Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic (Extended Version)

Christiano de O. Braga
PUC-Rio & SRI International
cbraga@inf.puc-rio.br

E. Hermann Hæusler
PUC-Rio
hermann@inf.puc-rio.br

José Meseguer
SRI International
meseguer@csl.sri.com

Peter D. Mosses
BRICS & University of Aarhus
pdmosses@brics.dk

PUC-RioInf.MCC39/00 September, 2000

Abstract: Action semantics (AS) is a framework for specifying the semantics of programming languages, in a very modular and readable way. Recently, the operational semantics of action notation (action semantics's specification language) has been rewritten using Modular SOS (MSOS), a new modular approach for specifying operational semantics. The new modular specification of action notation facilitates the creation of extensions to action semantics, to deal with new concepts, such as components. The Maude Action Tool uses the reflective capabilities of rewriting logic, implemented on the Maude system, to create an executable environment for action semantics and its potential extensions. This is achieved by a mapping between the MSOS and rewriting logic formalisms which, when applied to the MSOS semantics of each facet of action notation, yields a corresponding rewrite theory. Such rewrite theories are executed on *action programs*, that is, on the action notation translation of a given program P in a language L, according to L's action semantics.

Keywords: action semantics, rewriting logic, formal language transformation.

Resumo: Semântica de ações é um *framework* para a especificação modular da semântica de linguagens de programação. Recentemente, a semântica operacional da notação de ações (a linguagem de especificação de semântica de ações) foi reescrita utilizando SOS modular (MSOS), uma nova abordagem modular para a especificação de semântica operacional. A nova especificação modular da notação de ações facilita a criação de extensões a semântica de ações que permitem manipular novos conceitos como componentes. Maude Action Tool (MAT) usa as capacidades reflexivas da lógica de reescrita, implementada no sistema Maude, para criar um ambiente executável para semântica de ações e suas potenciais extensões. MAT é definida através de um mapeamento entre MSOS e lógica de reescrita que quando aplicado a especificação MSOS das facetas da notação de ações produz uma teoria de reescrita. Estas teorias são executadas em *action programs*, ou seja, a tradução em ações de um programa P numa linguagem L seguindo a semântica de ações de L.

Palavras-chave: semântica de ações, lógica de reescrita, tranformação formal entre linguagens.

Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic (Extended Version)

Christiano de O. Braga[*] E. Hermann Haeusler[†]

José Meseguer[‡] Peter D. Mosses[§]

September, 2000

Abstract

Action semantics (AS) is a framework for specifying the semantics of programming languages, in a very modular and readable way. Recently, the operational semantics of action notation (action semantics's specification language) has been rewritten using Modular SOS (MSOS), a new modular approach for specifying operational semantics. The new modular specification of action notation facilitates the creation of extensions to action semantics, to deal with new concepts, such as components. The Maude Action Tool uses the reflective capabilities of rewriting logic, implemented on the Maude system, to create an executable environment for action semantics and its potential extensions. This is achieved by a mapping between the MSOS and rewriting logic formalisms which, when applied to the MSOS semantics of each facet of action notation, yields a corresponding rewrite theory. Such rewrite theories are executed on *action programs*, that is, on the action notation translation of a given program P in a language L, according to L's action semantics.

Contents

[*]Dep. de Informática, PUC-Rio & Computer Science Laboratory, SRI International

[†]Dep. de Informática, PUC-Rio

[‡]Computer Science Laboratory, SRI International

[§]BRICS & Department of Computer Science, University of Aarhus

1 Introduction

Action semantics (AS) [13, 14] is a framework for modular specification of the semantics of programming languages. Action notation (AN) is the specification language of action semantics. Action notation specifications are organized in so-called *facets*, allowing new features to be added to a language without the burden of having to reformulate what has already been specified. AN has another interesting characteristic: it allows the creation of very readable specifications, close to English text, yet remaining completely formal.

AN has a detailed operational semantics. Its original specification [13, Appendix C] was, though precise, monolithic. Recently, Mosses has rewritten it using a new technique called Modular SOS (MSOS) [15]. The MSOS of AN is not only more modular and clearer but it also facilitates the extension of AN to support new semantic structures.

The Maude system [4] is an efficient implementation of rewriting logic, a logic that has been proposed as a logical and semantic framework [11] into which different specification languages can be embedded. Thanks to its reflective capabilities, the Maude system allows the creation of executable environments for the specification languages thus embedded.

This paper reports on a new tool, the Maude Action Tool (MAT): an executable environment for action semantics via the mapping of the MSOS of AN into rewriting logic. MAT allows the execution of programs based on their action semantics. Furthermore, the mapping to rewriting logic opens up the possibility of reasoning formally about the properties of AN definitions, using the formal tools [9, Chapter 5],[5] and model checking [8] capabilities of Maude.

MAT is an action semantics-driven interpreter: given an action semantics description of a programming language L, L_{ASD}, and a program written in L, $PROG_L$, the latter is executed according to L_{ASD}. This is done by first applying a *desugaring* function to L_{ASD}, producing an equational theory, L_{RL} (a Maude functional module) specifying L's abstract syntax, semantic entities and semantic equations. The *desugaring* function maps action semantics descriptions, which can be viewed as elements of the datatype AS, to equational theories, which in Maude are the elements of the datatype `FModule` of functional modules.

$$desugaring : AS \rightarrow \texttt{FModule}$$

The action semantics $PROG_{AN}$ of $PROG_L$ is then obtained using the equations in the theory $L_{RL} = desugaring(L_{ASD})$ to replace all L constructs by their associated action denotations. Finally, $PROG_{AN}$ is rewritten according to action notation's rewrite theory, AN_{RL}.

The *semantics* function sends MSOS specifications, which can be viewed as elements of a datatype $MSOS$, to rewrite theories, which in Maude are represented as elements of the datatype `Module` of system modules. Applying the *semantics* function to the MSOS specification of AN, AN_{MSOS}, generates action notation's rewrite theory AN_{RL}.

$$semantics : MSOS \rightarrow \texttt{Module}$$

The fact that equational and rewrite theories are available in Maude as *data* elements of respective sorts `FModule` and `Module` is part of Maude's reflective capabilities [4, 3]. Therefore the *desugaring* and *semantics* functions are metalevel functions. They use the metalevel interface of Maude, which allows the parsing, and pretty-printing of modules and terms, as well as the application of equations and rewrite rules according to user-defined strategies. For a more detailed explanation of Maude's metalevel interface, we refer to [4, Section 2.5].

Plan of the paper As explained above, the action semantics description of a programming language L, L_{ASD}, together with the Modular SOS of action notation, AN_{MSOS}, constitute the input to MAT. Section 2 presents (very briefly) the MSOS framework, its motivation, and the MSOS of AN, and Section 3 describes in more details the *semantics* mapping, explaining how the MSOS of AN is mapped to rewriting logic using Maude's object-oriented specifications. Section 4 concludes the paper with a discussion of related work, MAT's current state and ongoing work, and a sketch of an action notation extension to support software components.

2 Modularity in SOS and the Modular SOS of Action Notation

Structural Operational Semantic (SOS), sometimes called small-step operational semantics, was first proposed by Plotkin [22], where transitions specify how an abstract machine moves from one state to another. (Big-step or "natural" semantics may formally be regarded as a special case of small-step semantics.)

Partially stimulated by Wansbrough and Hamer's [24] work on modularity of the semantics of action notation using a monadic approach, Mosses [15] has developed a variant of the SOS framework called Modular SOS (MSOS). The main idea of MSOS is to encapsulate the semantic structures in the labels of the rules. If a new semantic structure is added, the existing rules do not change: the label variables simply range over a larger set.

Section 2.1 illustrates the modularity problem in SOS. Section 2.2 explains how MSOS copes with it and briefly explains how the operational semantics of action notation can be defined using the MSOS approach.

