r), \checkmark_{\mathbf{p}}, \pi_{\sigma'}(r)) \in R(\pi_1(p), s) \\
 & \qquad \wedge\ \pi_{p'}(r) \approx \pi_2(p) \wedge\ \pi_{\sigma'}(r) \approx s\}, \emptyset); \\
 & R_{\mathit{Seq_2}}(p,s) = \mathit{if}(is_{\mathit{Seq}}(p), \\
 & \quad \{r{:}\mathcal{R}_{at} \mid is_{\mathit{Seq}}(\pi_{p'}(r)) \\
 & \qquad \wedge\ at(\pi_\alpha(r), \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p), s) \\
 & \qquad\quad \wedge\ \pi_2(\pi_{p'}(r)) \approx \pi_2(p) \wedge\ \neg is_{\checkmark}(\pi_1(\pi_{p'}(r)))\}, \emptyset);
\end{array}$$

c) Conditional Choice: The conditional choice $c \to p$ allows the execution of behavior with respect to the evaluated condition. The operator only executes the behavior of process p if the data expression c evaluates to true. The operator is modeled by:

$$\mathit{Cond_1}(c, p)$$

Here $c \in \mathcal{E}$ is a data expression and $p \in \mathcal{P}$ is a process term.

The operand contains a syntactic Boolean data expression that requires a semantic interpretation. With the help of function $sem_{\mathcal{E}}$, we compute the semantic value for $\pi_c(p)$ (the projection function C applied to process term p) under the data valuation $s \in \mathcal{S}$. The function $\mathbb{B}_\downarrow$ evaluates whether the semantic interpretation, corresponds to the logical value true. Note, that the condition is written inside the guard of the *if*-statement, instead of the body of the set comprehension, to circumvent infinite rewrite sequences executed by tools.

The behavior for the operator $c \to p$ corresponds to the deduction rules *Cond*$_1$ and *Cond*$_2$ in Table I. for which we provide two data equations:

$$\begin{array}{ll}
\textbf{eqn} & R_{\mathit{Cond_1}}(p,s) = \mathit{if}(is_{\mathit{Cond_1}}(p) \wedge\ \mathbb{B}_\downarrow(sem_{\mathcal{E}}(\pi_c(p), s)), \\
 & \quad \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_1(p), s) \\
 & \qquad \wedge is_{\checkmark}(\pi_{p'}(r)) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset); \\
 & R_{\mathit{Cond_2}}(p,s) = \mathit{if}(is_{\mathit{Cond_1}}(p) \wedge\ \mathbb{B}_\downarrow(sem_{\mathcal{E}}(\pi_c(p), s)), \\
 & \quad \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_1(p), s) \wedge\ \neg is_{\checkmark}(\pi_{p'}(r))\}, \emptyset);
\end{array}$$

IV. Results

This section discusses some of the results obtained from our semantic dogfooding approach. We first reshape the deduction rules into a format that can be digested by our framework. Secondly, we outline the constructed models that have resulted from the transformation. Subsequently, we present a couple of illustrative examples that have been processed by both the framework and the mCRL2 toolset. Finally, we present the correspondence issues that have been discovered using the dogfooding approach. Here the defined semantics deviates from the implemented semantics, i.e., the defined semantics was stated incorrectly.

A. Conformance

Although the semantics of the language is formally defined, it still contained ambiguous behavior. To illustrate, the original specification states: “let there be a fresh variable d'”. Does this mean that d' is an unique fresh variable or do infinitely many fresh variables correspond to d'? We assumed the first since it provides a model that can be (exhaustively) simulated. When assuming the second, every introduction of a fresh variable would correspond to infinite branching behavior. Also, we have adapted the definition of a semantic multi-action. In the original semantics, the multi-action is a collection of semantic actions. It assumed that such a semantic multi-action is an equivalence class. However, this was never explicitly stated. Since we need to apply functions on these semantic multi-actions, we have explicitly stated that there indeed exists such an equivalence as included in Section II-B. We stress that these differences are related to the implementation of the mathematics and not to the mathematics themselves.

