Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 194 (2008) 93–109
www.elsevier.com/locate/entcs

Prototyping A&A ReSpecT in Maude

Matteo Casadei[1] Andrea Omicini[2] Mirko Viroli[3]

Alma Mater Studiorum—*Università di Bologna,*
via Venezia 52, 47023 Cesena, Italy

Abstract

The formal modelling of programming languages has always been a challenging activity due to the gap occurring between formal definition and actual implementation. On the other hand, the Maude rewriting language has already proven to be a suitable tool to bridge the gap between theory and practice when implementing the operational semantics of programming languages. In particular, Maude has been exploited to model languages belonging to different paradigms and levels of abstraction, leading to specifications that represent *de facto* executable prototypes of such languages.
In this paper we focus on A&A ReSpecT, a coordination language based on the agents and artifacts (A&A) meta-model, and exploit Maude to generate an execution machine for A&A ReSpecT programs, acting as an implementation of its operational semantics.

Keywords: Rewriting logic, Maude, coordination languages, ReSpecT, agents and artifacts.

1 Introduction

Developing successful coordination technologies has always been quite a critical aspect of distributed systems' engineering [16,7,8,15]. On one hand, coordination typically tackles the most crucial part of a system, namely, the locus where interactions between system's subparts occur and are managed—the very place where the set of components becomes *the system*. On the other hand, such a part is typically quite complex and tricky, so that underspecification of details easily leads to either undesired behaviour or incorrect systems [1,3]. The need for tackling these issues naturally calls for a formal treatment. This generally allows to devise coherent models of coordination technologies: models whose specification boundaries are well-defined, and whose internal structure can be proven correct (safe and live) according to the intended requirements.

[1] Email: m.casadei@unibo.it
[2] Email: andrea.omicini@unibo.it
[3] Email: mirko.viroli@unibo.it

1571-0661 © 2008 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2008.03.101

The formalisation of models or languages is typically provided through a BNF-like grammar for syntactic aspects, and a rule-based definition for the operational semantics, both written down by hand—in the context of coordination languages and models they are mostly based on the pioneering work described in [2]. As such formalisations become complex, they experience the same problems of the language or system they intend to model, making it difficult to check their validity, use them to prove any useful property, and finally guide the development of a correct implementation. Several tools have been introduced to provide quite an abstract language by which system specifications can be written, checked for well-formedness, automatically executed, and used by some property verifier based e.g. on model checking. One of these well-known frameworks is Maude [5], a term-rewriting system and language allowing syntactic aspects to be specified in an ad-hoc way (by using a type-based algebraic approach), and adopting term-rewriting rules in the style of standard structural operational semantics [17] to define behavioural aspect. More precisely, Maude is based on rewriting logic [9] and can be used to model the behaviour of a wide range of languages and distributed systems by means of rewriting rules, while all the syntactic aspects are addressed by defining algebraic functions. Maude has already been exploited for several languages such as π-calculus, CCS and LOTOS, as well as coordination models like Reo [10]—see more on its many applications through [5].

In this paper we adopt Maude to develop a specification of the A&A ReSpecT language [11] for programming *tuple centres* in the context of TuCSoN coordination infrastructure for multiagent systems (MAS) [14,13]. Tuple centres can be seen as coordination virtual machines, namely, engines, which can be programmed by coordination rules governing the interactions between agents acting in a distributed setting. Accordingly, A&A ReSpecT is a scripting language that can be used to change the default, Linda-like behaviour of a tuple centre, in terms of rules that intercept events and accordingly modify the tuple dataspace, thus implementing any coordination policy [6]. Our Maude specification of A&A ReSpecT is developed in a multiset rewriting style inspired by process algebraic approaches as in [2], and can be seen as a prototype implementation of a tuple centre. Its main applications include the ability to act as a reference specification for the language syntax and semantics, as a tool to verify the correctness of A&A ReSpecT programs in terms of intended behaviour, and as a basis for evaluating extensions of A&A ReSpecT to tackle new domains such as e.g. the stochastic-oriented setting.