2.1 The SOS Modularity Problem

To illustrate the SOS modularity problem, let us consider the following SOS rules for the evaluation of arithmetical addition expressions.

$$\begin{array}{ll} e ::= v \mid e_1 \texttt{ + } e_2 & \\ v ::= n & , n \in \mathbb{N} \end{array}$$

$$\frac{n = n_1 + n_2}{n_1 \texttt{ + } n_2 \to n} \qquad \frac{e_1 \to e_1'}{e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2} \qquad \frac{e_2 \to e_2'}{v_1 \texttt{ + } e_2 \to v_1 \texttt{ + } e_2'} \tag{1}$$

If one now considers a functional extension[1], the existing rules would have to be modified in order to support the notion of environments, besides the addition of the rule for function application. In the original formulation of the rules, the transition relation only involved expressions as configurations. Now it involves both expressions and an environment, so all the rules have to be reformulated:

$$\begin{array}{l} e ::= \ldots \mid x \\ v ::= \ldots \mid \lambda x (e) \\ x \in Var,\ \rho \in Environment = Var \to Val \end{array}$$

$$\frac{n = n_1 + n_2}{\rho \ \vdash \ n_1 \texttt{ + } n_2 \to n} \tag{2}$$

$$\frac{\rho \ \vdash \ e_1 \to e_1'}{\rho \ \vdash \ e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2} \qquad \frac{\rho \ \vdash \ e_2 \to e_2'}{\rho \ \vdash \ v_1 \texttt{ + } e_2 \to v_1 \texttt{ + } e_2'} \tag{3}$$

$$\frac{\rho \ \vdash \ e_1 \to e_1'}{\rho \ \vdash \ e_1\ e_2 \to e_1'\ e_2} \qquad \frac{\rho \ \vdash \ e_2 \to e_2'}{\rho \ \vdash \ v_1\ e_2 \to v_1\ e_2'} \tag{4}$$

$$\frac{\rho[x \mapsto v] \ \vdash \ e \to e'}{\rho \ \vdash \ \lambda x (e)\ v \to e'} \qquad \frac{\rho(x) = v}{\rho \ \vdash \ x \to v} \tag{5}$$

A further imperative extension would cause more modifications, adding the notion of stores. The transition relation now involves the syntax, the environment, and the store, and we obtain the following SOS rules:

$$\begin{array}{l} e ::= \ldots \mid l \texttt{ := } v \\ v ::= \ldots \mid l \mid \texttt{()} \\ l \in Loc,\ \sigma\ \sigma' \in Store = Loc \to Val \end{array}$$

[1] Here we illustrate extension with dynamic bindings, since to specify static bindings in small-step SOS, either auxiliary configurations or a substitution operation would be needed.

$$\frac{n = n_1 + n_2}{\rho \ \vdash \ \langle n_1 \texttt{ + } n_2, \sigma\rangle \rightarrow \langle n, \sigma\rangle} \tag{6}$$

$$\frac{\rho \ \vdash \ \langle e_1, \sigma\rangle \rightarrow \langle e_1', \sigma'\rangle}{\rho \ \vdash \ \langle e_1 \texttt{ + } e_2, \sigma\rangle \rightarrow \langle e_1' \texttt{ + } e_2, \sigma'\rangle} \qquad \frac{\rho \ \vdash \ \langle e_2, \sigma\rangle \rightarrow \langle e_2', \sigma'\rangle}{\rho \ \vdash \ \langle v_1 \texttt{ + } e_2, \sigma\rangle \rightarrow \langle v_1 \texttt{ + } e_2', \sigma'\rangle} \tag{7}$$

$$\frac{\rho \ \vdash \ \langle e_1, \sigma\rangle \rightarrow \langle e_1', \sigma'\rangle}{\rho \ \vdash \ \langle e_1 \ e_2, \sigma\rangle \rightarrow \langle e_1' \ e_2, \sigma'\rangle} \qquad \frac{\rho \ \vdash \ \langle e_2, \sigma\rangle \rightarrow \langle e_2', \sigma'\rangle}{\rho \ \vdash \ \langle v_1 \ e_2, \sigma\rangle \rightarrow \langle v_1 \ e_2', \sigma'\rangle} \tag{8}$$

$$\frac{\rho[x \mapsto v] \ \vdash \ \langle e, \sigma\rangle \rightarrow \langle e', \sigma'\rangle}{\rho \ \vdash \ \langle \lambda x \texttt{(} e \texttt{)} \ v, \sigma\rangle \rightarrow \langle e', \sigma'\rangle} \qquad \frac{\rho(x) = v}{\rho \ \vdash \ \langle x, \sigma\rangle \rightarrow \langle v, \sigma\rangle} \tag{9}$$

$$\rho \ \vdash \ \langle l \texttt{ := } v, \sigma\rangle \rightarrow \langle \texttt{()}, \sigma[l \mapsto v]\rangle \qquad \frac{\sigma(l) = v}{\rho \ \vdash \ \langle l, \sigma\rangle \rightarrow \langle v, \sigma\rangle} \tag{10}$$

2.2 Label Transformers and The Modular SOS of Action Notation

To improve the modularity of SOS specifications, and in particular to avoid the need for such reformulations as illustrated above, Mosses has proposed the use of labels in a labelled transition system to encapsulate the semantic structures. Every semantic structure is a component in the label structure, changed and accessed via *set* and *get* functions, independently of the other components.

In categorical terms, the labels can be seen as the arrows of a category, where the semantic states are the objects. For this reason such a transition system is called an Arrow-Labelled Transition System (ALTS). The source $pre(\alpha)$ of a label α on a transition represents the state before the transition, and its target $post(\alpha)$ represents the state afterwards. Two transitions can be adjacent only when their labels are composable. Identity labels ι have the same source and target, so transitions labelled with them never affect the state.

A label represents the state of the system both before and after a transition, possibly with some more information associated with the transition itself (such as the usual synchronization signals of CCS).

Formally, SOS is based on the concept of labelled transition systems (LTS). An LTS is a structure $(\Gamma, T, \mathbb{A}, \longrightarrow)$ where Γ is a set of configurations, $T \subseteq \Gamma$ is the set of terminal configurations, $\mathbb{A}$ is the set of labels and $\longrightarrow \subseteq \Gamma \times \mathbb{A} \times \Gamma$ is the transition relation.

MSOS is based on the concept of ALTS, which are labelled transition systems where the set of labels $\mathbb{A}$ is the set of the arrows of a category. The objects of $\mathbb{A}$, written $|\mathbb{A}|$, are the states of the ALTS. The composition of arrows α_1 and α_2 is defined iff $post(\alpha_1) = pre(\alpha_2)$. The identity arrows of $\mathbb{A}$ are written $\mathbb{I}^{\mathbb{A}}$, or simply $\mathbb{I}$ when $\mathbb{A}$ is evident; they are called *silent* or *unobservable* arrows and are used to label transitions that merely reduce configurations. For two transitions $\gamma \xrightarrow{\alpha_1} \gamma'$, $\gamma'' \xrightarrow{\alpha_2} \gamma'''$ to be adjacent, two conditions must be met, namely $\gamma' = \gamma''$, and $post(\alpha_1) = pre(\alpha_2)$. Terminating or infinite sequences of adjacent transitions then define the *computations* of the ALTS.

An ALTS can be reduced straightforwardly to a labelled transition system by moving back the states to the configuration and forgetting that the labels are the arrows of a category.

A key concept in MSOS is the one of *label transformers*. Label transformers assume that the category $\mathbb{A}$ comes equipped with additional structure, so that we can think of the arrows in $\mathbb{A}$ as having "components". We can get to each component by using a particular index in a set of indices. That is, it is assumed that there are sets *Index*, of indices, and *Univ*, of information structures, and functions $set : \mathbb{A} \times Index \times Univ \to \mathbb{A}$ and $get : \mathbb{A} \times Index \to Univ$, where the intuitive meaning is that $set(\alpha, i, u)$ changes the component of α indexed by i to u, leaving the rest unchanged; and $get(\alpha, i)$ yields the component of α indexed by i.

A label transformer T involves adding a fresh new index, say i, to the original set *Index*, and transforms the original category of labels $\mathbb{A}$ to the product category $T(\mathbb{A}) = \mathbb{A} \times \mathbb{C}_T$. The original *set* and *get* functions are extended in the obvious way, that is, by treating the arrows $\beta \in \mathbb{C}_T$ as the information stored in the new index i.

Mosses has identified three such label transformers that can be used as basic building blocks to extend the label categories of MSOS specification in a very modular way [15]. They correspond to the following three basic choices of $\mathbb{C}_T$:

- a discrete category E, typically understood as a set of environments, giving rise to the label transformer ***ContextInfo***(i, E).
- a cartesian product S^2, understood as a binary relation, and viewed as a category with set of objects S and arrow composition given by relational composition. S is typically understood as a set of stores. This gives rise to the label transformer ***MutableInfo***(i, S).
- a monoid (A, f, τ), viewed as a one-object category, and is typically understood as a set of actions. The ***EmittedInfo***(i, A, f, τ) label transformer rises with this choice.