When considering the deduction rules in Table I we observe some peculiarities. First, deduction rule Def_2 introduces fresh variables with respect to σ. As already explained this dictates infinite branching. However if we observe the behavior of the mCRL2's (exhaustive) simulation tools, they suggest that the rule should only specify one $\vec{v'}$, for which all variables are disjoint from $dom(\sigma)$. For our convenience, we assume $fresh(dom(\sigma), n) = \vec{v}$ that generates a list of variable of length n provided $dom(\sigma)$, such that $dom(\sigma) \cap \vec{v} = \emptyset$ holds. Hence, we redefine deduction rule Def_2 as:

$$(Def'_2)\frac{\begin{array}{c}(q[\vec{v} \mapsto \vec{v'}], \sigma[\vec{v'} \mapsto \{\![\vec{d}]\!\}^{\sigma}]) \xrightarrow{m} (q', \sigma'),\\ fresh(dom(\sigma), |\vec{v}|) = \vec{v'}\end{array}}{(X(\vec{v} = \vec{d}), \sigma) \xrightarrow{m} (q', \sigma')}$$

where $X(\vec{v}{:}\vec{\mathbb{B}}) = q \in PE$.

Second, the deduction rule Par_8 silently assumes that the values of the non freshly generated variables remain the same. To make the assumption explicitly we introduce the notation $\sigma' =_{dom(\sigma)} \sigma''$:

$$\sigma' =_{dom(\sigma)} \sigma'' \equiv \forall_{v \in dom(\sigma)} \sigma'(v) = \sigma''(v)$$

Since deduction rule Def'_2 now generates specific fresh variables, we need to rename all freshly generated variables from the second premise that were also introduced by the first premise. For this we use the same mechanism as for generating fresh variables. Let $v_{\vec{dup}}$ be a list of variables, then we define deduction rule Par_8 as:

$$(Par'_8)\frac{\begin{array}{c}(p, \sigma) \xrightarrow{m} (p', \sigma'),\ (q, \sigma) \xrightarrow{n} (q', \sigma''),\\ \sigma' =_{dom(\sigma)} \sigma'',\\ fresh(dom(\sigma') \cup dom(\sigma), |v_{\vec{dup}}|) = \vec{v'}\end{array}}{(p \| q, \sigma) \overset{(m|n)_{\sim}}{\longrightarrow} (p' \| (q'[v_{\vec{dup}} \mapsto \vec{v'}]), \sigma''')}$$

such that for σ''' holds:

$$\begin{array}{ll} & \forall_{v \in dom(\sigma')} \sigma'''(v) = \sigma'(v)\\ \wedge & \forall_{v \in dom(\sigma'') \setminus dom(\sigma')} \sigma'''(v) = \sigma''(v)\\ \wedge & \forall_{1 \leq n \leq |v_{\vec{dup}}|} \sigma'''((\vec{v'})_n) = \sigma''((v_{\vec{dup}})_n)\\ \wedge & dom(\sigma''') = dom(\sigma') \cup dom(\sigma'') \cup \{\vec{v'}\}\end{array}$$

where $\{v_{\vec{dup}}\}$ is the set interpretation of $v_{\vec{dup}}$ for which holds $\{v_{\vec{dup}}\} = (dom(\sigma'') \cap dom(\sigma')) \setminus dom(\sigma)$, and $(\vec{v})_i$ denotes the i^{th} element of $\vec{v}$.

B. Implementing the mCRL2 semantics

The complete untimed mCRL2 language is captured in (roughly) 1000 lines of mCRL2 code, by which we exhaustively simulated over 100 models. The signature of the process terms and sorts required for the action transitions are encoded in 30 lines of code each. To express and model the transition relation we require two lines of code; one line to declare the action relation and another one for conducting the transitions. Then 550 lines of code have been used to encode the 43 deduction rules from the original specification. A 150 lines of code have been used to model the functions that operate on transitions. An instance of a model is captured by a single line of code. The remaining lines of code are used to capture the auxiliary mathematical concepts, like substituting variables in a process term, encoding the order on a list of semantic actions such that a equivalence class has one representative, concatenating two lists of semantic actions, etc. . . .

