Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 175 (2007) 3–17
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Modular Structural Operational Semantics with Strategies*

Christiano Braga[1] Alberto Verdejo[2]

Facultad de Informática
Universidad Complutense de Madrid

Abstract

Strategies are a powerful mechanism to control rule application in rule-based systems. For instance, different transition relations can be defined and then combined by means of strategies, giving rise to an effective tool to define the semantics of programming languages. We have endowed the Maude MSOS Tool (MMT), an executable environment for modular structural operational semantics, with the possibility of defining strategies over its transition rules, by combining MMT with the Maude strategy language interpreter prototype. The combination was possible due to Maude's reflective capabilities. One possible use of MMT with strategies is to execute Ordered SOS specifications. We show how a particular form of strategy can be defined to represent an OSOS order and therefore execute, for instance, SOS specifications with negative premises. In this context, we also discuss how two known techniques for the representation of negative premises in OSOS become simplified in our setting.

Keywords: Modular SOS, Strategies, Ordered SOS, Negative Premises

1 Introduction

Strategies are a powerful mechanism for the specification of programming languages and systems. A strategy language describes how rules should be applied in a given rule-based specification by means of a combination of basic strategies. In Maude's strategy language [7], our language of choice, a basic strategy specifies that a rule, denoted by its label, can be applied possibly with a given substitution and using given strategies to solve its premises, if any. Strategy combinators are tests, conditionals, decomposition (i.e. a strategy applied to subterms), and search. Recursive strategies can also be defined. Non-trivial examples where the Maude's strategy language has been used to implement structural operational semantics are the Eden

* Research supported by MCyT Spanish project MIDAS (TIC2003-0100) and Ramón y Cajal program.

[1] Email: `cbraga@fdi.ucm.es` (On leave from Universidade Federal Fluminense, Brasil.)

[2] Email: `alberto@sip.ucm.es`

doi:10.1016/j.entcs.2006.10.024

language, that has several transition relations which can be specified and combined by means of strategies [5], and the ambient calculus, where strategies [14] are used to control communication, replication and termination.

We have endowed Modular SOS (MSOS) [10] specifications with strategies, by putting together the Maude MSOS Tool (MMT) [2], an executable environment for MSOS, with Maude's strategy language (MSL) [7]. The combined tool, named MMT+MSL, is implemented as a conservative extension of Maude's extensible module algebra implemented in Full Maude [4]. To illustrate the usefulness of our proposal, we show how Ordered SOS (OSOS) [15] specifications can be directly represented in MMT+MSL, where the transition rules are the same and the order is represented as a strategy. Then, using this representation, negative premises become executable in MMT+MSL by the application of the techniques given in [15], and yet, simplified. As a concrete example, we extend the modular SOS specification of CCS with priorities.

This paper is organized as follows. Section 2 overviews Maude's strategy language, exemplifies the syntax for specifications accepted by MMT+MSL, using CCS as example, and the implementation MMT+MSL in Full Maude. Section 3 explains how Ordered SOS specifications can be represented as specifications in MMT+MSL. Section 4 briefly recalls how negative premises can be represented in OSOS. Section 5 extends the CCS specification in Section 2 with a priority operator. Section 6 concludes the paper with our final remarks.

2 MMT+MSL

2.1 Maude's Strategy Language

Rewrite rules in rewriting logic need be neither confluent nor terminating. This theoretical generality requires some control when the specifications become executable, because it must be ensured that the rewriting process does not go in undesired directions. Maude's strategy language can be used to control how rules are applied to rewrite a term [7]. The simplest strategies are the constants '`idle`', which always succeeds by doing nothing, and '`fail`', which always fails. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and with the possibility of providing a substitution for the variables in the rule. In this case a rule is applied *anywhere* in the term where it matches satisfying its condition. When the rule being applied is a conditional rule with rewrites in the conditions, the strategy language allows to control how the rewrite conditions are solved by means of search expressions. An operation '`top`' to restrict the application of a rule just to the *top* of the term is also provided. Basic strategies are then combined so that strategies are applied to execution paths. Some strategy combinators are the typical regular expression constructions: concatenation ('`;`'), union ('`|`'), and iteration ('`*`' for 0 or more iterations, '`+`' for 1 or more, and '`!`' for a 'repeat until the end' iteration). Another strategy combinator is a typical 'if-then-else', but generalized so that the first argument is also a strategy. By using this combinator, we can define many other useful strategy combinators as derived operations: for example a binary '`orelse`' combinator that applies the second argument strategy only if the first fails, and a unary '`not`' combinator that fails when its argument is successful and vice versa. The language also provides a '`matchrew`' combinator that allows a term to be split in subterms, and specifies how these subterms have to be rewritten. An extended '`matchrew`', '`xmatchrew`', is also provided where rewriting modulo axioms associativity, commutativity, identity and idempotency is considered, when declared. Recursion is also possible by giving a name to a strategy expression and using this name in the strategy expression itself or in other related strategies.

Using the Maude metalevel, we have implemented a prototype of the strategy language as an extension of Full Maude [7]. Currently the language is being integrated in the Maude system.

