Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 205 (2008) 105–121
www.elsevier.com/locate/entcs

Maude Object-Oriented Action Tool

André Murbach Maidl[1]

Universidade Federal do Paraná
Curitiba, Brazil

Cláudio Carvilhe[2]

Pontifícia Universidade Católica do Paraná
Curitiba, Brazil

Martin A. Musicante[3]

Universidade Federal do Rio Grande do Norte
Natal, Brazil

Abstract

Object-Oriented Action Semantics (OOAS) incorporates object-oriented concepts to the Action Semantics formalism. Its main goal is to obtain more readable and reusable semantics specifications. Moreover, it supports syntax-independent specifications, due to the way classes are written. Maude Object-Oriented Action Tool (MOOAT) is an executable environment for Object-Oriented Action Semantics implemented as a conservative extension of Full Maude and Maude MSOS Tool (MMT). The Modular SOS of Action Notation has been implemented using MMT transitions and Full Maude has been used to implement the Classes Notation. The syntax created by MOOAT is fairly similar to the original Object-Oriented Action Semantics syntax. In addition to it, the tool combines the modularity aspects observed in the object-oriented approach with the efficient execution and analysis of the Maude system. We use MOOAT to describe syntax-independent specifications of programming languages. In this way, we show how Constructive Object-Oriented Action Semantics (COOAS) may be achieved as a combination between Object-Oriented Action Semantics and Constructive Action Semantics (CAS) using MOOAT, in order to increase the modularity aspects observed in the object-oriented formalism. This paper reports on the development of Maude Object-Oriented Action Tool and its application to the formal specification of programming languages.

Keywords: Constructive Action Semantics, Formal Semantics, Maude, Modular Structural Operational Semantics, Object-Oriented Action Semantics.

1 Introduction

Action Semantics [15,21] is a formal framework developed to improve the readability in programming languages semantics definitions. The framework inherits

[1] Email: murbach@inf.ufpr.br
[2] Email: carvilhe@ppgia.pucpr.br
[3] Email: mam@dimap.ufrn.br

doi:10.1016/j.entcs.2008.03.068

characteristics of both Denotational Semantics and Operational Semantics. In Action Semantics, semantic functions specify the meaning of the phrases of a language using *actions*. These actions represent the denotation of the phrases. The Action Notation is defined operationally and contains basic actions and action combinators.

Altough Action Semantics inherently presents good reusability, the standard Action Notation lacks of syntactic support for the definition of libraries and reusable components [9]. In order to overcome this problem, a modular approach for Action Semantics has been proposed in [7], where Action Semantics *modules* allow isolated specifications elements viewing and modules composition. In this way reusability has been successfully enhanced.

Based on the modular approach introduced by Doh and Mosses in [7], Object-Oriented Action Semantics has been proposed in [3] as a method to organize Action Semantics specifications by the use of *objects*. OOAS was defined using SOS [19] transitions in [3], and no tool for describing programming languages semantics using the object-oriented formalism had been implemented yet, as reported by [20].

The strategy of specification, in our vision, is an exclusive user´s choice. This means that the user has the option of representing language specifications utilizing the approach that better suits him. This paper does not discuss the comparison of Modular, Object-Oriented, and other modularization techniques; not even comparisons between AS tools are traced here. Our goal is to supply additional resources to OOAS, providing a tool where it is possible to write OOAS specifications and execute them, and giving a view of how OOAS specifications may be achieved in a constructive way.

In these regards, we propose MOOAT (Maude Object-Oriented Action Tool): the first tool for describing programming languages semantics using OOAS, completely described and available at [11]. MOOAT development was inspired by MAT (Maude Action Tool) [2] and its implementation using a Modular SOS [17] interpreter, which was developed as a Rewriting Logic [13] semantic framework in Maude [6] version 1. However, we propose some changes in OOAS syntax and semantics. One important difference between the tool and the formalism is that MOOAT uses Modular SOS [17] instead of plain SOS [19].

We have used the Maude system [6] to implement the Classes Notation of OOAS and its Action Notation has been implemented using the Modular SOS environment provided by MMT (Maude MSOS Tool) [4,5], mostly in accordance with the Modular SOS for Action Notation proposed in [16].

In addition to the implementation, we present a *constructive* approach for Object-Oriented Action Semantics. Basically, we combine Constructive Action Semantics [10,18] with Object-Oriented Action Semantics, in order to increase the modularity aspects observed in the object-oriented formalism and also to obtain a new syntax-independent style for describing programming languages.

This work is organized as follows: the next section briefly introduces Maude MSOS Tool. Section 3 gives an overview of Object-Oriented Action Semantics. Constructive Action Semantics is summarized in section 4. MOOAT notation and implementation are explained in section 5. In section 6 we discuss how Constructive

Object-Oriented Action Semantics descriptions might be obtained as a case study of MOOAT. Some final remarks and future work are exposed in section 7.

2 Maude MSOS Tool

Structural Operational Semantics (SOS) [19] is a formal framework extensely used to specify programming languages and other frameworks, such as Action Semantics [15]. However, modularity in SOS specifications was left open by Plotkin in [19]. The modularity problem in SOS specifications has been treated by Mosses' Modular SOS (MSOS) [17].