The latest stable release of the mCRL2 toolset (Release February 2012) has been used to implement the framework and validate the behavior. For validation we compared the behavior of different models by taking two routes for each of them. The first route takes an mCRL2 specification, linearizes the model using the tool `mcrl22lps`, after which we then generate the corresponding state space using the tool `lps2lts`. In the second route we take the same specification, provide it to our framework, use the result as the input for the tool `txt2lps`, and subsequently generate the state space for it. The results are afterwards compared for behavioral differences.

C. Illustrative dogfooding examples

From these 100 models we have selected three examples, for which their corresponding behavior is depicted in the Figures 2, 3 and 4. Every example corresponds to an LTS for a model in the meta notation and is generated using the mCRL2 toolset (Release February 2012). In an LTS every arrow depicts a transition relation. A node depicts a state. The initial state is indicated with a doubly lined node. The characteristics of the models that belong to the state spaces are provided in the corresponding captions.

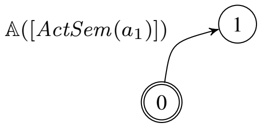

Fig. 2: $\sum_{v_1:\mathbb{B}} v_1 \rightarrow a_1$

Guarded action Figure 2 depicts the state space for the mCRL2 process term "$\sum_{v_1:\mathbb{B}} v_1 \rightarrow a_1$". The corresponding meta notation for this term is:

$$Sum(\mathbb{B}_{\mathcal{V}}(v_1), Alpha([Act(a_1)]))$$

where the data valuation is initially empty. When we compare the models from both routes we see that their behavior is identical.

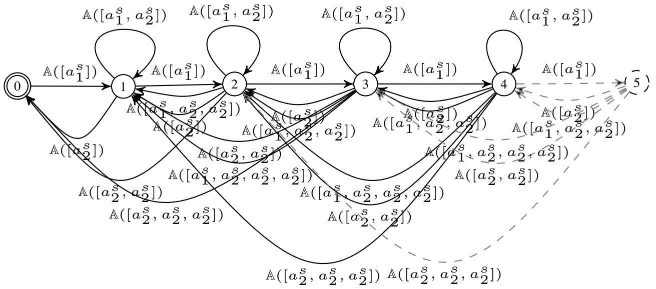
0
1
2
3
4
5
𝔸([a_1^s])
𝔸([a_1^s, a_2^s])
𝔸([a_2^s])
𝔸([a_2^s, a_2^s])
𝔸([a_2^s, a_2^s, a_2^s])
𝔸([a_1^s, a_2^s, a_2^s])
𝔸([a_1^s, a_2^s, a_2^s, a_2^s])

Fig. 3: $P_1 = a_1{\cdot}(a_2 \| P_1)$

Branching parallelism Figure 3 illustrates the behavior for the recursive process definition "$P_1 = a_1{\cdot}(a_2\|P_1)$". Every time the recursion is unfolded, the process allows more concurrent behavior. The corresponding meta notation for this term is:

eqn $PES(P_1) =$
$$Seq(Alpha([Act(a_1)]), \\ Par(Alpha([Act(a_2)]), Def(P_1, [])));$$

The initialization is provided through the mCRL2 process term "P_1" or, in meta notation, "$Def(P_1, [])$". It is a valid mCRL2 specification but since the process introduces more concurrency at every recursion, it can not be linearized by the toolset. That is, to linearize an mCRL2 specification it must be either in the pCRL format [28] or comply to the LPS format. Since the specification is none of the above, it cannot be used as input. Our framework produces models that conform to the LPS format, so the resulting models can be directly processed by the same mCRL2 toolset. This means that, natively, we can not input this mCRL2 process term and generate the corresponding state space. If we use the technique described in [32], and we first write the model in the meta notation, we *can* generate a part of the state space. Because the model unfolds infinitely we observe exponential (unbounded) growth in computation time (and memory usage) to calculate the transitions. Therefore we only show the corresponding transitions for the first five states[2].