2.2 CCS in MMT+MSL

Modular SOS is a variant of SOS that allows for specifications to be made modular by structuring the labels in the transition rules as extensible records. Semantic rules for a given constructor use certain indices from the record structure, so that newly added rules could range over (existing or) new indices, thus allowing that existing rules are not changed when new semantic entities are required. Therefore, semantic rules may be declared *once and for all.* For instance, rules for a functional fragment may access an environment from the label structure while rules for an imperative fragment may access the memory component.

MMT [1] is an executable environment for MSOS and was implemented as a formal tool in the precise sense presented in [3], that is, as a realization of a semantics preserving mapping between Modular SOS and rewriting logic. The modular SOS definition formalism is the specification supported by MMT. It allows MSOS specifications to be written in a quite succinct syntax that includes: support for grammar specification in BNF like syntax, implicit module inclusion, "type declaration" as alias for instantiated parameterized built-in types, automatic derived set and list declarations for each explicitly declared set in the BNF or aliasing sections, automatic variable declarations by appending "primes" and numbers on the set names, and explicit label structure declaration.

Let us discuss now how CCS can be specified and executed in MSDF. We also present a strategy that solves the rules premises in depth-first search. Concrete labels and process identifiers are declared to test the execution of our specification. No runs are shown in the paper but the tool and this example can be downloaded from http://maude-msos-tool.sf.net/mmt+msl/.

We follow the constructive approach for semantic descriptions proposed by Mosses in [11] and thus present each construct as a separate module in MSDF.

The module '`LABEL`' declares a set for action labels. The module '`ACTION`' declares the set '`Action`' that includes labels and the (unobservable) '`tau`' action.

```
(msos LABEL is
  Label .
  Label ::= ~ Label | a | b | c .
```

```
(msos ACTION is
  Action .
  Action ::= Label | tau .
sosm)                                        sosm)
```

The module ‘`PROCESS`’ declares the set of processes (‘`Process`’) and the idle process (‘`0`’).

```
(msos PROCESS is
  Process .
  Process ::= 0 .
sosm)
```

The MSOS label structure used in the modules below has an index ‘`trace’`’ representing the process trace. The quote in ‘`trace’`’ has a meaning: in MSOS terminology it is a write-only index, that is, it can only be updated.

Transition rules in MSDF represent quite directly standard mathematical notation for transition rules. A few explanations may clarify, however, the notation for label patterns. Labels may have ellipsis (`...`) or a dash (`-`) to represent all the indices in a label not explicitly mentioned. When ellipsis are used it means that the part of the label it refers to may be changed in a transition. The dash is used otherwise. When they occur more than once in the same rule, they refer to the same subset of the indices. Metavariables, such as `X1` and `X2`, may also be used to refer to a subset of the indices of a label and are used to distinguish between two sets of indices in the same rule.

The module ‘`PREFIX`’ declares an action prefix (‘`;`’) that adds an action to the trace. Note that the set ‘`Action*`’, for a possibly empty set of actions, has not been declared explicitly. It was automatically derived by the declaration of the set ‘`Action`’ in module ‘`ACTION`’, which was automatically imported by ‘`PREFIX`’.

```
(msos PREFIX is
  Process ::= Action ; Process [prec 20] .
  Label = {trace' : Action*, ...} .

  [prefix] (Action ; Process) : Process -{trace' = Action,-}-> Process .
sosm)
```

Summation (‘`+`’) means simply to choose one of the processes to evolve. Note that only one rule is needed since the operator is declared as commutative, with keyword ‘`comm`’ in the BNF declaration.

```
(msos SUMMATION is
  Process ::= Process + Process [assoc comm prec 30] .
  Label = {trace' : Action*, ...} .

           Process1 -{...}-> Process1'
  [sum] -- ---------------------------------------------
        (Process1 + Process2) : Process -{...}-> Process1' .
sosm)
```

Parallelism (‘`||`’) allows one process to evolve or both if they synchronize, that is, one performs ‘`Action`’ and the other ‘`~Action`’. The CCS semantics does not specify how synchronization behaves in the presence of side-effects. In our semantics no side-effects may be produced while synchronizing. (This is the semantics for synchronization in Reppy’s λ_{cv}, for instance, whose MSOS semantics is given in [9].)

```
(msos PARALLELISM is see ACTION .
  Process ::= Process || Process [assoc comm prec 25] .
  Label = {trace' : Action*, ...} .

           Process1 -{...}-> Process1'
  [par1] -- ------------------------------------------------------------
         (Process1 || Process2) : Process -{...}-> Process1' || Process2 .

           Process1 -{trace' = Action, -}-> Process1' ,
           Process2 -{trace' = ~ Action, -}-> Process2'
  [par2] -- ------------------------------------------------------------
         (Process1 || Process2) : Process -{trace' = tau, -}->
                                                  Process1' || Process2' .
sosm)
```