MSOS uses a general transition system, where components like environments and storage are implemented by labeled transitions instead of being part of configurations, in order to improve modularity. In SOS, configurations may be syntactic trees, computed values or auxiliary entities. In MSOS, configurations always will be just syntactic trees or computed values. Any necessary auxiliary entities will be included as part of a label.

In MSOS, labels have to be used in the transitions. A label α in a transition represent every information that is associated to it, including the current environment and storage state, before and after the computation described by the transition.

Such labels are seen as category morphisms that have composition operations. The labels category is usually the product category and a notation is provided to access and change specific components independently.

In labels, the pattern `{...}` is used to represent a completely arbitrary label; a pattern such as `{env=Env,...}` allows getting information from a specific component of a label without mentioning other components; and a pattern such as `{sto=Store,sto'=Store',...}` allows changing a specific component of a label. When it is not necessary to refer to label components, they can be simply omitted.

Maude MSOS Tool (MMT) [4,5] is an executable environment to Modular SOS specifications. MMT is a formal tool implemented as a conservative extension of Full Maude [8] that compiles MSOS specifications into Rewriting Logic [4,13].

The syntax adopted by MMT is based on the Modular SOS Definition Formalism (MSDF) created by Mosses to be used in MSOS specifications. Both languages are fairly similar, in this way, those that use MSOS probably would easily understand a MSOS specification in MMT. Nonetheless, small differences exist between them due to peculiarities in the Maude parser [5,6].

MMT is very useful to the formal specification of programming languages and formal frameworks since it is possible to define the abstract syntax using BNF and give the semantics by a set of labeled transitions containing the necessary semantic components. Also, it is possible to organize the specification into modules, guaranteeing the construction of a modular specification that presents good reusability.

The SOS formalism has been described by Plotkin in [19] while its modular version has been proposed by Mosses in [17]. The implementation of Modular SOS in Maude has been discussed in [4,5].

3 Object-Oriented Action Semantics

In Object Oriented Action Semantics (OOAS) [3], an *object* encapsulates some Action Semantics features. Class constructors and several other object-based operators are offered, in extension to the standard Action Notation, providing an object-oriented way of composing specifications.

In this section we present OOAS by examples, using a simple language of commands:

```
Command ::= Identifier ":=" Expression | Command ";" Command |
            "if" Expression "then" Command 'else" Command "end-if" |
            "while" Expression "do" Command
```

The previous BNF definition establishes that commands can be conditionals, assignments, iterations or sequences. The first step using OOAS is to define the *Abstract Class* (or base class). In a simple way, the main non-terminal symbol can be *elected* as such a class. Therefore, the following class is obtained:

```
Class Command
    syntax:
        Cmd
    semantics:
        execute _ : Cmd → Action
End Class
```

Command is the abstract class. The syntax section introduces the syntactic tree Cmd. The semantics section introduces the semantic function execute, establishing a mapping from commands to actions. In OOAS, semantic functions are seen as *methods*. Now, every particular Command behavior should be defined. The tactic is to define any specific command as a *specialized class*, as follows:

```
Class While
    extending Command
    using E:Expression, C:Command
    syntax:
        Cmd ::= "while" E "do" C
    semantics:
        execute ⟦ "while" E "do" C ⟧ =
            unfolding
            | evaluate E then
            | | execute C and then unfold else complete
End Class
```

Notice that While is a subclass of command. Reusability can be achieved employing object-orientation concepts. The extending directive states a particular command behavior (Iteration). The using directive allows us to reuse existing classes (as E:Expression and C:Command). In the semantics section, While is specified, as a plain Action Semantics *action*.

OOAS has shown to be a practical alternative to modularity. OOAS Notation is simple, inspired by the notions from object-oriented programming and similar to the original Action Notation. Instantiation and extension permit the construction of libraries of programming languages concepts, improving reusability. In fact, OOAS Notation is a mixture of the original Action Notation with the Class Notation created by the object-based approach. OOAS semantics has been specified by SOS rules and it has been reported in [3].

In OOAS, the semantics description of a given programming language is a hierarchy of classes. In this regard, all defined classes belong to a pre-defined root class called *State*. Such class is the base-class for all OOAS classes and it is responsible to implement the attributes and operations that are used in those classes specifications.

In other words, the classes defined in the object-oriented formalism are sub-classes of *State*, turning visible both attributes and operations to its sub-classes. The *State* attributes are concerned with the information processed by actions, such as: *transients*, *bindings* and *storage*. Some operations are available to handle *State* attributes. Such operations are all basic actions and action combinators.

The behavior of OOAS descriptions is similar to the original Action Semantics and can be classified into facets, as defined in [15]: **basic** (deals with pure control flow); **functional** (deals with actions that process transient data); **declarative** (deals with actions that produce or receive bindings); **imperative** (deals with actions that manipulate the storage); **reflective** (deals with abstractions); **hybrid** (deals with actions from more than one facet). No operation was defined to deal with the **communicative** facet.

A complete OOAS description as well as its SOS specification can be found in [3]. In addition to it, an OOAS library of classes has been proposed in [1,12], intensely using this method. In the following section we introduce Constructive Action Semantics. A constructive approach concerned with reusability and focused on the formal specification of programming languages syntax independently.