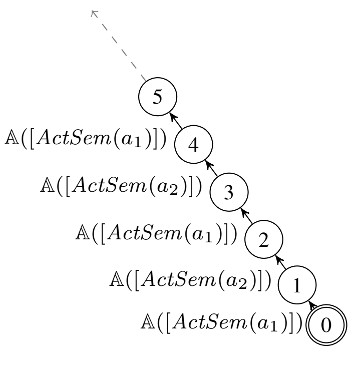

Fig. 4: $P_2(v_1{:}\mathbb{B}) = a_1{\cdot}a_2{\cdot}P_2(v_1{=}true)$

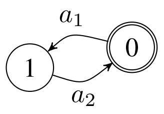

Fig. 5: Recursion in mCRL2

Infinite trace Figure 4 shows the behavior for the recursive process:
$$P_2(v_1{:}\mathbb{B}) = a_1{\cdot}a_2{\cdot}P_2(v_1{=}true)$$
This model specifies the alternating execution of the actions a_1 and a_2, after which the value for variable v_1 is set to $true$. Figure 5 depicts the state space for the native mCRL2 specification. Now, in meta notation we write:

eqn $PES(P_2) =$
$$Seq(Seq(Alpha([Act(a_1)]), Alpha([Act(a_2)])), \\ Def(P_2, [ProcParAss(\mathbb{B}_{\mathcal{V}}(v_1), \mathcal{E}_{\mathcal{V}}(true))]));$$

For the state space of the meta notation model, we witness a non-terminating path (illustrated by a dotted line). If we compare the state spaces, we the behavior between the two models deviates. Investigation shows that the difference is caused by the generation and subsequently renaming of the fresh variables in a process definition. The mCRL2 semantics states that every unfold of a process definition introduces a (fresh) variable. This means we will never visit a state for which both the process term and data valuation have previously encountered values.

Finally, the authors have proven, in previous work [34], [30], the isomorphic relation for the behavior executed by the specified model and the translated model for deduction rules that comply to the De Simone-format. Although the deduction rules of the mCRL2 language, the ones stated here, are in a more elaborate format, we believe that the same isomorphic relation holds between the specified model and a translated model. Under this assumption we conducted experiments with the tools to support this claim. Here, we saw that the assumed relation was stronger than the relation provided by tools, i.e., the tools preserve a strong-bisimulation equivalence relation on the model's behavior. Hence, native and counterpart mCRL2 models may depict different state spaces (e.g., as we have seen with the model $P_2(v_1{:}\mathbb{B}) = a_1{\cdot}a_2{\cdot}P_2(v_1{=}true)$, Figure 4).

D. Discovered correspondence issues

Although the semantics has been considered finalized since September 2009, the approach still discovered errors. These errors include simple oversights in the documentation such as duplicate deduction rules (e.g., for $\lfloor\!\lfloor$[3]) and a missing deduction rule for the parallel operator. Also, seven auxiliary operators were missing from the deduction rules as the one for ($Allow_2$) illustrated in Table II.

More interestingly, we also discovered that the implementation deviates from its intended semantics in at least two cases. The first deviation has already been discussed in Section IV-A,

[2] For illustration purposes we added a 6^{th} state, and renamed the transition labels $\mathbb{A}([ActSem(a_1)])$ and $\mathbb{A}([ActSem(a_2)])$ to a_1^s and a_2^s respectively

[3] The details of the operator can be found in [30], [31]

Missing operator	Added missing operator
$\dfrac{(p,\sigma) \xrightarrow{m} (p',\sigma'), m \cap V \neq \emptyset}{(\nabla_V(p),\sigma) \xrightarrow{m} (p',\sigma')}$	$\dfrac{(p,\sigma) \xrightarrow{m} (p',\sigma'), m \cap V \neq \emptyset}{(\nabla_V(p),\sigma) \xrightarrow{m} (\nabla_V(p'),\sigma')}$

TABLE II: Example of a missing operator in a deduction rule

where every iteration introduces fresh variables. The second semantic deviation is illustrated by the following example. The original specification assumed the following deduction rule:

$$(Sum_2)\frac{(p,\sigma[v \mapsto w]) \xrightarrow{m} (p',\sigma'),\ w{\in}M_{\mathbb{B}}}{(\sum\limits_{v:\mathbb{B}} p,\sigma) \xrightarrow{m} (p',\sigma')}$$

Now consider the following mCRL2 process:

proc $P = \sum\limits_{d:\mathbb{B}} a{\cdot}d \rightarrow b;$
init $Q = P \| P;$