Here we focus on Maude as a tool to verify correctness of programs. In particular, since A&A ReSpecT is an intrinsically nondeterministic language, its behaviour may appear correct after few tests when one programs a tuple centre. Nonetheless, there could be some hidden bugs that would appear only later due to some unpredictable different transition paths within the set of the allowed ones. By using our Maude specification, it is possible to search all computational paths and check if they all lead to correct and/or safe states.

The remainder of this paper is organised as follows. Section 2 provides a brief discussion of the A&A ReSpecT model; Section 3 presents the Maude specification

of A&A ReSpecT; Section 4 discusses two application cases that can be validated using MAUDE; finally Section 5 provides for final remarks.

2 A&A ReSpecT in Short

Tuple centres [13] are coordination media that can be described as programmable tuple spaces. In fact, the behaviour of a tuple centre can be defined through a specification language determining its coordinating behaviour. Tuple centres are adopted by MAS coordination infrastructures such as TuCSoN [14] and MARS [4].

A&A ReSpecT [11] is a logic-based language to specify the behaviour of tuple centres. As a behaviour specification language, A&A ReSpecT:

- enables the definition of computations within a tuple centre, called *reactions*, and
- makes it possible to associate reactions to events occurring in a tuple centre.

So, A&A ReSpecT has both a declarative and a procedural part. As a *specification language*, it allows events to be declaratively associated to reactions by means of specific logic tuples, called *specification tuples*, whose form is `reaction(E,G,R)`. Such a tuple associates a reaction $R\theta$ to an event Ev if $\theta = mgu(E,Ev)$[4] and `G` is a true guard. Accordingly, when an event Ev matching `E` occurs and all the guards predicates of `G` are true, reaction $R\theta$ is triggered for the execution in the tuple centre.

As a reaction language, A&A ReSpecT enables reactions to be procedurally defined in terms of sequences of logic reaction goals, each one either succeeding or failing. A whole reaction succeeds if all its reaction goals succeed, fails otherwise. Each reaction is executed sequentially with a transactional semantics: so, a failed reaction has no effect on the state of a logic tuple centre.

Guard predicates allow reactions to be triggered depending on the kind of an event: a request from an agent, a successful response to a link operation from another tuple centre, ... Reaction goals are essentially of four sorts: operations on the tuple-centre state, link operations on other tuple centres, observations on the triggering event, and simple computations—like, mathematical, or logic-based ones. Operations on a tuple-centre state can be performed by external agents, other tuple centres, and the tuple centre itself through reactions. All these operations have essentially the same syntax (mostly, Linda-like `in`, `inp`, `rd`, `rdp`, and `out`) and basically the same well-known semantics.

All the reactions triggered by an event are executed before serving any other event: so, agents perceive the service of the event and the execution of all the associated reactions altogether as resulting from a single transition of the tuple centre state. As a consequence, the effect of a coordination primitive on a logic tuple centre can be made as complex as needed by the coordination requirements of a MAS. Generally speaking, since A&A ReSpecT has been shown to be Turing-equivalent [6], any computable coordination law can be in principle encapsulated into a A&A ReSpecT tuple centre. This is why A&A ReSpecT can be assumed as

[4] *mgu* is the most general unifier, as defined in logic programming.

a general-purpose core language for coordination: a language that can be used to represent and enact policies and rules of any sort for collaboration support systems.

For more details on A&A ReSpecT syntax and semantics, we forward the interested reader to the A&A ReSpecT home page [18], as well as to the original ReSpecT [13,12] and A&A ReSpecT [11] papers.

3 Executable Specification of A&A ReSpecT in Maude

Maude is a high-performance reflective language supporting equational and rewriting logic specifications that make it possible to specify a wide range of applications. The basic unit of a Maude program is the *module*, which is essentially a set of definitions forming an algebra: a module can be of a *functional* or *system* kind. Functional modules contain type and operation declarations, together with *equations* that are *equational rewriting* rules defining abstract-data types and are useful to declare algorithmic aspects of computing systems. System modules define *rewriting laws*, i.e. transition rules that implement a concurrent *rewriting semantics* by which it is possible to deal with aspects related to interaction and system evolution.