4 Constructive Action Semantics

It is common to find semantically similar constructors in programming languages, even if these constructors have very different syntax. In this regard, it is interesting to reuse parts of the programming languages specifications that represent the same behavior. The constructive approach introduced in [18] helps with the code reuse by supporting independent definitions and using named modules to describe individual languages features.

Constructive Action Semantics [10,18] is based on the idea that each language feature is defined by a separate and independent basic abstract construct. The semantics of a complete language is achieved by translating its constructs into the basic abstract constructs. That is, the main idea of this approach consists in mapping concrete language constructs to combinations of basic abstract constructs.

Concrete constructs are related to the way programming languages implement their features while abstract constructs are concerned in representing these features using a language-independent prefix notation. For instance, a while-loop concrete syntax could be written in the following way: `while (Exp) do Cmd`; and its respective abstract syntax could be translated to: `cond-loop(Exp, Cmd)`.

Notice that the concrete construct of the while-loop is composed by an expression and a command to represent, respectively, the loop condition and the loop body. The concrete syntax shown above inhibits the parsing ambiguity among the

constructs while the corresponding abstract syntax is used just to differentiate one construct from other constructs.

Constructs might be considered as basic or derived due to the several constructs present in the developed programming languages. Basic constructs represent common features that have the same interpretations and are found in many programming languages. The constructs that have a specific behavior regarding the programming language are classified as derived constructs, they are included in few languages and specified by combining basic constructs.

The modular approach of Action Semantics has been used with the constructive approach in order to obtain a high degree of modularity in Constructive Action Semantics, how it might be observed in [10]. In this related work, the Action Semantics Definition Formalism (ASDF) has been specially designed to write Action Semantics descriptions of single language constructs using Action Semantics modules.

Now, we will see how the examples shown in section 3 may be written in the formalism presented in this section. The ASDF notation has been used to specify the necessary modules to implement an individual basic abstract construct for a while-loop command.

```
Module Cmd
requires C:Cmd
semantics execute : Cmd → Action
```

The above module defines a variable C from a sort Cmd which will be used to specify basic abstract constructs for commands. A semantic function called execute is also defined to express the implemented basic abstract construct's Action Semantics.

```
Module Cmd/While
syntax Cmd ::= cond-loop(Exp, Cmd)
requires Val ::= Boolean
semantics execute cond-loop(E, C) =
      unfolding
            evaluate E then
                  execute C and then unfold else complete
```

In the module Cmd/While we have a derived module from Cmd module. Notice that the individual basic abstract construct cond-loop(Exp, Cmd) was used to specify the semantics of a while-loop command with a boolean condition independently of its syntax. In this way, such a module could be used in any project that needs a while-loop. The specification of a whole language in Constructive Action Semantics is achieved by combining modules that implement the necessary basic individual constructs into a single module.

We have introduced the constructive approach proposed in [18], focused on its usage with Action Semantics. However, it also can be used with Modular SOS. The concrete constructs of Core ML have been translated to basic constructs as a Constructive Action Semantics case study in [10] and the formalism has been successfully validated to describe programming languages semantics syntax independently. Motivated by these results, in section 6 we will present how constructive semantics can be applied to Object-Oriented Action Semantics using the tool described in next section.

5 Maude Object-Oriented Action Tool

In this section we will present MOOAT - a tool that provides an executable environment to the formal specification of programming languages using Object-Oriented Action Semantics. It has been developed using MMT [4] to implement the Modular SOS of Action Notation, introduced by Mosses in [16], and also using Maude [6] to implement the ideas proposed by [3].

5.1 Notation

A formal specification in Object-Oriented Action Semantics is a finite set of classes. Such classes create the necessary hierarchy to represent the formal specification of a language or library. The relationship among these classes is defined according to the instantiated objects, as well as the position where classes are found in the specified hierarchy.

The described tool is a conservative extension of Full Maude [8] and MMT [5]. In this way, programming languages can be described in Object-Oriented Action Semantics by MOOAT, while Full Maude and MMT still might be used in connection with Rewriting Logic [13] and Modular SOS [17], respectively.

Nevertheless, the environment has the same limitations of Maude and MMT, as explained in [4,6,11]. We shall cite *pre-regularity check* [4,11] and *ad-hoc overloading* [6,11] as the limitations frequently found in the MOOAT development process. Due to these limitations MOOAT syntax has some differences from OOAS syntax, yet both are very similar.

MOOAT notation is given using BNF as follows:

(1) *ClassModule* ::= "class" *ClassName* "is" ⟨ *ClassExtends* ⟩* ⟨ *ClassDefinition* ⟩* "endclass"

(2) *ClassExtends* ::= "extends" *ClassName* ⟨ "," *ClassName* ⟩* "."

(3) *ClassDefinition* ::= ⟨ *SyntacticPart* ⟩* ⟨ *SemanticPart* ⟩*

The class structure is defined in the rules (1) to (3). A class begins with the directive class and ends with endclass. The body class is composed basically by the declaration of base classes and by the class definition. Notice that more than one class can be specified as a base class.