MSOS specifications have two key ingredients: the labelled transition rules, that specify how configurations change, and the category of labels, that encapsulate the pre and post states and additional semantic information. As already mentioned, the category of labels can be extended by means of label transformers without any need for modifying the previous given transition rules in any such extensions, that is, in a fully modular way.

We can exemplify the modularity of MSOS with the language introduced in Section 2.1. The MSOS labelled transition rules for that language can be specified as follows:

$$\frac{n = n_1 + n_2}{n_1 \texttt{+} n_2 \xrightarrow{\iota} n} \qquad \frac{e_1 \xrightarrow{\alpha} e_1'}{e_1 \texttt{+} e_2 \xrightarrow{\alpha} e_1' \texttt{+} e_2} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{v_1 \texttt{+} e_2 \xrightarrow{\alpha} v_1 \texttt{+} e_2'} \tag{11}$$

$$\frac{e_1 \xrightarrow{\alpha} e_1'}{e_1\ e_2 \xrightarrow{\alpha} e_1'\ e_2} \qquad \frac{e_2 \xrightarrow{\alpha} e_2'}{v_1\ e_2 \xrightarrow{\alpha} v_1\ e_2'} \tag{12}$$

$$\frac{\alpha' = set(\alpha, env, get(\alpha, env)[x \mapsto v]) \qquad e \xrightarrow{\alpha'} e'}{\lambda x(e)\ v \xrightarrow{\alpha} e'} \tag{13}$$

$$\frac{get(\iota, env)(x) = v}{x \xrightarrow{\iota} v} \tag{14}$$

$$\frac{\alpha = set_{post}(\iota, sto, get_{pre}(\iota, sto)[l \mapsto v])}{l \texttt{ := } v \xrightarrow{\alpha} ()} \tag{15}$$

$$\frac{get_{pre}(\iota, sto, l) = v}{l \xrightarrow{\iota} v} \tag{16}$$

The label category is the following nested application of label transformers:

$$\boldsymbol{MutableInfo}(sto, Store)(\boldsymbol{ContextInfo}(env, Environment)(\mathbb{1}))$$

That is, $Index = \{env, sto\}$ corresponding to adding two new "components" to the trivial category $\mathbb{1}$ (consisting of just one object and one arrow), namely the arrows and the objects of the categories formed from the sets Environment and Store by the label applied transformers. The set_{post} and get_{pre} functions are auxiliary functions involved in the label transformer ***MutableInfo***, used in the imperative language extension.

Note that the above rules did not require any change to accommodate the functional and imperative extensions and that they would not have to be changed at all in a further extension. To allow concurrency and message-passing, for instance, one would add a new semantic structure to the labels. The new rules would *set* and *get* data to and from the new index involved in the extension, but the previous rules wouldn't have to be changed [15].

Action notation has been given an MSOS semantics [16]. Such an MSOS semantics is defined in a modular way for each facet. This involves defining the syntax of actions for each facet, the transition rules based on that syntax (so that a transition $\gamma \xrightarrow{\alpha} \gamma'$ involves syntactic structures γ and γ' defined by the facet's syntax), and the corresponding category of labels α, which is typically built using one or more label transformers. The key advantage of this style of semantics over the previous SOS semantics for action notation [13] is its modularity, in that the addition of new constructs and facets to AN does not require any changes to the previously given semantic definitions.

3 Mapping Action Notation to Rewriting Logic

In order to describe a mapping from one language to another, one must relate their syntaxes and semantics. [2]

Arrow-labelled Transition Systems are the models of MSOS specifications and $\mathcal{R}$-systems are the models of rewrite theories. Section 3.1 briefly describes the computational interpretation of $\mathcal{R}$-systems and also establishes a relationship between these two systems.

[2] In this paper we will use the word *transformation* when referring to the syntactical mapping between the languages of the MSOS and rewriting logic frameworks and simply *mapping* when referring to the relationship of the models of the frameworks. We also use the word *mapping* to relate MSOS and rewriting logic in a general way.

In [16] Mosses has presented the MSOS of action notation using the CASL language. In this paper the syntax of MSOS is given using the MSOS Specification Language (MSOS-SL), a language designed to capture MSOS' most important concepts, namely label categories and arrow-labelled transition rules.[3] The syntax of rewriting logic is given by the Maude language. Section 3.2 presents both MSOS-SL and Maude syntaxes and the transformation function from the former to the latter. An excerpt of action notation's basic facet is used to exemplify both MSOS-SL and Maude syntaxes. This example also illustrates the application of the transformation function to the MSOS specification of action notation.

Section 3.3 describes an example of the use of the Maude Action Tool, an implementation of the mapping described in this Section using the Maude system. Section 3.3 also describes an action notation specification of a language using Maude's syntax, L_{RL}, and the result of a computation of a small program, P_L. The interpretation process uses AN_{RL}, a rewrite theory of action notation's modular structural operational semantics produced via the application of the mapping to action notation's MSOS specification.

3.1 Mapping Arrow-labelled Transition Systems to $\mathcal{R}$-systems

Introduced in [12] (and summarized in the next paragraph), the computational interpretation of $\mathcal{R}$-systems will be used in this section in order to explain the relationship between Arrow-labelled Transition Systems and $\mathcal{R}$-systems.

$\mathcal{R}$-systems are *systems* equipped with algebraic structures. A *system* is a state machine with transitions, from one state to another, which are reflexive and transitive. In other words, a system is a *category* whose objects are the system's states and morphisms are the system's transitions. Quoting [12, page 104]:

> What the *algebraic structure* captures is twofold. Firstly, the states themselves are *distributed* according to such a structure, for Petri nets (see [12, Section 5.3.1]) the distribution takes the form of a multiset that we can visualize with tokens and places (...). Secondly, concurrent transitions are themselves distributed according to the same algebraic structure; this is what the notion of $\mathcal{R}$-systems captures, and is for example manifested in the concurrent firing of Petri nets and, more generally, in any type of concurrent rewriting. The expressive power of rewrite theories to specify concurrent transition systems is greatly increased by the possibility of having not only transitions, but also parameterized transitions, i.e. procedures. (...) The rewrite rule is then a procedure (...) (and) the fact that (the local application of a substitutions can happen locally and) can take

[3]Arrow-labelled transition rules are the labelled transition rules specified in MSOS, whose labels are the arrows of the label category.

System	$\leftrightarrow$	Category
State	$\leftrightarrow$	Object
Transition	$\leftrightarrow$	Morphism
Procedure	$\leftrightarrow$	Natural Transformation
(Distributed) State Structure	$\leftrightarrow$	Algebraic Structure

Figure 1: Systems and $\mathcal{R}$-systems correspondence

place concurrently with other transitions "below" is precisely what the concept of a natural transformation formalizes.

Thus, concurrent rewritings [12] are (informally) defined as one-step rewritings which may have several substitutions applied locally in a (distributed) state. The correspondence among systems and $\mathcal{R}$-systems is summarized in Figure 1.

As explained in Section 2.2, an Arrow-labelled Transition System is a transition system whose labels are the arrows of a category, formalized as a structure $(\Gamma, T, \mathit{Morph}(\mathbb{A}), \longrightarrow)$ where Γ is a set of configurations, $T \subseteq \Gamma$ is the set of terminal configurations, $\mathbb{A}$ is the label category, and $\longrightarrow \subseteq \Gamma \times \mathit{Morph}(\mathbb{A}) \times \Gamma$ is the transition relation with $\mathit{Morph}(\mathbb{A})$ being the set of morphisms (or arrows) of the category $\mathbb{A}$. The objects of $\mathbb{A}$, written $|\mathbb{A}|$, are to be interpreted as the states of the ALTS. The composition of the arrows α_1 and α_2 is defined iff $\mathit{post}(\alpha_1) = \mathit{pre}(\alpha_2)$. The identity arrows of $\mathbb{A}$ are written $\mathbb{I}^{\mathbb{A}}$, or simply $\mathbb{I}$ when $\mathbb{A}$ is evident; they are called *silent* or *unobservable* arrows and are used to label transitions that merely reduce configurations. For two transitions $\gamma \overset{\alpha_1}{\to} \gamma'$, $\gamma'' \overset{\alpha_2}{\to} \gamma'''$ to be adjacent, two conditions must be met, namely $\gamma' = \gamma''$, and $\mathit{post}(\alpha_1) = \mathit{pre}(\alpha_2)$. Terminating or infinite sequences of adjacent transitions then define the *computations* of the ALTS.