The process P selects a Boolean value and assigns it to the variable d after which it performs the actions a and then b, if d was assigned $true$. Now, let Q be $P\|P$. If left P sets $d \approx true$ and performs the action a, and then the right P sets $d \approx false$ and performs the action a, the process comes into a deadlock state. The sum operator does not introduce a fresh variable resulting in a global rather than a local value change. The deviation has been resolved by redefining deduction rule Sum_2 from the original specification to following deduction rule[4]:

$$(Sum_2)\frac{(p[v \mapsto v'],\sigma[v' \mapsto w]) \xrightarrow{m} (p',\sigma'),\ w{\in}M_{\mathbb{B}}}{(\sum\limits_{v:\mathbb{B}} p,\sigma) \xrightarrow{m} (p',\sigma')}$$

V. Related Work

Dogfooding is applied in (ordinary) software development for developing new, or extending existing (software) products. Examples can be found in e.g., compilers [36], where dogfooding (or in this case bootstrapping) is applied to construct them. Other examples include the Eclipse framework [16] that is used to develop plug-ins for Eclipse. Wolfram Research [22] states that (parts) of their web sites, applications, documentations, and test and build processes are driven by the *Mathematica* Language. Within academia we also see examples of people eating their own dogfood. For example, almost all interactive theorem provers, e.g., Isabelle [8], Coq [10], PVS [3], ACL2 [2], perform a form of dogfooding when using their own tools. Dogfooding as it is performed in this paper, i.e., where we have a model checker rewrite and interpret the formal syntax and semantics of its own language, is not that common. In fact, we believe that our semantic dogfooding approach is unique and the first of its kind.

To the best of our knowledge, Maude [11] would be the only candidate to directly express the SOS of the mCRL2 language. Maude is a high-level language and high-performance system supporting both equational and rewriting logic. To illustrate, in [7] a translation from Modular SOS (MSOS) [23], [24] to the Maude rewriting logic is outlined and proven correct. In [6] GSOS/OSOS rules are modeled in Maude allowing them to execute Ordered-SOS specifications. The work of [18] shows the implementation of Eden (the parallel extension of the functional language Haskell [19]) in Maude. More recent work [29] presents the implementation of the semantics for the πCRWL calculus and the formalization of AADL in [25]. In all, we believe that it is possible to implement the syntax and semantics of the mCRL2 language in Maude. However, we would have to use *separate* toolsets meaning we would not have discovered the implementation issues where the intended formal semantics and implemented semantics do not correspond. That is, with our approach we are not only able to validate the semantics but also its implementation.

VI. Conclusions and Future Work

In this paper we effectively demonstrate that our initial framework is extensible and is mature enough to capture the SOS semantics of a rich language like mCRL2. Our semantic dogfooding approach shows that the mCRL2 toolset can actually eat *and* exhaustively simulate the Structural Operational Semantics (with a given model) of its own language. To illustrate the feasibility and applicability, we have included a number of illustrative examples with their generated state spaces.

The approach revealed, that despite the formal characterization of mCRL2, a number of (subtle) differences between the specified formal semantics, intended formal semantics and the implemented semantics. For that we have modeled 43 deduction rules in an mCRL2 specification that is slightly over a 1000 lines of code. The differences have been studied by using 102 different models, specially selected for the experiments. These mismatches range from simple oversights to (minor) implementation issues and semantic errors; we have found 7 simple oversights in the documentation, discovered 2 duplicate deduction rules, one missing deduction rule, and two deduction rules for which the implemented semantics deviated from the specified semantics. Moreover, by using the approach we were also able to specify (legal) behavior that was not permitted by tool limitations.

In the future, we would like to dogfood the timed fragment of the mCRL2 language. Performing a direct interpretation of the dense time domain would pose all kind of new challenges, that need to be resolved first before one can do (exhaustive) simulation. An option worthwhile to consider is to extend the framework with partitioning rules, that partition and discretize the dense time domain.

In conclusion, we believe that our approach is definitely applicable to other formal languages and possibly even other toolsets. To use and benefit from the semantic dogfooding approach, the formal language and toolset have to support the definition and computation of set comprehensions, have the ability to deal with quantifiers and support a mechanism to systematically perform transitions (i.e. can implement an LPS). Given the number of discovered issues, we encourage others to conduct experiments similar to ours.