If the tool notation is compared to the OOAS notation introduced in [3], some differences might be found. Amongst which we point out the changing of extending to extends and the exclusion of the directive using. The former was done for the reason that the directive extending already exists in Maude and if we redefined it we would have the *pre-regularity* problem discussed in [4]. The latter was done for the fact that now object declarations are done automatically in the methods definition.

(4) *SyntacticPart* ::= *SyntacticSort* "." | *SyntacticSort* "::=" *syntax-tree* "."

(5) *SemanticPart* ::= ⟨ *SemanticFunctions* ⟩* ⟨ *SemanticEquations* ⟩*

(6) *SemanticFunctions* ::= ⟨ *SemanticFunction* ⟩+

(7) *SemanticFunction* ::=
"absmethod" *FunctionName Tokens* "->" "Action" "." |
"absmethod" *FunctionName Tokens* "->" *Data* "."

(8) *Tokens* ::= *TokenName* ⟨ "," *Tokens*⟩*

(9) *SemanticEquations* ::= ⟨ *SemanticEquation*⟩+

(10) *SemanticEquations* ::=
"method" *FunctionName syntax-tree-with-objects* "=" *Action* "."
"method" *FunctionName syntax-tree-with-objects* "=" *Data* "."

(11) *ObjectDeclaration* ::= *Identifier* ":" *SyntacticSort*

In the rules from (4) to (11) we have the definition of the body class. Like in the original Object-Oriented Action Semantics, the body class is composed by two parts: syntax and semantics. The syntactic part follows the BNF introduced by MMT. The semantic part may be composed by semantic functions, which will be called as abstract methods, and semantic equations, which will be called as simply methods.

An abstract method works as a signature of a method that will be specified in a class or in a sub-class and may result in an action or in a data sort. A method implements the semantics of a programming language concept using actions and action combinators. Methods must be used to give the semantics of the abstract syntax defined in the class. When such abstract syntax is used in the method declaration, syntactic sorts must be changed by object declarations.

The automatic object creation concept is introduced, since the syntactic sorts are changed by identifiers that represent these syntactic sorts in the methods implementations. These objects will be used to perform encapsulated actions and to give the desired semantic meaning.

(12) *Action* ::= *Action* "or" *Action* | "fail" | "commit" | *Action* "and" *Action* |
"complete" | "indivisibly" *Action* | *Action* "and then" *Action* |
Action "trap" *Action* | "escape" | "unfolding" *Action* | "unfold" |
"diverge" | "give" *Yielder* | "regive" | "choose" *Yielder* |
"check" *Yielder* | *Action* "then" *Action* | "escape with" *Yielder* |
"bind" *Yielder* "to" *Yielder* | "rebind" | "unbind" *Yielder* |
"produce" *Yielder* | "furthermore" *Action* | *Action* "moreover" *Action* |
Action "hence" *Action* | *Action* "before" *Action* |
"store" *Yielder* "in" *Yielder* | "unstore" *Yielder* | "reserve" *Yielder* |
"unreserve" *Yielder* | "enact" *Yielder* |
"indirectly bind" *Yielder* "to" *Yielder* | "indirectly produce" *Yielder* |
"redirect" *Yielder* "to" *Yielder* | "undirect" *Yielder* |
"recursively bind" *Yielder* "to" *Yielder* |
Action "else" *Action* | "allocate" *Yielder*

(13) *Yielder* ::= *Data* | "a" *Yielder* | "the" *Yielder* | "nothing" |
"the" *DataSort* "yielded by" *Yielder* | "it" | "them" |
"given" *DataSort* | "given" *DataSort* # *Int* |
"current bindings" | "the" *DataSort* "bound to" *Yielder* |
Yielder "receiving" *Yielder* |
"current storage" | "the" *DataSort* "stored in" *Yielder* |
"application" *Yielder* "to" *Yielder* | "closure" *Yielder* |
"encapsulate" *Action* | "indirect closure" *Yielder*

(14) *Data* ::= *Datum* | *DataSort*

(15) *Datum* ::= "none" | "unknown" | "uninitialized" | *Abstraction* |
"<" *Int* ">" | "<" *Boolean* ">" | "<" *Token* ">" | "<" *Cell* ">" |
"<" *Transients* ">" | "<" *Bindings* ">" | "<" *Storage* ">" |

(16) *DataSort* ::= "integer" | "truth-value" | "token" | "cell" | "abstraction" | "value"

Rules (12) to (16) define the Action Notation to be used in the methods body. A coercion function (< >) must be used with the implemented data due to the *ad-hoc overloading* problem cited in [6]. To exemplify the use of MOOAT we will see how the classes shown in section 3 are written:

```
(class Command is
 Cmd .
 absmethod execute Cmd -> Action .
endclass)
```

In the above example we have the abstract class `Cmd` which just defines the syntactic sort `Cmd` and the abstract method `execute` that maps a command to an action.

```
(class While is
 extends Command .
 Cmd ::= while Exp do Cmd .
 method execute (while E:Exp do C:Cmd) = unfolding
    ((evaluate E:Exp) then
    ((execute C:Cmd and then unfold) else complete)) .
endclass)
```

The `While` class is the sub-class of `Cmd` and a while-loop is implemented in it. The syntactic sort `Cmd` is overloaded with the command syntax and the semantics is given in the overloaded method `execute`. The objects `E:Exp` and `C:Cmd` were used to give the while semantics, that is, the command in the object `C:Cmd` is executed until the expression in the object `E:Exp` is true.