The correspondence between systems and $\mathcal{R}$-systems may be used to establish a relationship between ALTS and $\mathcal{R}$-systems if we *map the ALTS to a system*. The main idea is to move to the configuration, from the labels, the semantic information. Formally, we have to map the transition relation from a ternary relation of the form

$$\gamma \xrightarrow{\alpha} \gamma' : \Gamma \times \mathit{Morph}(\mathbb{A}) \times \Gamma$$

to a binary relation on multisets of the form

$$\gamma\ o \longrightarrow \gamma'\ o' : (\Gamma \times |\mathit{Pre}^{\mathbb{A}}|) \times (\Gamma \times |\mathit{Pre}^{\mathbb{A}}|)$$

where $\mathit{Pre}^{\mathbb{A}}$ is the *pre-order* category generated from $\mathbb{A}$.

Definition 1 (Pre-order) *A pre-order set $(P, \leq)$, with $\leq$ a reflexive and transitive (ordering) relation, gives rise to a category whose objects are the elements of set P. Between each pair of objects p and p' with $p \leq p'$, there is a* single *arrow representing this fact; there is no arrow from p to p' when $p \not\trianglelefteq p'$. Composition of arrows is clearly associative, since there is at most one arrow between any given pair of objects. The identity law corresponds to the reflexive* *condition for pre-orders while the existence of composite arrows corresponds to transitivity.*

We recall now the definition of a (freely generated) monoid.

Definition 2 (Monoid) *A (free) monoid* $(P^*, \cdot, e)$*, freely generated from* P*, may be represented as a category with a single object. The elements of* P^* *are represented as arrows from this object to itself, the identity element* e *is represented as the identity arrow, and the operation* $\cdot$ *is represented as composition of arrows.*

Before giving the mapping of label categories to pre-order categories, let us present the definition of a *prefix*.

Definition 3 (Ordering relation $\leq$) *Consider a monoid as in Definition 2. An element* $p \in P^*$ *is a prefix of* $p' \in P^*$ *if the following formula holds.*

$$p \leq p' \quad = \quad \exists p_1 \in P^*.\ p \cdot p_1 = p' \tag{17}$$

This formula defines an ordering relation *between* p *and* p'*.*

Definition 4 (Pre-order Label Category) *The functor (map) that transforms a label category* $\mathbb{A}$ *into a pre-order category* $Pre^{\mathbb{A}}$ *is defined inductively in the following way.*

$$\begin{aligned}
map(\mathbb{1}) \quad &= \quad \mathbb{1} && (18)\\
map(\boldsymbol{ContextInfo}(i,E)(\mathbb{A})) \quad &= \quad \boldsymbol{ContextInfo}(i,E)(map(\mathbb{A})) && (19)\\
map(\boldsymbol{MutableInfo}(i,S)(\mathbb{A})) \quad &= \quad \boldsymbol{MutableInfo}(i,S)(map(\mathbb{A})) && (20)\\
map(\boldsymbol{EmittedInfo}(i,A,f,\tau)(\mathbb{A})) \quad &= && (21)\\
MonoidAsPreorder(i,m)(map(\mathbb{A})) \quad & \quad\quad where\ m = Monoid(A,f,\tau)
\end{aligned}$$

The $MonoidAsPreorder(i, Monoid(A,f,\tau))$ *may be seen as a representation*[4] *of the monoid category as a pre-order category with an ordering relation as in Definition 3.* $MonoidAsPreorder(i,M)(\mathbb{A})$ *adds a pre-order component, as in Definition 1, to the objects of the category* $\mathbb{A}$*. The objects of the category* $\mathbb{A}$*, at index* i*, will be the elements of the set* $(A,\leq)$*, where the* $\leq$ *relation of the pre-order is the ordering relation defined in equation 17.*

The label category $\mathbb{A}$ is mapped into a pre-order to give a homogeneous treatment to the label category in the mapping from ALTS to systems. In this way only the *objects* from $\mathbb{A}$ have to be handled and the morphisms may be abstracted.

Now we may state the following proposition regarding the mapping of label categories to pre-order label categories.

[4] *MonoidAsPreorder* is in fact a functor from the category of the arrows of the monoid to the pre-order.

Proposition 1 *In the context of a MSOS specification, an ALTS*

$$\mathcal{A} = (\Gamma, T, Morph(\mathbb{A}), \longrightarrow)$$

is equivalent to an ALTS

$$\mathcal{B} = (\Gamma, T, Morph(Pre^{\mathbb{A}}), \longrightarrow)$$

with respect to their computations.

Proof sketch. The pre-order $Pre^{\mathbb{A}}$ preserves the semantic information in the label category $\mathbb{A}$. This is accomplished by the definition of *map* in Definition 4. The label category $\mathbb{A}$ is built via the composition of the label transformers ***ContextInfo***, ***MutableInfo*** and ***EmittedInfo***. The only label transformer that adds structure to the labels of the label category is the ***EmittedInfo***, which enhances the label category structure with a monoid. A pre-order is a special case of a monoid. In this way the ALTS $\mathcal{B}$, formed by mapping the label category $\mathbb{A}$ of $\mathcal{A}$ into a pre-order, is equivalent to $\mathcal{A}$. □

This is the first part of mapping an ALTS into a system: to view its label category as a pre-order, such that the labels of the label category are abstracted. The second part is to add the reflexive transitive closure to the transition relation.

Proposition 2 *The ALTS*

$$\mathcal{B} = (\Gamma, T, Morph(\mathbb{A}), \xrightarrow{*})$$

preserves the finite computations of the ALTS

$$\mathcal{A} = (\Gamma, T, Morph(\mathbb{A}), \longrightarrow),$$

where $\xrightarrow{*}$ *is the reflexive transitive closure of* $\longrightarrow$.

Proof sketch. Every computation in $\mathcal{A}$ is in $\mathcal{B}$ since B's transition relation is the reflexive-transitive closure of A's transition relation. Therefore, every *computation* in $\mathcal{A}$ is a computation in $\mathcal{B}$ by the definition of computation presented in Section 2.2. Moreover, for any finite computation in $\mathcal{B}$ one can find a computation in $\mathcal{A}$ by removing all reflexive steps, and expanding the transitive-closure steps to sequences of single steps. Those non-terminating computations in $\mathcal{B}$ that finish with an infinite sequence of reflexive steps might not correspond to any computations in $\mathcal{A}$. All other computations in $\mathcal{B}$ do. □

There now may be established a relationship between Arrow-labelled Transition Systems and $\mathcal{R}$-systems using the latter's computational interpretation via the mapping from ALTS to a system. In other words, the systems generated by the application of the mapping that we have characterized with Propositions 1 and 2 is isomorphic to $\mathcal{R}$-systems. The correspondence between the

Reduced ALTS	$\leftrightarrow$	System
$o \in \lvert Pre^{\mathbb{A}} \rvert$	$\leftrightarrow$	Object
$t \in \xrightarrow{*}$	$\leftrightarrow$	Morphism
Data types and Label Transformers	$\leftrightarrow$	Algebraic Structure

Figure 2: Correspondence between an ALTS and a System

reduced ALTS, generated by the application of the mapping, and $\mathcal{R}$-systems is summarized in Figure 2.

The last component in the correspondence is the algebraic structure which defines the distributed structure in the concurrent rewritings.

This structure is formed by the data types defined in (or included by) the MSOS specification together with the label category structure defined by the label transformers. They are both mapped to algebraic structures in rewriting logic. From a syntactical point of view, the latter is *transformed* into class declarations in the Maude language. This transformation is part of the subject of Section 3.2.

3.2 Transforming MSOS-SL specifications into Object-oriented Maude specifications

This section describes how MSOS-SL specifications are transformed into object-oriented Maude specifications by presenting:

- excerpts of MSOS specification of action notation's basic facet and the grammar of MSOS-SL;
- the object-oriented Maude specification generated by the application of the transformation function to the MSOS specification of the basic facet;
- and the main aspects of the transformation function itself.

3.2.1 MSOS-SL's Syntax

MSOS-SL is an algebraic specification language designed to capture the most important concepts of the MSOS framework, namely label categories and arrow-labelled transition rules. Figure 3 exemplifies its syntax with action notation's basic facet `commit` and `and` operations.