Relabeling (‘`rel`’) substitutes a performed action label by another one.

```
(msos RELABELLING is see ACTION .
  Process ::= rel (Process, Label, Label) [prec 20] .
  Label = {trace' : Action*, ...} .

           Process1 -{trace' = Action1, ...}-> Process1'
  [rel1] -- -----------------------------------------------------------
         (rel (Process1, Action2, Action1)) : Process
                                   -{trace' = Action2, ...}-> Process1' .

           Process1 -{trace' = ~ Action1, ...}-> Process1'
  [rel2] -- -----------------------------------------------------------
         (rel (Process1, Action2, Action1)) : Process
                                 -{trace' = ~ Action2, ...}-> Process1' .

           Process1 -{trace' = Action3, ...}-> Process1' ,
           Action3 =/= Action1,
           Action3 =/= ~ Action1
  [rel3] -- -----------------------------------------------------------
         (rel (Process1, Action2, Action1)) : Process
                                   -{trace' = Action3, ...}-> Process1' .
sosm)
```

Finally, restriction (‘\’) of an action means that a process is allowed to evolve if it does not signal the given action or its negation.

```
(msos RESTRICTION is see ACTION .
  Process ::= Process \ Label [prec 25] .
  Label = {trace' : Action*, ...} .

           Process1 -{trace' = Label2, ...}-> Process1' ,
           Label2 =/= Label1,
           Label2 =/= ~ Label1
  [res] -- ------------------------------------------------------------
        (Process1 \ Label1) : Process
                          -{trace' = Label2, ...}-> Process1' \ Label1 .
sosm)
```

MSDF is implemented in MMT as a conservative extension of Full Maude. Therefore functional modules (for equational specifications) and system modules (for equational and rule-based specifications) in Maude may be used together with MSDF specifications. Double-negation of labels are specified as an equation in the functional module '`LABEL-CONGRUENCE`' which is then combined with the above MSDF modules in the '`CCS`' system module.

```
(fmod LABEL-CONGRUENCE is
   inc LABEL .
   eq ~ ~ Label:Label = Label:Label .
 endfm)
```

```
(mod CCS is
  inc PROCESS . inc PREFIX . inc SUMMATION .
  inc PARALLELISM . inc RELABELLING .
  inc RESTRICTION . inc LABEL-CONGRUENCE .
 endm)
```

Before we explain the details of the strategy module, a word is needed on how to represent Modular SOS computations in Maude. Maude implements the rewriting logic calculus which has four inference rules given by reflexivity (a term can be rewritten to itself), transitivity (if t rewrites to t' and t' to t'', then t rewrites to t''), congruence (a rule can be applied to the subterms of t), and substitution (a rule can be applied to a kind preserving substitution). SOS does not have such a calculus. The present authors, with others, have proposed several techniques (e.g. [16,8]) to represent both modular and plain SOS computations in rewriting logic and have implemented them in Maude. Using a strategy, however, these techniques are not necessary since one has full control of the rule application. Reflexivity and transitivity are controlled by basic strategies, that is, if a basic strategy is applied, it represents one (rewriting) step. Congruence, however, needs to be controlled, that is, the application of a basic strategy should be done at the *top* operator and not on its subterms. That is why the '`top`' strategy is applied. The substitution inference rule is desired and therefore needs not to be controlled.

Instead of using the '`top`' strategy, we could have also used the technique implemented in MMT [8] to control Maude's default rewriting strategy, which is essentially a rewrite rule (labeled '`step`') with extra configuration constructors that impose a one-step rewrite for each rule application. It simplifies the strategy but adds extra declarations related to the '`step`' rule to the generated Maude module. Thus, the choice for the '`top`' operator produces cleaner Maude modules.

When applying a rule with premisses, the strategy should specify which is the strategy applied to solve each premise. In order to make the strategy extensible, we use a "abstract" strategy that will be instantiated later on.[3]

```
(stratdef PREM-STRAT is
  sop prem-strat .
 endsd)
```

Another implementation detail is that here we make explicit that the strategy '`prem-strat`' is applied in depth-first search to the premises of the transition rules. Another alternative could be to use breadth-first search if infinite recursive processes were allowed by using contexts.

The definition of the strategy is also modular. For each construct we define a strategy that applies the semantic rules to the top of each term and applies the

[3] What we really need is a parameterized strategy module, but the extension of the Maude strategy language with parametric modules is currently under study.

generic strategy on the premises, as explained above. We illustrate the strategy definition below for the prefix and parallel constructs.

```
(stratdef PREFIX-STRAT is  including PREFIX .
  sop prefix-strat .
  seq prefix-strat = top(prefix) .
endsd)

(stratdef PARALLELISM-STRAT is
  including PARALLELISM . including PREM-STRAT .
  sop parallelism-strat .
  seq parallelism-strat = top(par1{dfs(prem-strat)})
                        | top(par2{dfs(prem-strat) dfs(prem-strat)}) .
endsd)
```