Notice that, previously an object was declared by an identifier and its respective class in the directive using. However, now it is directly declared by an identifier and its respective syntactic sort in the method definition. In this way we have introduced the automatic object creation concept.

In the next section we will see how the Modular SOS for Action Notation was implemented using MMT and how the Classes Notation was specified in Maude.

5.2 Implementation

MOOAT has been built as a conservative extension of Full Maude and MMT. Action Notation is given by MSDF modules which implement the available actions trough Modular SOS transitions. Some Maude operations and equations have been implemented to support the Classes Notation proposed by the object-oriented style of Object-Oriented Action Semantics.

This implementation was possible since the modularity in Modular SOS specifications is in the labels used by transitions. Therefore, using MMT was possible to implement the Action Notation modules defined in [16] and the Classes Notation which supports specifications in the OOAS style.

First of all, we will show how Action Notation transitions were implemented as well as Data Notation. After that we will elucidate how the `LOOP-MODE` [4,6,8] was changed to accept OOAS classes and translate them into Maude system modules, the same technique employed in object-oriented modules of Full Maude [8] and MSDF modules of MMT [4]. Then we will mentionate how *State* class, detailed in

section 3, is treated.

Data Notation has been specified by reusing datatypes that are provided by both Maude and MMT. For instance, the sorts `Int` and `Boolean` defined by MMT were used to represent integers and truth-values respectively. However, some coercion functions were needed to avoid the *ad-hoc overloading* problem in these datatypes specification. It can be observed in the rule (15) in section 5.1.

The abstract syntax of Action Notation has been specified in MMT using datatype declarations due to the extended-BNF syntax provided by it. Mixfix operations might be defined in order to specify the abstract syntax of the language or formalism that is being defined, as well as it is shown in the following module:

```
(msos BasicSyntax is
 see BasicData .

 Action ::= Action or Action | fail | commit |
            Action and Action | complete |
            indivisibly Action |
            Action and‘then Action |
            Action trap Action | escape |
            unfolding Action | unfold | diverge .

 Yielder ::= the DataSort yielded‘by Yielder |
            nothing | Data .
sosm)
```

Basic facet abstract syntax is being implemented by the MSDF module `BasicSyntax`. Notice that the sorts `Action` and `Yielder` were extended to defined the *actions* and *yielders* available in basic facet, and also that data components are regarded as already evaluated.

Action performance and yielder evaluation may compute values. These values can be defined algebraically in MMT as sorts, operations and predicates, as well as it is shown in the following module:

```
(msos FunctionalOutcomes is
 see BasicOutcomes .
 see FunctionalData .

 Completed .

 Terminated ::= Completed .

 Completed ::= completed .
 Completed ::= gave (Data) .

 gave (none) : Action --> completed .
sosm)
```

Notice that the MSDF modules `BasicOutcomes` and `FunctionalData` were included in `FunctionalOutcomes`. In such module the `Terminated` sort is redefined to accept the values introduced by the sort `Completed`, which are: `completed`, to determine that an action is completed, and `gave (Data)`, to specify a transient data production. When no data is given then the action is simply completed.

```
(msos BasicConfigurations is
 see BasicSyntax .
 see BasicOutcomes .

 Action ::= Terminated | Action @ Action .
sosm)
```

In the MSDF module `BasicConfigurations` we have introduced that configurations for non-distributed action performance are always the same. The sort

`Terminated` is related to the results produced by actions and it may depends on the facet. An auxiliary construct (`Action1 @ Action2`) were defined to be used just by the basic facet and with the `unfolding` construct.

Each facet of Action Notation usually requires the implementation of labels to treat with the facet's components. For instance, functional facet treats with transient data, in this way its label have to carry the current transient data produced by the current actions.

```
(msos FunctionalLabels is
 see BasicLabels .
 see FunctionalData .

 Label = {data : Transients, data' : Transients, ...} .
sosm)
```

The label for the functional facet were implemented in the MSDF module `FuncionalLabels` as a read-write label. Its indexes `data` and `data'` carry the current transient data since they are being represented by the sort `Transients`, which is in fact a map from a positive number to a simple datum. Now we will see some of the implemented transitions needed to understand the MOOAT development process. Basically, we have three main kinds of transitions.

Those transitions that neither change or use the labels' components. Such as shown in the following example which defines an interleaved performance of "`Action1 and Action2`".

```
           Action1 -{...}-> Action'1
 -- ------------------------------------------------------
 Action1 and Action2 : Action -{...}-> Action'1 and Action2 .

           Action2 -{...}-> Action'2
 -- ------------------------------------------------------
 Action1 and Action2 : Action -{...}-> Action1 and Action'2 .
```

Those transitions that just use the labels' components. In the following example we have a rule that takes the transient data carried by the component `data` and turn them available for the next actions.

```
 regive : Action -{data = Transients, data' = Transients,-}-> gave (< Transients >) .
```

Those transitions that either change and use the labels' components. Now we have an example were the label component has been changed beyond it has been used. The transient data `Data1` and `Data2` are processed and the current transient data, represented by `Transients`, are changed to have them as the new transient data, represent by `Transients''`.

```
       Transients' := (1 |-> Data1) / Transients,
       Transients'' := (2 |-> Data2) / Transients'
 -- ------------------------------------------------------------------------------------
 gave (Data1) and gave (Data2) : Action -{data = Transients, data' = Transients'',-}->
    gave (concatenation(Data1, Data2)) .
```

In MOOAT, the *State* class has been represented partially by a system module called `STATE` and partially by the Classes Notation that will be detailed since here.