A module is declared using the `msos` syntax constructor. Firstly the module imports the module `BASIC-DATA` which declares data types used by the `BASIC-FACET` module.

Then the module defines a label category named `Basic`. This label category is constructed by the application of the label transformer `emitted`, parameterized by the index `commitment`, the sort `TruthValue`, the operator `combine` and the `TruthValue` constant `false value`. The last three arguments specify the monoid structure that enhances the arrow structure of the label category, as

```
msos BASIC-FACET is
     inc BASIC-DATA .

     label category Basic is
           emitted(commitment, TruthValue, combine, false value)
     endlc

     grammar is
           Action ::= commit | _and_ : Action Action .
     endg

     var alpha : Morph[Basic] . var iota : IdMorph[Basic] .

     tr alpha = set(iota, commitment, true value) =>
        -----------------------------------------
                commit - alpha -> completed .

     var A1 A2 A1' A2' : Action .

     tr        A1 - alpha -> A1'          =>
        ---------------------------------
        A1 and A2 - alpha -> A1' and A2 .

     tr        A2 - alpha -> A2'          =>
        ---------------------------------
        A1 and A2 - alpha -> A1 and A2' .
endms
```

Figure 3: Excerpt of MSOS-SL specification of action notation's basic facet

follows. The second argument, `TruthValue` in Figure 3, specify the type of the elements of the monoid. The third argument, `combine` in Figure 3, specifies the associative (or concatenation) function of the monoid. The fourth and last argument specifies the neutral element of the monoid structure.

Thirdly the module specifies the `grammar` of the language being defined which in this case is composed of the constructors `commit` and `and`. After the grammar declaration the module declares one variable named `alpha` of type `Morph[Basic]` which is the sort of the morphisms of the category `Basic` and one variable, `iota` of type `IdMorph[Basic]` which is the type of the identity morphisms of the label category `Basic`.

The last part is responsible for the arrow-labelled transition rules declaration. The first rule specifies that the system may change from configuration `commit` to `completed` with a label `alpha` at index `commitment` set to `true value`. Four variables of type `Action` are declared and the last two rules state that the composition of two actions evolves nondeterministically.

Figure 4 presents an excerpt of the MSOS-SL grammar specified in Maude. Nonterminal tokens are represented as sorts and terminals as operations. Basically a MSOS specification has four parts:

- module importation list where modules specifying data types or other label categories may be included;
- label category declaration where the label category is specified using label transformer composition applied to the trivial category or to an already existing one;
- grammar declaration specifying the language whose semantics is being defined using MSOS-SL;
- arrow-labelled transition rules declaration which defines how the arrow-labelled transition system evolves.

3.2.2 Maude's Syntax

Figure 5 shows the object-oriented Maude specification generated by the application of the transformation function to the MSOS-SL specification of the basic facet presented in Figure 3.

The object-oriented module `BASIC-FACET` is declared via the operator `omod`. Then the module imports the modules `BASIC-DATA` and `PREORDER`. The `PREORDER` module is parameterized with the sort `TruthValue` and its operator `f` and constant `neutral` are renamed to operators `combine` and `false value` respectively.

After the importation list a class named `Basic` is declared with one attribute, `commitment`, of type `Preorder[TruthValue]`. Then a subsort relationship between the sorts `Action` and `Message`, the constant `commit` of type `Action`, the infix operator `_and_` of type `Action Action -> Action`, and several variables are declared.

```
--- Module declaration
sorts MSOSModuleComponentList MSOSModuleDecl .

op msos_is_endms : Identifier MSOSModuleComponentList ->
                   MSOSModuleDecl .

sort MSOSModuleComponent .
subsort MSOSModuleComponent < MSOSModuleComponentList .

op __ : MSOSModuleComponentList MSOSModuleComponentList ->
        MSOSModuleComponentList .

sorts   ModuleInclusionDeclList LabelCategoryDeclList GrammarDecl
        VarDeclList TransitionRuleDeclList .
subsort ModuleInclusionDeclList LabelCategoryDeclList GrammarDecl
        VarDeclList TransitionRuleDeclList < MSOSModuleComponent .

--- Module inclusion declaration
subsort ModuleInclusionDecl < ModuleInclusionDeclList .
op inc_ : Identifier -> ModuleInclusionDecl .

--- Label category declaration
sorts LabelCategoryDecl LabelTransfomerDeclList .

subsort LabelCategoryDecl < LabelCategoryDeclList .

op label`category_is_endlc :
   Identifier LabelTransformerDeclList -> LabelCategoryDecl .
op label`category_adds`to__endlc :
   Identifier Identifier LabelTransformerDeclList -> LabelCategoryDecl .

--- Grammar declaration
sort GrammarComponent .
op grammar`is_endg : GrammarComponent -> GrammarDecl .

--- Variable declaration
sort VarDecl . subsort VarDecl < VarDeclList .

op var_`:_`. : Identifier LabelType -> VarDecl .
op vars_`:_`. : IdentifierList LabelType -> VarDecl .
op var_`:_`. : Identifier Identifier -> VarDecl .
op vars_`:_`. : IdentifierList Identifier -> VarDecl .

op Morph`[_`] : Identifier -> LabelType .
op IdMorph`[_`] : Identifier -> LabelType .

--- Arrow-labelled transition rule declaration
sorts PremiseList Conclusion .
op tr_=>_`. : PremiseList Conclusion -> TransitionRuleDecl .
```

Figure 4: MSOS-SL grammar excerpt

```
omod BASIC-FACET is
     pr BASIC-DATA .

     pr PREORDER[TruthValue]
        *(op f to combine, op neutral to false value) .

     class Basic | commitment : Preorder[TruthValue] .

     subsort Action < Message .

     op commit : Action .
     op _and_ : Action Action -> Action .

     vars o o' : Oid . var ptv : Preorder[TruthValue] .
     var A1 A2 A1' A2' : Action .
     vars attrSet attrSet' : AttrSet .

     rl [BASIC-FACET-1] :
        commit
        < o : Basic | commitment : ptv, attrSet > =>
        completed
        < o : Basic | commitment: combine(ptv, true value), attrSet > .

     crl [BASIC-FACET-2] :
         A1 and A2
         < o : Basic | attrSet > =>
         A1' and A2 < o : Basic | attrSet' >
      if A1 < o : Basic | attrSet > => A1' < o : Basic | attrSet' > .

     crl [BASIC-FACET-3] :
         A1 and A2 < o : Basic | attrSet > =>
         A1 and A2' < o : Basic | attrSet' >
      if A2 < o : Basic | attrSet > => A2' < o : Basic | attrSet' > .
endom
```

Figure 5: Excerpt of the generated MSOS-SL specification of action notation's basic facet

The last three declarations are rewrite rule declarations. The first one declares an unconditional rewrite rule, as opposed to the following conditional ones. In all of them, the configuration being rewritten is a multiset of `Actions` and objects, which are instances of the class `Basic` that defines the data handled by action notation's basic facet. Objects are represented in a record-like syntax. An object

$$< \mathtt{o} : \mathtt{C} \mid \mathtt{a_1} : \mathtt{AttrSort_1},\ \ldots,\ \mathtt{a_n} : \mathtt{AttrSort_n} >$$

has an object identifier `o` of sort `Oid`. `C` is the class which the object is an instance of. $\mathtt{a_i}$ is an attribute of sort $\mathtt{AttrSort_1}$. A rule condition of the form

$$< \mathtt{o} : \mathtt{C} \mid \mathtt{attrSet} > => < \mathtt{o} : \mathtt{C} \mid \mathtt{attrSet}' >$$

where `attrSet` and `attrSet'` are variables of sort `AttrSet` (set of attributes), means that the attributes of the object `o` may change in the rewrite. More formally, it specifies that a certain rule, which that rewrite is a condition of, could be applied (according to the rewrite strategy being adopted) if there exists a rewrite takes the object `o` with state given by `attrSet` and lead it to a state defined by `attrSet'`. We refer to [4] for a through explanation on Maude's syntax and grammar, and object-oriented specifications.

3.2.3 The Transformation Function

The MSOS-SL grammar has four major parts:

- module importation list declaration;
- label category declaration;
- grammar declaration;
- variable declarations;
- arrow-labelled transition rules declaration.