The complete strategy, specified in module ‘CCS-STRAT’, is given by the union strategy ‘|’ of the basic strategies for each operator.

```
(stratdef CCS-STRAT is
  inc CCS .
  inc PREFIX-STRAT .       inc SUMMATION-STRAT .
  inc PARALLELISM-STRAT .  inc RELABELLING-STRAT .
  inc RESTRICTION-STRAT .
  sop ccs-strat .
  seq ccs-strat = prefix-strat
                | summation-strat
                | parallelism-strat
                | relabelling-strat
                | restriction-strat .
endsd)
```

Note that this strategy can be automatically generated by inspecting the semantics rules. It reflects the MSOS derivation mechanism, but it is not CCS dependent. Moreover, the strategy `ccs-strat` can only be used after concretizing the strategy `prem-strat` as the following module does.

```
(stratdef CCS-STRAT+ is
  including CCS-STRAT .
  seq prem-strat = ccs-strat .
endsd)
```

This mechanism, that allows the modular definition of the strategies, will be further exemplified below when CCS will be extended with a priority operator in Section 5.

2.3 The Implementation of MMT+MSL

The current version of MMT+MSL relies on the prototypes for MMT and MSL implemented in Full Maude. As we mentioned before, Full Maude implements an extensible module algebra for Maude. It provides a basic infrastructure, that is, a set of meta-functions, to extend Maude, that relies on Maude's meta-programming interface. (For instance, ‘`metaParse`’ produces a term out of a given list of identifiers and a grammar.)

The Maude predefined ‘`LOOP-MODE`’ module defines a read-eval-print loop that should be extended in order to define a command-line interface for a Maude extension. It defines a triple containing the input (of sort '`QidList`'), the current state of the system (of sort '`State`'), and the output (of sort '`QidList`'), given by the infix operator '`[_,_,_] : QidList State QidList -> System`'. The descent functions above should then manipulate these values. This is what Full Maude does, as described below.

In the reminder, we first comment on the general technique to extend Maude, and then move to Full Maude. The following steps should be done: to define a module M representing the syntax of the language that one wants to represent in Maude; to define a meta-function that given a meta-term in the meta-representation of M, produces a meta-term in the meta-representation of a Maude module; to define an interface that encapsulates how commands in the language captured by M are translated into commands over the Maude representation of M; and how the "answer" given by the Maude system is translated back into the language of M.

Full Maude provides an infrastructure to implement all these steps. There is a parsing infrastructure to handle Maude-based modules; a transformation infrastructure that given a structured Maude module, that is, a Maude module with inclusions, flattens it into a single Maude module; a database, that is, a term that holds all the modules loaded in Full Maude, together with information necessary to execute Full Maude's commands (the database structure may be extended to "cache" information that may be necessary for computing with (the representation of) terms in M); and finally a pretty-printing infrastructure. This infrastructure is already used by Full Maude to specify parameterized modules, and object-oriented modules for instance.

MMT and MSL were implemented as Full Maude extensions individually. (Concrete details on how both tools have been implemented at the metalevel can be found in [2,7].) The combination was straightforward: we wrote a few modules that joined each of these parts, that is, parsing, transformation, database handling, and pretty printing. The module '`MMT+MSL-SIGN`' extends the module '`STRAT-GRAMMAR`' (that itself extends Full Maude's grammar with the one for strategies) with the grammar for MSDF syntax defined in module '`MSDF-SIGNATURE`'.

```
fmod MMT+MSL-SIGN is
  including META-STRAT-SIGN .
  op MMT+MSL-GRAMMAR : -> FModule .
  eq MMT+MSL-GRAMMAR = addImports((including 'MSDF-SIGNATURE .), STRAT-GRAMMAR) .
endfm
```

The module '`MMT+MSL`' puts the Maude modules from both tools together. It replaces Full Maude's module for handling input and output since there can not be non-determinism between Full Maude's rules and MMT+MSL. First it includes the extended grammar, then the database handling modules for MSL and MMT, the predefined units for MMT and finally the loop mode module.

```
mod MMT+MSL is
  pr MMT+MSL-SIGN .           pr STRAT-DATABASE-HANDLING .
  pr MMT-DATABASE-HANDLING .  pr PREDEF-UNITS .
  inc LOOP-MODE .
```

Three rules handle the read-eval-print loop for MMT+MSL. The rule labeled ‘`init`’ below simply initializes Full Maude’s database with its default values and adds a “banner” to the MMT+MSL. Full Maude’s database is the state of the system declared by the ‘`LOOP-MODE`’ module. It uses Maude object-oriented notation. The database structure was extended both by MSL and the MMT. The attributes ‘`state`’, ‘`stratDefs`’, ‘`results`’, and ‘`repeat`’ are used by MSL to, respectively, represent the search tree (either a stack, representing backtrack points, for depth-first search or a queue, with unsolved terms, for breadth-first search), a meta-module representing the strategy definitions, a set of terms representing the solutions found so far, and a flag for the option of showing or not repeated results. The attribute ‘`step-flag`’ is declared by MMT and holds the option to use MMT’s built in technique to handle MSOS computations (the ‘`step`’ rule) or not.

```
  rl [init] : init
   => [nil, < o : STRATDB | db : initial-db, input : nilTermList, output : nil,
              default : 'CONVERSION, state : emptyP, step-flag : false,
              stratDefs : none, results : emptyTermSet, repeat : false >,
          ('\n '\t '\s '\s '\s '\s '\s '\s '\s
           'MMT 'and 'Strategy 'Full 'Maude '2.1.1 'Combined '\s  '\n)] .
```