The executable specification of A&A ReSpecT is written in Maude and based on the formal specification described in [11]. This specification is composed of three Maude modules:

- `RESPECT-SYNTAX`, defining the syntax of the language.
- `RESPECT-SEMANTICS-TYPES`, providing a definition of all the types needed to specify the actual semantics of A&A ReSpecT.
- `RESPECT-SEMANTICS`, implementing the semantics of A&A ReSpecT in terms of Maude rewriting rules.

The following subsections report a description of these modules. Instead of describing few snippets of the specification as in other works, we here prefer to provide a complete description. The main reason is that the Maude specification should really play the role of the formal model of A&A ReSpecT, and we hence aim at providing a complete account of it—for this is actually compatible with the space available for this paper. Only few selected features are actually left out of the description, since they include purely algorithmical parts of computation.

3.1 `RESPECT-SYNTAX` Module

This module contains the definition of the A&A ReSpecT syntax. Since every A&A ReSpecT reaction can be viewed as a logic term, this module defines the A&A ReSpecT syntax by extending `TERMS-SYNTAX`, which models a logic term according to logic programming. More precisely, `TERMS-SYNTAX` allows a logic term to be expressed as `'name(TL)` where `'name` is a quoted identifier, that is, an arbitrarily chosen string starting with a single quote, and `TL` is a variable denoting a list of one or more logic terms. Furthermore, while a ground term is represented by a quoted identifier, a logic variable can be expressed in `TERM-SYNTAX` as `v('X)`, where `'X` is

```
mod RESPECT-SYNTAX is
  pr TERMS-SYNTAX .
  pr NAT .

  sort MultiTerm .
  subsort Term < MultiTerm .

  op nilMT : -> MultiTerm [ctor] .
  op _|_ : MultiTerm MultiTerm -> MultiTerm [ctor assoc comm id: nilMT] .

endm
```

Fig. 1. Definitions in `RESPECT-SYNTAX` module.

a quoted identifier denoting the variable name.

Figure 1 shows the `RESPECT-SYNTAX` code. The module defines a *multiset of terms* by introducing the `MultiTerm` sort. The `nilMT` constant denotes the empty multiset. Adopting these definitions, an A&A ReSpecT reaction can be expressed as a logic term of the form

```
'reaction(E, 'g(G), 'r(R))
```

where: *(i)* `E` is a logic term representing the event by which the reaction can be triggered, *(ii)* `G` is a list of logic terms denoting the guard predicates that must be satisfied for the reaction to be triggered, *(iii)* `R` is a list of logic terms denoting the reaction body, that is, the reaction goals that are executed when the reaction is triggered. For instance

```
'reaction( 'out('mytuple(v('X))),
           'g( 'from-agent ),  'r( in('tuple('done)))  )
```

models a valid reaction expressed according to the definitions provided in `RESPECT-SYNTAX` and `TERM-SYNTAX`.

3.2 RESPECT-SEMANTICS-TYPES Module

This module extends `RESPECT-SYNTAX` and `TERM-SUBS` in order to define the entities necessary for modelling A&A ReSpecT semantics. `TERM-SUBS` provides a MAUDE-based implementation of the *most general unifier* algorithm, as defined in logic programming.

Figure 2 reports the definitions in `RESPECT-SEMANTICS-TYPES`. This module defines the shape of the engine upon which A&A ReSpecT specifications are executed. According to `RESPECT-SEMANTICS-TYPES`, a tuple centre is modeled by the `TCState` sort as

```
< TCId # TCEl >
```