The *State* class of MOOAT have five attributes instead of only three as we mentioned in section 3. Such attributes are used in the specification of programming languages and have been represented by labels' indexes implemented by the Action Notation described above.

Since we are describing an implementation using MMT, those attributes must

be initialized as the initial configuration needed to compute an action. Such configuration has been represented by the command `compute` which has been defined in `STATE` module receiving an action as parameter. This command must be used with Full Maude's command `rewrite` in order to init the MRS [5,14] configuration implemented by MMT and used by MOOAT to create the executable environment that computes an action.

State attributes are classified and initialized according to the following table:

Attribute	Data Flow	Initial Value
commitment	a boolean value for *commit*	false
unfolding	an action	fail
data	the transient data	empty map (`Int` → `Datum`)
bindings	the bindings	empty map (`Token` → `Data`)
storage	the storage	empty map (`Cell` → `Data`)

Notice that in MOOAT, the operations that act over the main class are the transitions implemented by each one of the implemented facets. In this way, the actions provided by the Object-Oriented Action Semantics of MOOAT are quite similar to the actions provided by the Modular SOS for Action Notation described in [16], as it was proposed as a further work in [3].

The technique employed in the development of the Classes Notation is similar to the used on the construction of object-oriented modules in Full Maude [8] and on the definition of MSDF modules in MMT [5]. That is, a specific syntax is created and when it is used its constructions are translated to a code that Maude is able to interpret.

MOOAT classes are composed basically by three parts: extends part, syntactic part and semantics part. The use of these parts might be optional or sequential. In other words, they may be not used, used just once or more than once. On the other hand, a class must have at least one of these three parts. It is not allowed the definition of an empty class.

Classes in MOOAT were implemented as alternative MSDF modules where just the three parts mentioned before are accepted in a class definition. The definition of a sub-class is simply translated to lines that use the Maude's directive `including`. In the syntactic part we have reused the definition of syntax trees in the BNF style implemented by MMT; these trees and syntactic sorts are translated, respectively, to operations, sorts and subsorts of Maude.

The methods in the semantic part are translated to a set of equations supported by Maude system modules. While the objects defined automatically are treated as meta-variables from Maude in those equations set. As well as in Object-Oriented Action Semantics, in MOOAT every class is a direct sub-class of *State*. In this way, when we create a new class the main class is automatically included to turn available the defined Action Notation. For the fact that a class is converted to a system module, the *State* class was implemented directly as a system module. For this reason it is possible to add it in every converted class.

A methods environment is created by the inclusion of the converted MOOAT classes into the Maude's module database. The MOOAT root class provides the

necessary operations to change its attributes and the environment for the methods definition that is still being read just by the operations previously defined. However, when a new class is added its respective methods are also added. The inclusion of the converted modules into the Maude's database were implemented as it is being shown in the following piece of code.

```
rl < O : X@Database | db : DB, input : ('class_is_endclass[T, T']), step-flag : B,
                      output : nil, default : MN, Atts > =>
   < O : X@Database | db : mooat-proc-unit('class_is_endclass[T, T'], step-flag(B), DB),
                      input : nilTermList, step-flag : B,
                      output : ('Introduced 'OOAS 'class
                               header2QidList(parseHeader(T)) '\n),
                      default : parseHeader(T), Atts > .
```

Notice that the inclusion of a class into the database is initialized by the equation `mooat-proc-unit` which implements the necessary rules to the parsing of a MOOAT class. The process is finished when the control is passed to the equation `mooat-eval-preunit` since it really converts the class into a system module and after that it is added to the database present in the Maude system.

In this section we have presented some aspects from the tool's implementation. Such implementation consists on the adaptation of the Action Notation present in the original Object-Oriented Action Semantics SOS rules to MSOS rules introduced by Mosses and supported by MMT. Moreover, the Classes Notation were implemented using the powerful system provided by Maude. The implementation aspects were summarized in this paper to introduce the use of MOOAT and also to the comprehension of its implementation. For more details about it we shall indicate [11] as the main source.

6 Constructive Object-Oriented Action Semantics

In sections 3 and 4 we have presented Object-Oriented Action Semantics, an approach for language definition using Action Semantics with object-oriented concepts, and Constructive Action Semantics, a constructive approach that also can be used with Action Semantics. The former is based on the modularity in Action Semantics by splitting descriptions into classes, the latter is based on the idea of a collection of basic abstract constructs that may be used in different programming languages projects.

We also have introduced a tool that supports the definition of programming languages or libraries in Object-Oriented Action Semantics. Hence, we will show that is possible to combine the formalisms described in sections 3 and 4, in order to obtain Constructive Object-Oriented Action Semantics as a MOOAT case study.