Rule ‘`in`’ allows for both modules to be entered in MMT+MSL by invoking ‘`metaParse`’ with the combined grammar in module ‘`MMT+MSL-GRAMMAR`’. There is another ‘`in`’ rule that handles syntax errors in the input.

```
  crl [in] :
    [QIL, < O : X@Database | input : nilTermList, output : nil, Atts >, QIL']
    => [nil, < O : X@Database |
                input : getTerm(metaParse(MMT+MSL-GRAMMAR, QIL, 'Input)),
                output : nil, Atts >, QIL']
    if QIL =/= nil /\  metaParse(MMT+MSL-GRAMMAR, QIL, 'Input) : ResultPair .
```

Rule ‘`out`’ simply prints to the screen what was produced as output by Full Maude.

```
  crl [out] :
    [QIL, < O : X@Database | output : QIL', Atts >,  QIL'']
    => [QIL, < O : X@Database | output : nil, Atts >, (QIL'' QIL')]
    if QIL' =/= nil .
endm
```

Finally, the MMT+MSL tool can be used after loading into Maude the modules implementing MMT and MSL and the two modules described above.

3 Representing Ordered SOS with Strategies

In this section we show how a Maude strategy can be defined to execute an ordered SOS specification as defined in [15].

A set of rules with an ordering (any binary relation) is called *ordered SOS* (OSOS) if it contains positive GSOS rules only (that is, no rule has negative premises). In [15] it is shown that GSOS and OSOS have the same expressive power. A GSOS rule is an inference rule in the following form

$$\frac{\left\{ X_i \overset{\alpha_{ij}}{\rightarrow} Y_{ij} \right\}_{i \in I, j \in J_i} \quad \left\{ X_k \overset{\beta_{kl}}{\not\rightarrow} \right\}_{k \in K, l \in L_k}}{f(X_1, \ldots, X_n) \overset{\gamma}{\rightarrow} C[\mathbf{X}, \mathbf{Y}]}$$

where I and K are subsets of $\{1, \ldots, n\}$ and all J_i and L_k are finite subsets of $\mathbb{N}$; $\mathbf{X}$ is the sequence $X_1, \ldots, X_n$ and $\mathbf{Y}$ is the sequence of all Y_{ij}; and C is a context.

For example, the following rules define the behavior of an hypothetical operator f, for constants a and b:

$$\frac{X \overset{a}{\rightarrow} Y}{f(X) \overset{a}{\rightarrow} f(Y)}\, r_1 \qquad \frac{X \overset{b}{\rightarrow} Y}{f(X) \overset{b}{\rightarrow} g(Y)}\, r_2 \qquad \{r_1 < r_2\}$$

where the relation $r_1 < r_2$ between the rules specifies that the first rule (r_1) is only applied when the second rule (r_2) cannot be applied. That is, the binary relation on rules defines the *order of their application* when deriving transitions. So, a rule r can be used to derive a transition if all its premises are valid and no rule *higher* than r is applicable (it contains a premise which is not valid).

A Maude strategy can be used to take into account this order on rules. First, inference rules are represented as SOS rules in MMT but their application will be controlled by a strategy. The strategy makes use of the '`top`' combinator to restrict the application of the given strategy to the outermost term. The '`not`' combinator checks if the higher rules cannot be applied.

For the previous example with $r_1 < r_2$, the part of the strategy that tries to apply r_2 is simply `top(`r_2`{`s`})`, where s is the abstract strategy to be used to solve the premise. The part of the strategy regarding r_1 is a bit more complex since it has rules higher in the rule ordering. It is '`not(top(`r_2`{`s`})) ; top(`r_1`{`s`})`', which means that before applying r_1 we have to know that r_2 cannot be applied. The strategy '`not(top(`r_2`{`s`}))`' succeeds if `top(`r_2`{`s`})` fails.

For an OSOS specification $(\Sigma, A, R, <)$ the following algorithm builds the strategy identified by s that controls the way rules in R should be applied. It uses a function *rules*(f) to obtain the rules in R that define the behavior of the operator f and a function *higher*(r) that returns all the rules $r' \in R$ such that $r < r'$.

```
strategy := 'idle'
for each operator $f \in \Sigma$ do
  for each rule $r_f \in rules(f)$ do
    if $higher(r_f) = \emptyset$ then
      append '| top($r_f${$s$})' to the strategy
    else
      append '| not($union(higher(r_f))$) ; top($r_f${$s$})' to the strategy
where $union(r_1, \ldots, r_n)$ = top($r_1${$s$}) | ... | top($r_n${$s$}).
```

The strategy '`not(top(`r_1`{`s`}) | ... | top(`r_n`{`s`}))`' succeeds when none of the rules $r_1, \ldots, r_n$ can be applied. Note that if a given rule r has $m > 1$ premises then the strategy s should be repeated m times within the curly brackets. If $m = 0$ then r is not parameterized.