where `TCId` is a logic term representing the name of the tuple centre, and `TCEl` is a multiset of tuple-centre elements. Since A&A ReSpecT supports *linkability* [11], the executable specification needs to model the concept of distributed tuple centres. In particular, this is done by adopting the `DistributedState` sort, that allows to define multisets of tuple centres. The generic *tuple-centre element* is further specialized by defining several different entities, required to fully model a tuple

```
mod RESPECT-SEMANTICS-TYPES is
 pr RESPECT-SYNTAX .
 pr TERMS-SUBS .

 *** TUPLE-CENTRE DEFINITION

 sorts DistributedState TCState TCEl .
 subsort TCState < DistributedState .

 op nilEl : -> TCEl [ctor] .
 op _|_ : TCEl TCEl -> TCEl [ctor assoc comm id: nilEl] .

 op <_#_> : Term TCEl -> TCState [ctor] .

 op nilState : -> DistributedState [ctor] .
 op _|_ : DistributedState DistributedState -> DistributedState
                                        [ctor assoc comm id: nilState] .

 *** TUPLE-CENTRE-ELEMENT DEFINITION

 sorts VM-TCEvt EvtQueue Cause TCResult .
 subsort VM-TCEvt < EvtQueue .
 subsort VM-TCEvt < TCEl .
 subsort Term < TCResult .

 op emptyQ : -> EvtQueue [ctor] .
 op _,_ : EvtQueue EvtQueue -> EvtQueue [ctor assoc id: emptyQ] .

 op [_._._] : Cause Cause TCResult -> VM-TCEvt .
 op [_._._._] : Term Term Term Time -> Cause [ctor] .

 sorts ReactionState ReactionSet ReactionBody .
 subsort ReactionBody ReactionSet ReactionState < TCEl .

 op noReactionExec : -> ReactionState [ctor] .
 op [_@_] : Term MultiTerm -> ReactionSet [ctor] .

 op [_] : TermList -> ReactionBody [ctor] .

 op <_> : TCEl -> ReactionState [ctor] .

 sort TupleSet .
 subsort TupleSet < TCEl .

 op [_] : MultiTerm -> TupleSet [ctor] .

 sort OperationExec .

 op [_,_,_] : MultiTerm Term BindList -> OperationExec [ctor] .
 op X[_,_,_] : MultiTerm Term BindList -> OperationExec [ctor] .
 op [_,_] : Term MultiTerm  -> OperationExec [ctor] .


 sort Queue .
 subsort Queue < TCEl .
 op [_@_] : Term EvtQueue -> Queue [ctor] .

 *** FUNCTION DEFINITIONS

 op triggeredReactions : Term VM-TCEvt MultiTerm -> MultiTerm .
 ...

 op guard : Term VM-TCEvt TermList -> Bool .
 ...

 op execObs : Term VM-TCEvt TermList -> TermList .
 ...

 op unifyReactions : BindList TermList -> TermList .
 ...

endm
```

Fig. 2. Definitions in RESPECT-SEMANTICS-TYPES module.

centre.

For instance, the `VM-TCEvt` sort represents an event occurring on an A&A ReSpecT tuple centre, according to the definition given in [11]. More specifically, this sort is defined by the following syntax:

```
[StartCause . Cause . Res ]
```

where `StartCause` denotes the initial cause of the event, `Cause` represents the current cause and `Res` is a logic term denoting the event result. The initial and current causes of an A&A ReSpecT event are modelled by the `Cause` sort as follows:

```
[EvtOp . Src . Target . Time]
```

where `EvtOp` is the logic term denoting the tuple-centre operation that generates the event, while `Src` and `Target` are identifiers representing the source and the target of `EvtOp`, respectively. The `Time` variable denotes the time at which `EvtOp` was first emitted by `Src` on the tuple centre. As pointed out in [11], both the source and the target of a tuple-centre operation can be either an agent or a tuple centre.

A list of tuple-centre events is modelled by the `EvtQueue` sort, while the input and the output queue of a tuple centre are represented by the `Queue` sort as follows:

```
[ Id @ Q ]
```

where `Id` is a logic term that can be equals to `'input` or `'output` depending on the type of the queue one wants to define. `Q` denotes a variable of `EvtQueue` sort representing the list of tuple-centre events waiting to be served in an input queue or already served in an output queue.

The `ReactionSet` sort models a set of reactions:

```
[Type @ ReactionSet]
```

`Type` identifies a logic term that can be set to `'trig` or `'spec`. While `'trig` represents the set of reactions triggered by a tuple-centre event, `'spec` denotes the entire set of reactions acting as a specification of the tuple-centre behaviour.

To model the set of tuples contained in a tuple centre, we adopted the syntax

```
[TupleSet]
```

where `TupleSet` is a variable of `MultiTerm` sort representing the tuples within the tuple centre.

To model the environment where all the reaction goals in a reaction body are executed, we introduced the `ReactionState` sort by adopting the syntax `< El >`, where `El` denotes the multiset of the tuple-centre elements necessary to perform the execution of the reaction goals. For instance, `El` has to contain both the tuple-centre element denoting the temporary multiset of tuples target of the reaction-goal execution, and the tuple-centre element denoting the reactions triggered during the execution of the reaction goals. The `noReactionExec` constant represents the empty reaction execution environment, that is, the tuple-centre condition where no pending reactions are to be executed.