To be more specific, using MOOAT we will demonstrate that the modularity aspects observed in the object-oriented approach might be improved by adding the constructive ideas into it. Furthermore, an approach with good modularity, easy readable and that helps on the programming languages specifications syntax independently can be achieved.

Again we will use examples related to commands in order to introduce the proposed ideas.

```
(class Cmd is
 Cmd .
 absmethod execute Cmd -> Action .
endclass)
```

The class `Cmd` works as an abstract class since it just introduces the syntactic sort `Cmd`, which will be used in the definition of the commands constructs that will be implemented, and the abstract method `execute` to work as a semantic function. Notice that the presented class is exactly the same if compared to its respective class in section 5.

```
(class Cmd/While is
 extends Cmd .
 Cmd ::= cmd-while (Exp, Cmd) .
 method execute (cmd-while(E:Exp, C:Cmd)) = unfolding
    (evaluate E:Exp then
    ((execute C:Cmd and then unfold) else complete)) .
endclass)
```

In the `Cmd/While` class we have a specialized class of `Cmd` that implements the construct of a while-loop, `cmd-while (Exp, Cmd)`. Notice that the main difference between Object-Oriented Action Semantics and Constructive Object-Oriented Action semantics is in overloading the syntactic sort with a basic abstract construct instead of using the language concrete construct. This basic abstract construct is also used when the method `execute` is overloaded in order to give its semantics.

We have created a set of classes to Constructive Object-Oriented Action Semantics in [11]. Those classes were specified in the same style as `Cmd` and `Cmd/While` to deal with expressions, commands, declarations, values and programs.

For expressions constructs we have `Exp` class and its respective sub-classes that deal with arithmetical and logical values and expressions. As well as in the examples mentioned in this section, the class `Cmd` and its sub-classes are related to commands constructs. The class `Dec` introduces declarations constructs which will be used in its specialized classes of declarations. A class to specify a program construct was also implemented and it was called `Prog`.

Now we will see how a programming language would be defined using the classes mentioned above. For this reason, we present the specification of a toy language called μ-Pascal. This language is fairly similar to Pascal language; μ-Pascal is an imperative programming language containing basic commands and expressions. Its respective syntax was defined using BNF in [11].

```
(class Micro-Pascal is
 extends Exp/Val, Exp/Val-Id .
 extends Exp/Sum, Exp/Sub, Exp/Prod .
 extends Exp/True, Exp/False, Exp/LessThan, Exp/Equality .
 extends Cmd/Assignment, Cmd/Repeat, Cmd/While .
 extends Cmd/Sequence, Cmd/Cond .
 extends Dec/Variable, Dec/DecSeq .
 extends Prog .
endclass)
```

The concrete constructs of μ-Pascal are represented by each class specified in the `extends`' lines. Such classes implement the basic abstract constructs needed by the specified programming language. The reason why we do not have reference to the abstract classes is that they are already referenced by their own sub-classes.

In these regard, the class `Micro-Pascal` defines the formal semantics of μ-Pascal using Constructive Object-Oriented Action Semantics and just extending the spe-

cialized classes that were designed to give the semantics of specific features. The definition of the programming language was achieved independently of its syntax for the fact that those features were implemented by basic abstract constructs as well as it has been proposed in [18].

Like in Constructive Action Semantics, in Constructive Object-Oriented Action Semantics we have to translate the concrete syntax of the language to its respective combination of basic abstract constructs. This was done in the example above and any language that have similar features to μ-Pascal could be specified in this way. The whole Constructive Object-Oriented Action Semantics can be found in [11].

7 Final Remarks and Future Work

Action Semantics and Object-Orientation are important topics to formal specification. OOAS offers the alternative of combining both subjects. However, no executable environment for writing and executing OOAS specifications had been provided yet. MOOAT has been developed considering this demand and based on an earlier Action Semantics tool called MAT, which was developed using an earlier MSOS interpreter developed in Maude.

In this paper we have presented MOOAT, the first implementation of Object-Oriented Action Semantics. Our first idea consisted on developing an isolated OOAS tool from scratch. However, it seemed impracticable knowing the existence of MAT. The better choice, in our view, was creating an extension of MMT. The idea was to aggregate the object-oriented apparatus to a brand new tool based on MAT and developed using MMT. In MOOAT, the user can create OOAS specifications, and perform the necessary tests, inside the standard Maude environment. This is interesting, considering that the user can create Maude, MMT, and MOOAT specifications using the same tool.

This implementation has contributed to the formal specification of OOAS since a Modular SOS definition has been provided to its Action Notation using MMT and the Maude system has been used to define its Classes Notation. It means that the previous specification of OOAS using SOS has been rewritten using MSOS and implemented using the MSOS language provided by MMT. Also, that a language to specify OOAS classes has been built in Maude.

As far as we are concerned, now it is easier to update and insert new features to OOAS due to the MSOS specification of MOOAT. Even being written using MMT, MOOAT is the update of OOAS since its formal specification has been translated from SOS to MSOS. Also, the fact of using Maude and MMT to develop MOOAT provided the first executable environment for OOAS specifications. Furthermore, MOOAT notation covers a rich set of actions and action combinators, which includes the complete OOAS notation.