In [15] a transition relation $\rightarrow$, that takes into account the ordering on rules, is associated to a process language $(\Sigma, A, R, <)$. Formally, $\rightarrow = \bigcup_{l<\omega} \rightarrow^l$, where the transitions in $\rightarrow^l$ are defined as follows

$$p \xrightarrow{\alpha} p' \in \rightarrow^l \text{ if } d(p) = l \text{ and } \exists r \in R, \rho.\Big(\rho(con(r)) = p \xrightarrow{\alpha} p' \text{ and}$$
$$\rho(pre(r)) \subset \textstyle\bigcup_{k<l} \rightarrow^k \text{ and } \forall r' \in higher(r).\rho(pre(r')) \not\subset \bigcup_{k<l} \rightarrow^k \Big).$$

Theorem 3.1 *The transition relation induced by an OSOS specification is preserved by the associated SOS specification with the strategy built by the above algorithm.*

Proof sketch. By induction on the depth of the process term. The base case is when the process is a constant, and therefore the rule r does not have any premise and $higher(r) = \emptyset$. The strategy produced by the algorithm has r as one of its alternatives. For the inductive case, since $\forall r' \in higher(r).\rho(pre(r')) \not\subset \bigcup_{k<l} \rightarrow^k$ holds, then by inductive hypothesis the strategy `not(`$union(higher(r))$`)` succeeds since the application of each rule r' fails; and also, $\rho(pre(r)) \subset \bigcup_{k<l} \rightarrow^k$ holds, thus by inductive hypothesis the strategy that is applied to the premises of r succeeds, which makes the application of strategy r`{`s`}` successful. □

4 Representing Negative Premises

In this section we first recall how a GSOS specification with negative premises can be represented in OSOS and then how a strategy can be used for that matter. We adapt material from [15] while recalling how negative premises are represented in OSOS.

For OSOS specifications with no constraints whatsoever regarding the rule ordering (besides being simply a binary relation among rules), given a rule with a negative premise, a new rule is generated above the given rule in the rule order. Its single premise is a positive version of the negative premise in the given rule. As for its conclusion, it has the same left-hand side of the conclusion of the given rule, but with process **0** on the right-hand side. Moreover, the generated rule should never be enabled for a configuration where its premise holds, hence, it should be above itself in the rule order.

Let us consider the specification for an hypothetical operator f given by the following rule:

$$\frac{X \xrightarrow{b} Y \quad X \overset{a}{\nrightarrow}}{f(X) \xrightarrow{b} t'}\ r_1$$

This specification can be written in OSOS simply by removing the negative premise from rule r_1, declaring the rule r_2 below, and an order where r_2 is above r_1. The specification then becomes as follows, where Y is a new variable in r_2.

$$\frac{X \xrightarrow{b} Y}{f(X) \xrightarrow{b} t'}\, r_1 \qquad \frac{X \xrightarrow{a} Y}{f(X) \xrightarrow{a} t'}\, r_2 \qquad \{r_1 < r_2,\ r_2 < r_2\}$$

Our SOS specification with a strategy is then given by rules r_1 and r_2, as above, together with the strategy '$s =$ `not(`r_2`{`s`})` `;` r_1`{`s`}` `|` `not(`r_2`{`s`})` `;` r_2`{`s`}`'. (The abstract strategy technique is not used here for simplicity.)

Clearly, the strategy '`not(`r_2`{`s`})` `;` r_2`{`s`}`' is not necessary (since it always fails) and the strategy could be simplified. Also, note that rule r_2 is really *never* applied. In the strategy '`not(`r_2`{`s`})` `;` r_1`{`s`}`' the strategy '`not`' only checks if the premises hold. Another remark is that a rule for the operator f with the premise of r_2 could already exist in the original GSOS specification. In this case the specification is called *natural* in [15]. Thus, from natural specifications is not necessary to generate a new rule and the strategy could simply take the existing rule into account. Note that to handle this case properly, the implementation of the function *higher* needs to properly handle loops in the rule ordering.

OSOS specifications can be *partial*, meaning that its order is irreflexive and transitive. (Partial OSOS specifications are also equivalent to GSOS specifications according to [15].) Since the order has to be partial, the technique of having a rule above itself can not be used. The technique to represent the negative premise in partial OSOS relies on an extended action set with an *error* action and a rule that restricts process evolution to processes that do not signal *error*. A rule is also generated in the form of the one produced by the technique for non-partial OSOS, which is also above the given rule in the rule order, but with the *error* action in the conclusion. Also, the initial configuration should be augmented with the restriction to *error*.