The *importation list declaration* is transformed directly to a Maude module inclusion declaration.

Label category declarations are transformed into class declarations in the Maude language, where the index parameters in the label transformers declarations are transformed into attributes in the class declarations with type given by sort parameter in the label transformer declaration. The ***EmittedInfo*** label transformer is a special case. The index parameter in the label transformer declaration is transformed into an attribute declaration with type `Preorder` parameterized by the sort parameter in the label transformer declaration, and concatenation function and identity constant renamed with the respective names in the label transformer declaration.

Figure 6 presents the transformation function for the label transformers declaration. Overlined terms are *metarepresented* terms in the Maude language.

```
sort LabTransDeclTransRet .

op <_,_> : AttrDecl ModExp -> LabTransDeclTransRet.

op transform : LabelTransformerDecl -> LabTransDeclTransRet .

vars I S F N : Indentifier .

eq transform(context(I, S)) = I : S .

eq transform(mutable(I, S)) = I : S .
```

$$\begin{array}{l} \texttt{eq} \quad \texttt{transform(emitted(I, S, F, N))} = \\ \quad < \texttt{I} : \overline{\texttt{Preorder[S]}}, \overline{\texttt{Preorder[S]}} * (\texttt{op } \overline{\texttt{f}} \texttt{ to F, op } \overline{\texttt{neutral}} \texttt{ to N}) > \ . \end{array}$$

Figure 6: MSOS-SL label transformers declaration transformation function

The overlined notation is used in order to not to discuss reflection in Maude at this point. We refer to [4] for a through explanation on *metaparsing* in Maude.

Note that the transformation of an ***EmittedInfo*** label transformer declaration has also the effect of adding the inclusion of the module `PREORDER` parameterized by the sort assigned to the variable `S`. This is done because the parameterized sort `Preorder` is part of a *prelude* module that is loaded at initialization time. Thus, the module has to be explicitly included by the MSOS specifications that declare a label category that applies the ***EmittedInfo*** label transformer.

The *grammar declaration* has a simple translation to operation declarations in Maude, where a non-terminal is the sort of a operation's return value. This can be easily seen in the object-oriented module `BASIC-FACET` shown in Figure 5 where the MSOS-SL `grammar` declaration

```
grammar is
        Action ::= commit | _and_ : Action Action .
endg
```

is transformed into the following operation declarations.

```
op commit : Action .
op _and_ : Action Action -> Action .
```

The *variable declaration* part is kept the same modulo the label variable variables, which are replaced by *two* variables of type `Object`, holding the state prior and posterior to the transition.

The transformation of an arrow-labelled transition rule declaration follows the work of Martí-Oliet and Meseguer in [11]. Arrow-labelled transition rules are mapped into conditional rewrite rules where the left hand side and right hand side of the rule play the rôle of the conclusion part of the transition rule and the conditions play the rôle of the premises. This approach leads to a direct interpreter for the language being described, as opposed to the one where transitions are mapped into the terms being rewritten.

Let us now focus on the transformation of `set` and `get` operations, an important part of a rule declaration transformation.

As explained in Section 3.1, the mapping process maps a ternary relation of the form

$$\gamma \xrightarrow{\alpha} \gamma' : \Gamma \times \mathit{Morph}(\mathbb{A}) \times \Gamma$$

to a binary relation on multisets of the form

$$\gamma\ o \longrightarrow \gamma'\ o' : (\Gamma \times |\mathit{Pre}^{\mathbb{A}}|) \times (\Gamma \times |\mathit{Pre}^{\mathbb{A}}|)$$

where $\mathit{Pre}^{\mathbb{A}}$ is the pre-order category generated from $\mathbb{A}$.

Informally, the semantic information that was encapsulated inside the labels is moved to the configurations and the label category is mapped into a pre-order in order to give a homogeneous treatment to the category such that only the objects of the pre-order have to be handled and the arrows may be abstracted.

Thus, `set` and `get` operations on *labels* in a MSOS specification are transformed into `set` and `get` operations on *objects*,[5] in the generated Maude specification. This transformation has to be defined for every label transformer in the MSOS framework.[6]

Let us begin with the ***EmittedInfo*** label transformer. This label transformer adds a monoid structure to the labels of the label category. The equations for the transformation for the `set` operation on labels are presented in Figure 7. The `pre` and `post` functions generate appropriate variable names, of type object, based on the original label variable names, by concatenating the string `pre` and `post` to the variable name.

Informally, the equations specify that the state *after* a transition at an index `i` is equal to the state *prior* to the transition, at index `i`, concatenated with a certain value `v`, whose type was defined in the label category declaration section of the MSOS specification being transformed.

The predicate

```
op context? : LabelCategoryDecl Indentifier -> Boolean .
```

checks whether an index was declared as an ***EmittedInfo*** index in a given label category declaration. The function

```
op type : VarDecl Identifier -> LabelType .
```

returns the type of a label variable.

[5]The word "objects" is used here in the object-oriented sense of the word and also in categorical terms, since the objects of the label category are mapped into objects in the generated rewrite theory.

[6]If one decides to add a new label transformer to the MSOS framework, the transformation function for `set` and `get` operations on labels will have to be extended. The major property that has to hold is that the label category transformation has to generate a pre-order.

```
var : LC : LabelCategoryDecl . var VDL : VarDeclList .
vars ALPHA ALPHA' IOTA I V : Identifiers .

op transform : LabelCategoryDecl VarDecList Premise -> ConditionSet .
```

```
 ceq  transform(LC, VDL, ALPHA′ = set(ALPHA, I, V))) =
      (pre(ALPHA′) = pre(ALPHA) and
       post(ALPHA′) = set(pre(ALPHA), I, f(get(pre(ALPHA), I), V)))
  if  context?(LC, I) and
      type(VDL, ALPHA) == Morph[getname(LC)] .

 ceq  transform(LC, VDL, ALPHA = set(IOTA, I, V))) =
      (pre(ALPHA) = pre(IOTA) and pre(IOTA) = post(IOTA) and
       post(ALPHA) = set(pre(IOTA), I, f(get(IOTA, I), V)))
  if  context?(LC, I) and
      type(VDL, IOTA) == IdMorph[getname(LC)]) .
```

Figure 7: Transformation for the `set` operation for the ***EmittedInfo*** label transformer

As a consequence of the approach used to transform ***EmittedInfo*** indices, the transformation of `get(ALPHA, i)` is `get(post(ALPHA), i)`.

Since the label component is *sliced* into two components (one in the state prior to the transition and another after a transition), `get(ALPHA, i)` should refer always to the *most recent* value associated with index `i`. If index `i` *changes*, then `get(post(ALPHA), i)` will retrieve the most recent value associated with `i`. Otherwise, `pre(ALPHA) = post(ALPHA)` and `get(post(ALPHA), i)` still reflects the most updated value associated with `i`.

Now let us look at the transformation function for `set` and `get` operations for the ***ContextInfo*** label transformer. The ***ContextInfo*** label transformer adds a component to the objects of the label category. Moreover, the value bound to that index does not change in a transition. The transformation function for the `set` and `get` operations for ***ContextInfo*** label transformer is presented in Figure 8.

The last label transformer is the ***MutableInfo*** label transformer which adds a component to the objects of the label category. As opposed to a ***ContextInfo*** index, the value bound to a ***MutableInfo*** index may change in a transition. The transformation function for the `set`, `get`, `set-post` and `get-pre` operations for ***MutableInfo*** label transformer is presented in Figure 8.