[4] This is the same deduction rules as the one mentioned in Table I

References

[1] The mCRL2 toolset. http://www.mcrl2.org/. Visited: September 25, 2012.

[2] ACL2 Theorems About Commercial Microprocessors. In M.K. Srivas and A.J. Camilleri, editors, *FMCAD*, volume 1166 of *Lecture Notes in Computer Science*, pages 275–293. Springer, 1996.

[3] C.H. Applebaum and J.G. Williams. PVS - design for a practical verification system. In *ACM Conference on Computer Science*, pages 58–68. ACM, 1984.

[4] 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.

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

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

[7] C. de O. Braga, E.H. Haeusler, J. Meseguer, and P.D. Mosses. Mapping Modular SOS to Rewriting Logic. In *LOPSTR*, pages 262–277, 2002.

[8] L.C. C. Paulson. Isabelle: The Next 700 Theorem Provers. *CoRR*, cs.LO/9301106, 1993.

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

[10] T. Coquand and G.P. Huet. Constructions: A Higher Order Proof System for Mechanizing Mathematics. In B. Buchberger, editor, *European Conference on Computer Algebra (1)*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer, 1985.

[11] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The maude ltl model checker. *Electr. Notes Theor. Comput. Sci.*, 71:162–187, 2002.

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

[13] J.F. Groote. Transition system specifications with negative premises (ext. abstract). In *CONCUR*, volume 458 of *LNCS*, pages 332–341. Springer, 1990.

[14] J.F. Groote, A.J.H. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. The Formal Specification Language mCRL2. In *MMOSS*, number 06351 in Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, 2007.

[15] J.F. Groote and M.R. Mousavi. *Modelling and Analysis of Communicating Systems (Draft)*. http://www.win.tue.nl/~jfg/educ/2IW26/herfst2011/mcrl2-book-printed-version.pdf. Visited: September 25, 2012.

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

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

[18] 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.

[19] P. Hudak, J. Hughes, S.L. Jones Peyton, and P. Wadler. A history of Haskell: being lazy with class. In Barbara G. Ryder and Brent Hailpern, editors, *HOPL*, pages 1–55. ACM, 2007.

[20] J.J.A. Keiren and M.A. Reniers. Type checking mCRL2. Computer Science Report No. 11-11, Eindhoven University of Technology, 2011.

[21] R. Mariani. My History of Visual Studio (Part 10, final). http://blogs.msdn.com/ricom/archive/2009/10/19/my-history-of-visual-studio-part-10-final.aspx. Visited: September 25, 2012.

[22] J. McLoone. Eating Your Own Dogfood. http://blog.wolfram.com/2007/05/10/eating-your-own-dogfood/, May 2007. Visited: September 25, 2012.

[23] P.D. Mosses. Exploiting Labels in Structural Operational Semantics. In *SAC*, pages 1476–1481, 2004.

[24] P.D. Mosses. Modular Structural Operational Semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[25] 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 *LNCS*, pages 47–62. Springer, 2010.

[26] G.D. Plotkin. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

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

[28] 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.

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

[30] F.P.M. Stappers. *Bridging Formal Models: An Engineering Perspective*. PhD thesis, Eindhoven University of Technology, 2012. To appear.

[31] F.P.M. Stappers, M.A. Reniers, J.F. Groote, and S. Weber. Dogfooding the structural operational semantics of mCRL2. Computer Science Report No. 11-18, Eindhoven, 2011.

[32] F.P.M. Stappers, M.A. Reniers, and S. Weber. Transforming SOS Specifications to Linear Processes. In *FMICS*, volume 6959 of *LNCS*, pages 196–211. Springer, 2011.

[33] F.P.M. Stappers, M.A. Reniers, and S. Weber. Transforming SOS specifications to linear processes. Technical Report 11-07, Eindhoven University of Technology, 2011.

[34] F.P.M. Stappers, M.A. Reniers, and S. Weber. Transforming SOS Specifications to Linear Processes. Computer Science Report No. 11-07, Eindhoven University of Technology, May 2011.

[35] 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.

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