The specification for r_1 in partial OSOS is given by the following three rules:

$$\frac{X \xrightarrow{b} Y}{f(X) \xrightarrow{b} t'}\, r_3 \qquad \frac{X \xrightarrow{a} Y}{f(X) \xrightarrow{error} \mathbf{0}}\, r_4 \qquad \{r_3 < r_4\}$$

$$\frac{X \xrightarrow{\alpha} Y}{X\backslash error \xrightarrow{\alpha} Y\backslash error}\, r_5 \qquad \alpha \neq error$$

where the initial configuration with process $f(p)$ should be augmented with the restriction to action *error*, as in $f(p)\backslash error$.

In this case the strategy would be '$s =$ `not(`r_4`{`s`})` `;` r_3`{`s`}` `|` r_5`{`s`}`'. Again, a simplification is possible, due to the same reason as for the non-partial case. Since the strategy '`not`' does not apply a rule, only checks for its premises, the conclusion of r_4 is irrelevant. Therefore there is neither a need to extend the action set with the *error* action nor to add r_5 to the rule set. With this simplification the resulting strategy is '$s =$ `not(`r_4`{`s`})` `;` r_3`{`s`}`'.

Both translations (including the generation of new rules and the corresponding order) can be done automatically by inspecting the GSOS rules. Then the strategy can be generated as explained in Section 3. (The implementation of this transformation, however, is part of future work.)

5 CCS with Priority

5.1 A Priority Operator with Strategies

An example of the usage of negative premises is a priority operator θ [12], which given a process P builds a new process that performs action α of P if P cannot perform any action with a priority higher than α. This operator is specified by the following rule scheme r_θ.

$$\frac{X \xrightarrow{\alpha} X' \quad \forall_{\beta>\alpha}\, X \overset{\beta}{\not\rightarrow}}{\theta(X) \xrightarrow{\alpha} \theta(X')}\, r_\theta$$

Given a finite set of actions, the above scheme can be represented by many rules like r_θ but without the negative premise and with an ordering among them. An example strategy for r_θ is '$s =$ `not(`r_{θ_c}`{`s`} | `r_{θ_b}`{`s`}) ; `r_{θ_a}`{`s`} | not(`r_{θ_c}`{`s`}) ; `r_{θ_b}`{`s`} | `r_{θ_c}`{`s`}`', given a set of action labels $\{a, b, c\}$ with the ordering $\{a < b, a < c, b < c\}$, and the rules for the priority operator labeled r_{θ_a}, r_{θ_b}, and r_{θ_c}. (Again the abstract strategy technique is not applied for simplicity.)

However, this specification can be further simplified. Strategies may be applied with a particular *substitution*. Thus, instead of having three rules, in this example, we may specify a single rule r_θ with an action variable that may become bound to the three different label actions, thus giving rise to the strategy '$s =$ `not(`r_θ`[A` $\leftarrow c$`]{`s`} | `r_θ`[A` $\leftarrow b$`]{`s`}) ; `r_θ`[A` $\leftarrow a$`]{`s`} | not(`r_θ`[A` $\leftarrow c$`]{`s`}) ; `r_θ`[A` $\leftarrow b$`]{`s`} | `r_θ`[A` $\leftarrow c$`]{`s`}`', where '`A`' is an action variable.

For arbitrary large (but finite) set of actions, the strategy could be parameterized by a list of action labels representing the action labels above a given one. If we consider the following function *forall* below that produces a strategy out of a list of action labels, the strategy for an action label a with the function application *higher*(a) returning a list of action labels, would be given by the expression '`not(`*forall*(*higher*(a))`) ; `r_θ'.

$$\mathit{forall}_{r_\theta}(l, ls) = r_\theta\,\texttt{[A} \leftarrow l\texttt{]} \;\texttt{|}\; \mathit{forall}_{r_\theta}(ls)$$
$$\mathit{forall}_{r_\theta}(nil) = \mathit{idle}$$

5.2 CCS with the Priority Operator in MMT+MSL

The specification of CCS in Section 2.2 can be very easily extended, given the representation of priorities as strategies in Section 5.1. First the syntax of processes must be extended with the priority operator and the transition rule set must be extended with a new transition rule for priorities as r_θ in Section 5.1 but without the negative premises as we explained before. The Maude module '`CCS-PRI`', that includes the '`CCS`' module above and '`PROCESS-WITH-PRIORITY`' is also defined.

```
(msos PROCESS-WITH-PRIORITY is
  Process ::= theta (Process) [prec 20] .
  Label = {trace' : Action*, ...} .

              Process -{trace' = Action, ...}-> Process'
  [theta] -- -----------------------------------------------------------------
              theta (Process) : Process -{trace' = Action, ...}-> theta (Process') .
sosm)
```

The strategy has to be extended with the new strategies to represent the negative premises as explained in Section 5.1.

```
(stratdef PRI-STRAT is
  including PREM-STRAT .
  sop pri-strat .
  seq pri-strat = not(top(theta[Action <- c]{dfs(prem-strat)}) |
                      top(theta[Action <- b]{dfs(prem-strat)}))
                    ; top(theta[Action <- a]{dfs(prem-strat)})
                  | not(top(theta[Action <- c]{dfs(prem-strat)}))
                    ; top(theta[Action <- b]{dfs(prem-strat)})
                  | top(theta[Action <- c]{dfs(prem-strat)})) .
endsd)
```