The transformation function still has to be proven correct. It is necessary to prove that the underlying transition system, induced by the rewrite theory generated by the transformation function, preserves the computations of the original ALTS. The proof is done on the depth of the proof terms.

$$
\begin{aligned}
\mathtt{eq}\quad & \mathtt{transform(ALPHA'} = \mathtt{set(ALPHA,\ I,\ V))} = \\
& \mathtt{(pre(ALPHA'))} = \overline{\mathtt{set(pre(ALPHA),\ I,\ V)}}\ \mathtt{and} \\
& \overline{\mathtt{get(pre(ALPHA),\ I)}} = \overline{\mathtt{get(post(ALPHA),\ I)}}\ \mathtt{and} \\
& \overline{\mathtt{get(pre(ALPHA'),\ I)}} = \overline{\mathtt{get(post(ALPHA'),\ I)}})\ .
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{eq}\quad & \mathtt{transform(ALPHA} = \mathtt{set(IOTA,\ I,\ V))} = \\
& \mathtt{(pre(ALPHA'))} = \overline{\mathtt{set(pre(IOTA),\ I,\ V)}}\ \mathtt{and} \\
& \overline{\mathtt{get(pre(ALPHA),\ I)}} = \overline{\mathtt{get(post(ALPHA),\ I)}}\ \mathtt{and} \\
& \mathtt{pre(IOTA)} = \mathtt{post(IOTA)).}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{eq}\quad & \mathtt{transform(V} = \mathtt{get(ALPHA,\ I))} = \\
& \mathtt{(V} = \overline{\mathtt{get(pre(ALPHA),\ I)}}\ \mathtt{and} \\
& \overline{\mathtt{get(pre(ALPHA),\ I)}} = \overline{\mathtt{get(post(ALPHA),\ I)}}\ .
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{eq}\quad & \mathtt{transform(V} = \mathtt{get(IOTA,\ I))} = \\
& \mathtt{(V} = \overline{\mathtt{get(pre(IOTA),\ I)}}\ \mathtt{and} \\
& \mathtt{pre(IOTA)} = \mathtt{post(IOTA))}\ .
\end{aligned}
$$

Figure 8: Transformation for the `set` and `get` operations for the ***ContextInfo*** label transformer

$$
\begin{aligned}
\mathtt{eq}\quad & \mathtt{transform(ALPHA'} = \mathtt{set(ALPHA,\ I,\ V))} = \\
& \mathtt{pre(ALPHA')} = \overline{\mathtt{set(pre(ALPHA),I,V)}}\ .
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{eq}\quad & \mathtt{transform(ALPHA} = \mathtt{set(IOTA,I,V))} = \\
& \mathtt{(pre(ALPHA)} = \overline{\mathtt{set(pre(IOTA),I,V)}}\ \mathtt{and\ pre(IOTA)} = \mathtt{post(IOTA)).}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{eq}\quad & \mathtt{transform(ALPHA'} = \mathtt{set\text{-}post(ALPHA,\ I,\ V))} = \\
& \mathtt{(pre(ALPHA')} = \mathtt{pre(ALPHA)\ and\ post(ALPHA')} = \overline{\mathtt{set(pre(ALPHA),I,V))}}\ .
\end{aligned}
$$

$$
\mathtt{eq}\quad \mathtt{transform(V} = \mathtt{get(ALPHA,I))} = \mathtt{V} = \overline{\mathtt{get(pre(ALPHA,I))}}\ .
$$

$$
\begin{aligned}
\mathtt{eq}\quad & \mathtt{transform(V} = \mathtt{get(IOTA,I))} = \\
& \mathtt{(V} = \overline{\mathtt{get(pre(IOTA),I)}}\ \mathtt{and\ pre(IOTA)} = \mathtt{post(IOTA))}\ .
\end{aligned}
$$

$$
\mathtt{eq}\quad \mathtt{transform(V} = \mathtt{get\text{-}pre(ALPHA,I))} = \mathtt{V} = \overline{\mathtt{get(pre(ALPHA),I)}}\ .
$$

Figure 9: Transformation for the `set`, `get`, `set-post` and `get-pre` operations for the ***MutableInfo*** label transformer

3.3 Using The Maude Action Tool

MAT is an implementation of the mapping of action notation to rewriting logic using the Maude system. In order to interpret a program written in a language L, let us say P_L, using MAT, one has to firstly write down the specification of the syntax and semantics of the language L in action notation (the equational theory L_{RL}) using Maude's syntax. [7]

The program P_L, entered in the system as a term, is reduced according to L_{RL} equational theory, which then produces a term, P_{AN}, which is the program P_L with L's constructs substituted by their action notation counter parts. Finally, P_{AN} is interpreted using the rewriting logic theory, AN_{RL}, created once and for all using the transformation described in Section 3.2, and implemented as a metafunction in the Maude system.[8]

Let us now present an example of the input and output produced by the MAT system.[9]

Figure 10 shows the syntax and semantics of the language SIMPLE (L_{RL}). SIMPLE programs may access the memory via variables, change and retrieve their values, evaluate boolean tests with the less than operator `_<_` and execute statement loops with the `while` operator.

Figure 11 shows the class declaration for the language SIMPLE. Objects instances of class Simple, the *semantic states* of the arrow-labelled transition system, will hold attributes for the storage, from the imperative facet, and transients of the functional facet. For this reason, class `Simple` inherits from `ImperativeFacet` and `FunctionalFacet` classes.

Figure 12 show a SIMPLE program (P_L) that makes two iterations of a `while` loop, via testing the value of a variable that is initialized with value 1 and decremented in every iteration.

Figure 13 shows the execution of P_{AN}. The final configuration is given by a multiset composed of the `completed` keyword and an object. The latter represents the semantic state, that is, the value of the `storage` (imperative facet) and `data` (functional facet) indices, together with `unfolding` and `commitment` indices, attributes related with the control flow of the basic facet.

4 Concluding Remarks and Current Work

We conclude the paper by placing our proposal in the context of related work, summarizing the current state of the MAT system, and sketching a potential extension of action notation to support the notion of software component.

[7] A pre-processor that would receive L_{AN} as input and produce L_{RL} as output could also be written using Maude's reflective functions.

[8] The complete implementation of MAT using the Maude system is composed of a parser, a pretty-printer and the transformation function itself. All implemented as metafunctions in Maude.

[9] The input and output shown here were written to and produced by, respectively, a prototype of MAT, whose current state is discussed in Section 4.2, implementation presented in [6], and the implementation of the `rcrl` strategy presented in [1].

```
in functional.fm
in imperative.fm

(fmod SIMPLE-SYNTAX is
      sort Id Expr Stmt .
      op _:=_ : Id Expr -> Stmt .
      op while_do_ : Expr Stmt -> Stmt .

      op value`of_ : Id -> Expr .
      op _"<"_ : Expr Expr -> Expr .
endfm)

(omod SIMPLE-SEMANTICS is
      inc SIMPLE-SYNTAX .

      inc IMPERATIVE-TRANSITIONS .
      inc FUNCTIONAL-TRANSITIONS .

      subsort Character < Id Cell .
      subsort MachineInt < Expr Storable .

      op execute_ : Stmt -> Action .

      var EXP EXP' : Expr . var STMT : Stmt .
      var ID : Character . var VAL : MachineInt .

      eq execute ( ID := VAL ) = store VAL in ID .

      eq execute ( while EXP do STMT ) =
         (unfolding (
          (give (evaluate EXP)) then (
             (check (NOT(the given TruthValue)))
             or
             ((check (the given TruthValue)) and then
               ((execute STMT) and then unfold))))) .

      op evaluate_ : Expr -> Yielder .

      eq evaluate VAL = VAL .

      eq evaluate value of ID = the MachineInt stored in ID .

      eq evaluate ( EXP "<" EXP' ) =
         (evaluate EXP) less than (evaluate EXP') .
endom)
```

Figure 10: Action Semantics of a simple language using Maude as metanotation

```
(omod OO-SIMPLE-SEMANTICS is
      pr SIMPLE-SEMANTICS .

      class Simple .
      subclass Simple < FunctionalFacet ImperativeFacet .

      sort O . op o : -> O . subsort O < Oid .
endom)
```

Figure 11: The specification of class Simple

```
set show loop stats on .

set show loop timing on .

(eval (execute (while ((value of "a") "<" 2) do ("a" := 2) ))
       < o : Simple | unfolding : completed , commitment : false value ,
                       data : 1 , storage : [ "a" := 1 ] >
.)
```

Figure 12: A (very) simple program

```
Evaluating in OO-SIMPLE-SEMANTICS :

execute ( while value of "a" "<" 2 do "a" := 2 )
 < o : Simple | unfolding : completed , commitment : false value ,
                 data : 1 , storage : [ "a" := 1 ] >  .

Result ANConfiguration :
completed  < o : Simple | commitment : false value ,
                           unfolding : completed ,
                           storage : [ "a" := 2 ] , data : 1 >
```

Figure 13: The result of the execution of the simple program

4.1 Related Work

The Actress system [2, 17], developed by Watt's group at the University of Glasgow, interprets action notation and compiles it into C. It deals with most of action notation, except the communicative facet.