In addition to the implementation, we have combined Constructive Action Semantics (CAS) with Object-Oriented Action Semantics (OOAS) in order to achieve *Constructive Object-Oriented Action Semantics* (COOAS) as a case study of MOOAT. This constructive semantics represents a novel view to the Object-

Oriented Action Semantics system. This is the first time that both formalisms are combined.

Moreover, the introduction of constructs in OOAS has contributed to improve the modularity aspects observed in the object-oriented approach. Since such combination is capable to describe syntax-independent specifications of programming languages. This happens due to the fact that we see the constructive approach as an idea that helps with code reuse when adopted into existent formalisms, as well as it happened with MSOS, AS and now OOAS. Notice that COOAS has been shown using MOOAT though it could be used as a specification approach independently of the tool usage.

In [11], MOOAT development is described in more details as well as other case studies are presented. Besides COOAS we also present, in [11], the implementation of LFLv2 [12], a new version for the Language Features Library (LFL) [1], and how to specify a simple imperative language, called μ-Pascal, using OOAS, LFLv2 and COOAS in MOOAT. As future work we would implement the **communicative** facet of Action Notation. Since COOAS is a rigorous analisys subject, we will present it in more details as well as a careful comparison between LFLv2 and COOAS in the future.

Acknowledgement

We would like to thank MMT authors, Christiano Braga and Fabricio Chalub, for the interest in this work and for the helpful support while we were using their tool to develop MOOAT. Also, we would like to thank the anonymous referees for the comments that helped to improve this paper. We are most greatful with LSFA 2007 organizers for funding the presentation of this research at such nice event.

References

[1] Araújo, M. and M. A. Musicante, *Lfl: A library of generic classes for object-oriented action semantics*, in: *XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), 11-12 November 2004, Arica, Chile* (2004), pp. 39–47.

[2] Braga, C., 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*, Lecture Notes in Computer Science **1816** (2000), pp. 407–421.

[3] Carvilhe, C. and M. A. Musicante, *Object-oriented action semantics specifications*, Journal of Universal Computer Science **9** (2003), pp. 910–934.

[4] Chalub, F. and C. Braga, *Maude msos tool*, Technical report, Universidade Federal Fluminense (2005), http://maude-msos-tool.sourceforge.net/mmt-manual.pdf.

[5] Chalub, F. and C. Braga, *Maude msos tool*, in: G. Denker and C. Talcott, editors, *Proceedings of 6th International Workshop on Rewriting Logic and its Applications*, WRLA (2006), to appear in Electronic Notes in Theoretical Computer Science.

[6] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, *Maude manual (version 2.3)* (2007), http://maude.cs.uiuc.edu/maude2-manual/html/.

[7] Doh, K.-G. and P. D. Mosses, *Composing programming languages by combining action-semantics modules*, in: M. van den Brand and D. Parigot, editors, *LDTA'01*, ENTCS **44.2** (2001).

[8] Durán, F. and J. Meseguer, *The Maude specification of Full Maude*, Technical report, SRI International (1999).

[9] Gayo, J. E. L., *Reusable semantic specifications of programming languages*, in: *SBLP 2002 - VI Brazilian Symposium on Programming Languages*, 2002.

[10] Iversen, J., "Formalisms and tools supporting Constructive Action Semantics," Ph.D. thesis, BRICS International PhD School (2005).

[11] Maidl, A. M., "A Maude Implementation of Object-Oriented Action Semantics," Master's thesis, Universidade Federal do Paraná (2007), http://www.inf.ufpr.br/murbach/mooat (In portuguese).

[12] Maidl, A. M., C. Carvilhe and M. A. Musicante, *Using visitor patterns in object-oriented action semantics*, in: *SBLP 2007 - XI Brazilian Symposium on Programming Languages*, 2007.

[13] Martí-Oliet, N. and J. Meseguer, *Rewriting logic as a logical and semantic framework*, Technical report, SRI International (1993).

[14] Meseguer, J. and C. Braga, *Modular rewriting semantics of programming languages*, in: C. Rattray, S. Maharaj and C. Shankland, editors, *In Algebraic Methodology and Software Technology: proceedings of the 10th International Conference, AMAST 2004*, LNCS **3116** (2004), pp. 364–378, iSSN 0302-9743, ISBN 3-540-22381-9.

[15] Mosses, P. D., "Action Semantics," Cambridge Tracts in Theoretical Computer Science **26**, Cambridge University Press, 1992.

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

[17] Mosses, P. D., *Modular structural operational semantics*, J. Logic and Algebraic Programming **60–61** (2004), pp. 195–228, special issue on SOS.

[18] Mosses, P. D., *A constructive approach to language definition*, Journal of Universal Computer Science **11** (2005), pp. 1117–1134.

[19] Plotkin, G. D., *A structural approach to operational semantics*, J. Logic and Algebraic Programming **60–61** (2004), pp. 17–139, special issue on SOS.

[20] van den Brand, M., J. Iversen and P. D. Mosses, *An action environment*, Sci. Comput. Program. **61** (2006), pp. 245–264.

[21] Watt, D. A. and M. Thomas, "Programming language syntax and semantics," Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1991.