The module 'CCS-PRI-STRAT', that replaces 'CCS-STRAT+', combines the strategy for basic CCS with the strategy for the priority operator, and establishes that the premises should be rewritten using the new whole strategy is the following one:

```
(stratdef CCS-PRI-STRAT is
  including CCS-STRAT .  including PRI-STRAT .
  sop ccs-pri .
  seq ccs-pri = ccs-strat | pri-strat .
  seq prem-strat = ccs-pri .
endsd)
```

6 Final Remarks

In [13] the authors present a prototype for GSOS specifications in Maude using the meta-level. Our approach represents OSOS, which is equivalent to GSOS [15], using the object level. Of course, it is still necessary to automate the translation from negative premises to orders and then to strategies. Moreover, to represent OSOS (and therefore GSOS) is *one possible* application of strategies. Maude (with strategies) could be used directly to represent any application with strategies, including OSOS. However, if one wants to make its specifications modular, the rewrite theories would have to be extended to cope with the modularity requirements.

The current version of the prototype does not support strategy module inclusion, even though there is notation (and semantics) for them in [7]. All the strategy definitions have to be declared in a single module. Part of our future work is to fully support the strategy language. Besides the automation of the translation of negative premises to strategies, a case study that we would like to approach in a near future is the implementation of E-LOTOS semantics [6], where negative premises are used in order to guarantee that urgent actions occur before time elapses.

Acknowledgments

We would like to thank Fabricio Chalub for his support on the implementation of MMT+MSL, Narciso Martí-Oliet for his encouragement, the anonymous referees for their insightful comments, and Peter Mosses for his careful review.

References

[1] Chalub, F., "An Implementation of Modular Structural Operational Semantics in Maude," Master's thesis, Universidade Federal Fluminense (2005), http://www.ic.uff.br/~frosario/dissertation.pdf.

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

[3] Clavel, M., F. Durán, S. Eker, J. Meseguer and M.-O. Stehr, *Maude as a formal meta-tool*, in: J. Wing, J. Woodcock and J. Davies, editors, *FM'99 — Formal Methods, Proc. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Volume II*, Lecture Notes in Computer Science **1709** (1999), pp. 1684–1703.

[4] Durán, F., "A Reflective Module Algebra with Applications to the Maude Language," Ph.D. thesis, Universidad de Málaga (1999).

[5] Hidalgo-Herrero, M., A. Verdejo and Y. Ortega-Mallén, *Looking for Eden through Maude and its strategies*, in: F. López-Fraguas, editor, *V Jornadas sobre Programación y Lenguajes, PROLE 2005* (2005), pp. 13–23.

[6] ISO/IEC, *Information technology — Enhancements to LOTOS (E-LOTOS)*, International Standard ISO/IEC FDIS 15437 (2001).

[7] Martí-Oliet, N., J. Meseguer and A. Verdejo, *Towards a strategy language for Maude*, in: N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, Electronic Notes in Theoretical Computer Science **117** (2005), pp. 417–441.

[8] Meseguer, J. and C. Braga, *Modular rewriting semantics of programming languages*, in: C. Rattray, S. Maharaj and C. Shankland, editors, *Algebraic Methodology and Software Technology: 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 2004, Proceedings*, Lecture Notes in Computer Science **3116** (2004), pp. 364–378.

[9] Mosses, P. D., *A modular SOS for ML concurrency primitives*, Technical Report BRICS-RS-99-57, Department of Computer Science, University of Aarhus (1999).

[10] Mosses, P. D., *Modular structural operational semantics*, Journal of Logic and Algebraic Programming **60-61** (2004), pp. 195–228.

[11] Mosses, P. D., *A constructive approach to language definition*, Journal of Universal Computer Science **11(7)** (2005), pp. 1117–1134.

[12] Mousavi, M., "Structuring Structural Operational Semantics," Ph.D. thesis, Technische Universiteit Eindhoven (2005).

[13] Mousavi, M. and M. A. Reniers, *Prototyping SOS meta-theory in Maude*, in: P. D. Mosses and I. Ulidowski, editors, *Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005)*, Electronic Notes in Theoretical Computer Science **156(1)** (2006), pp. 135–150.

[14] Rosa-Velardo, F., C. Segura and A. Verdejo, *Typed mobile ambients in Maude*, in: H. Cirstea and N. Martí-Oliet, editors, *Proceedings of the 6th International Workshop on Rule-Based Programming (RULE 2005)*, Electronic Notes in Theoretical Computer Science **147** (2006), pp. 135–161.

[15] Ulidowski, I. and I. Phillips, *Ordered SOS process languages for branching and eager bisimulations*, Information and Computation **178** (2002), pp. 180–213.

[16] Verdejo, A. and N. Martí-Oliet, *Executable structural operational semantics in Maude*, Journal of Logic and Algebraic Programming **67** (2006), pp. 226–293.