Abaco [7] is an acronym for algebraic-based action compiler, which produces compilers (coded in C++) from action semantics descriptions. An unified algebras[10] [13] translator is responsible for generating dynamic and static libraries, which are then used by the code generator and the generated compiler itself, respectively.

Also related to this work are the ASD [23] tools and the OASIS [20] compiler. The latter provides the most convincing evidence so far of the practical applicability of action-semantics-based compiler generation. It is capable of generating optimizing compilers for simple procedural and functional languages. The former was a set of tools developed by van Deursen and Mosses, which allowed specifying ASDs using a syntax-directed editor (or a plain text one), checking and interpreting them.

The MAT tool contributes to these efforts in the following ways:

- It is formally specified and implemented, following AN's Modular SOS, via the mapping of MSOS to rewriting logic. Such a mapping makes it easy to check the correctness of the embedding and supports formal analysis and verification of the resulting specification.
- It supports the full action notation described in [13], including the communicative facet.
- It preserves the modularity of the MSOS specifications. Specially, the rewrite rules associated to different facets do not have to be modified when those facets are combined. This is achieved by means of the multiple inheritance mechanism of classes is Maude object-oriented specifications.
- It allows experimentation with new semantic structures, through the addition of new facets to action notation, via MSOS. The component extension, sketched in Section 4.3 would be an example of a facet addition.

4.2 Current State of MAT

The Maude Action Tool is currently in a prototype state. We have implemented most of the action notation, including the communicative facet, producing manually the AN_{RL} specification. Once the prototype is finished (i.e., the whole action notation is specified in Maude), it will be able to interpret a program P, written in a language L, according to L's action semantics. Also, at present the specification of L_{RL} has to be manually produced for each language L.

MAT provides a simplified version of the Data Notation described in [13, Appendix E], which parameterizes AN. Any additional datatypes needed for a particular AS specification can be specified, equationally, as Maude modules.

[10] Unified algebras is the algebraic framework used in action semantics type system.

The next step is the encoding of the transformation process for MSOS specifications (the function *semantics*) and the *desugaring* function (see Section 1). The *semantics* function will be crucial for the experiments with extensions to action notation, like the one sketched in Section 4.3, since it will allow the input of MSOS specifications into the system.

4.3 The Component Facet

The original operational semantics of action notation was, though precise, monolithic. In [18], Musicante and Mosses proposed an extension to the communicative facet with shared memory, which required a major change in the operational specification of the kernel of action notation.

Extensions and changes in the new MSOS specification of AN do not imply major changes: they can be made local to a facet. If one decides to adopt a new concurrency model, the other facets will not need to be re-specified, since they are self contained in general.

Stimulated by the new modular semantic description, we plan to extend the framework with a new semantic structure, namely, components. Software components are currently being used as the main technique to achieve interoperability among reusable pieces of software. The work described in [19, 21, 10] represents just a few of them. However, work on components still lacks formalization.

We propose to formalize the notion of software component in an extension of the current action notation, equipping the framework with a mechanism to allow the description of component-based languages. Components written in a language whose semantics is given in terms of the extended action notation will interoperate by construction. The mechanism will also allow connection with already existing components. The new facets will be as follows:

- *synchronous communication*. This specialization of the communicative facet will add the notion of synchronization events.
- *thread*. This facet adds the notion of lightweight processes, known as threads. The main difference between threads and the agents of the communicative facet is that threads share a memory.
- *service*. A service is a special action, that may be dynamically required, and that must satisfy a given specification of its pre- and post-conditions.
- *component*. Finally, the component facet defines the concept of a component, which is a structure with a well-defined interface of required and provided services. Also, a component is (internally) a multi-threaded unit that may signalize and listen to events.

Acknowledgements Braga would like to thank Francisco Durán for his (numerous) explanations about Maude and Full-Maude; to Alberto Verdejo for his suggestions to improve the `rcrl` strategy and for the partial support from

CAPES grant No. CBE/PDEE 0009/1998. This work has been supported by DARPA through Rome Laboratories Contract F30602-97-C-0312, by DARPA and NASA through Contract NAS2-98073, by NSF through Grant CCR-9900334, and by ONR Contract N00012-99-C-0198. In addition, Mosses gratefully acknowledges support from BRICS (Centre for Basic Research in Computer Science), established by the Danish National Research Foundation in collaboration with the Universities of Aarhus and Aalborg, Denmark.

References

[1] C. Braga. Embedding SOS into Maude. 1999.

[2] D. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In *CC'92, Proc. 4th Int. Conf. on Compiler Construction,Paderborn*, volume 641 of *LNCS*, pages 95–109. Springer-Verlag, 1992.

[3] M. Clavel. *Reflection in General Logics and in Rewriting Logic*. PhD thesis, Univesity of Navarre, 1998.

[4] M. Clavel, F. Durán, S. Eker, N. Martí-Oliet, P. Lincoln, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, `http://maude.csl.sri.com`, January 1999.

[5] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98, Sumazu, Japan. CafeOBJ Project*, April 1998.

[6] C. de O. Braga, E. H. Haeusler, J. Meseguer, and P. D. Mosses. Maude action tool: Using reflection to map action semantics to rewriting logic. In *AMAST'00, Proc. 8th Intl. Conf. on Algebraic Methodology and Software Technology, Iowa City, IA, USA*, LNCS, May 2000.

[7] L. C. de Sousa Menezes and H. P. de Moura. The Abaco system an algebraic based action compiler. In A. Haeberer, editor, *AMAST'98, Proc. 7th Intl. Conf. on Algebraic Methodology and Software Technology, Amazonia, Brazil*, volume 1548 of *LNCS*, pages 527–529. Springer, 1999.

[8] G. Denker, J. Meseguer, and C. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DISCEX 2000, Proc. Darpa Information Survivability Conference and Exposition, Hilton Head, South Carolina*, volume 1, pages 251–265. IEEE Computer Society Press, January 2000.

[9] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, Escuela Técnica Superior de Ingeniería Informática, 1999.

[10] R. Ibrahim and C. Szyperski. Formalization of component object model (COM) - the COMEL language. In *ECOOP'98 PhD Workshop*, July 1998.

[11] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, SRI International, August 1993. To appear in D. Gabbay, ed., *Handbook of Philosofical Logic*, Kluwer Academic Publishers.

[12] J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.

[13] P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[14] P. D. Mosses. Theory and practice of action semantics. In *MFCS'96, Proc. 21st Int. Symp. on Mathematical Foundations of Computer Science, Cracow, Poland*, volume 1113 of *LNCS*, pages 37–61. Springer-Verlag, 1996.

[15] P. D. Mosses. Foundations of Modular SOS (extended abstract). In *MFCS'99, Proc. 24th Intl. Symp. on Mathematical Foundations of Computer Science, Szklarska-Poreba, Poland*, volume 1672 of *LNCS*, pages 70–80. Springer-Verlag, 1999. The full version appears as Tech. Report RS-99-54, BRICS, Dept. of Computer Science, Univ. of Aarhus.

[16] P. D. Mosses. A modular SOS for Action Notation. Research Series BRICS-RS-99-56, BRICS, Dept. of Computer Science, Univ. of Aarhus, 1999.

[17] H. Moura and D. A. Watt. Action transformations in the Actress compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh*, volume 786 of *LNCS*, pages 16–30. Springer-Verlag, 1994.

[18] M. A. Musicante and P. D. Mosses. Communicative action notation with shared storage. Tech. Mono PB-452, Dept. of Computer Science, Univ. of Aarhus, 1993.

[19] Object Management Group. *The Common Object Request Broker Architecture and Specification - Revision 2.0*, July 1995. `http://www.omg.org`.

[20] P. Ørbæk. OASIS: An optimizing action-based compiler generator. In *CC'94, Proc. 5th Intl. Conf. on Compiler Construction, Edinburgh*, volume 786 of *LNCS*, pages 1–15. Springer-Verlag, 1994.

[21] J. Penix and P. Alexander. Toward automated component adaptation. In *Ninth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, June 1997. `http://www.ececs.uc.edu/~kbse/pub/ps/seke97.ps`.

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

[23] A. van Deursen and P. D. Mosses. ASD: The action semantic description tools. In *AMAST'96, Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 579–582. Springer-Verlag, 1996.

[24] K. Wansbrough and J. Hamer. A modular monadic action semantics. In *Conference on Domain Specific Languages*, pages 157–170. The USENIX Association, 1997. `http://www.cl.cam.ac.uk/users/kw217/research/msc/thesis/`.