Adopting all the syntactic representations so far given, it is easy to define a tuple centre as

```
<TcId # ['input @ Q] | ['output @ Q'] |
              [TupleSet] | ['spec @ ReactSet] | noReactionExec >
```

The above syntax denotes a tuple centre with no reactions waiting to be served.

Finally, `RESPECT-SEMANTICS-TYPES` provides a definition of the following functions:

- *triggeredReactions*, given a set of reactions and an A&A ReSpecT event, returns the subset of reactions triggered by the event.
- *guard*, given a list of guard predicates and an A&A ReSpecT event, returns the boolean value `true` if all the predicates in the list are satisfied, `false` otherwise.
- *execObs*, given an observation predicate[5] `'obsPredicate(v('Obs))` and an A&A ReSpecT event, returns a logic term representing the most general unifier between the logic variable `'Obs` and the event property target of the observation predicate.
- *unifyReactions*, considering a reaction `'reaction(Evt, 'g(G), 'r(R))`, an A&A ReSpecT event e and $f = mgu(e, \texttt{Evt})$, returns the result deriving from the application of f to `R` .

3.3 RESPECT-SEMANTICS Module

The complete code of `RESPECT-SEMANTICS` is reported in Figure 3. This module implements the A&A ReSpecT semantics in terms of rewriting rules that can be viewed as logically divided in two parts defining *(i)* the semantics of the `in`, `rd`, `out` and `no` primitives, and *(ii)* the semantics of an A&A ReSpecT tuple centre.

The semantics of an A&A ReSpecT tuple centre is defined by the following rewriting rules:

- `log`, modelling the logging of new operations from the input queue to the tuple centre. When the input queue of a tuple centre has a new event to be served and there are no triggered reactions to be executed, then the event is moved from the input queue to the tuple centre and the reactions to be triggered are calculated by the *triggeredReactions* function.
- `start-reaction-req`, modelling the phase where the reactions triggered by a previously logged operation are prepared for the execution.
- `service`, defining the phase in which operations waiting for response are served and moved to the output queue. This rule is a *conditional rewriting rule* where the condition is composed of another rewriting rule that represents the execution of the operation to be served.
- `start-reaction-resp`, modelling the phase where the reactions triggered during the service phase are prepared for the execution.

Then, a set of additional rewriting rules is defined. The goal of these rules is to specify the behaviour of the reaction execution phase:

[5] An observation predicate is a predicate that observes the state of an A&A ReSpecT event in order to get information on the event itself.

```
mod RESPECT-SEMANTICS is
pr RESPECT-SEMANTICS-TYPES .
pr BOOL .

*** DEFINITION OF LINDA-PRIMITIVE SEMANTICS

vars T T' : Term .
var Tu : MultiTerm .
var BL : BindList .
rl [out] :  [('out(T)) , Tu ] => [(Tu | T) , T , empty] .

crl [in] :  [('in(T)) , Tu | T' ] => [Tu , T' , BL]
if
BL := T // T' /\
BL :: BindList .

crl [rd] :  [('rd(T)) , Tu | T' ] => [Tu | T' , T' , BL]
if
BL := T // T' /\
BL :: BindList .

crl [no] :  [('no(T)) , Tu ] => [Tu , 'nil , empty]
if
absent(T , Tu) .

crl [in-fail] :  [('in(T)) , Tu ] => X[Tu , 'nil , empty]
if
absent(T , Tu) .

     crl [rd-fail] :  [('rd(T)) , Tu ] => X[Tu , 'nil , empty]
if
absent(T , Tu) .

crl [no-fail] :  [('no(T)) , Tu | T' ] => X[Tu | T' , 'nil , empty]
if
BL := T // T' /\
BL :: BindList .

vars Node Name Type OpName' OpName Evt Evt AgId TCId TCc TCct : Term .
vars Tu' TuO TuO' Res Res' Res'' TuTmp TRes Re' Re Sigma Sigma' : MultiTerm .
vars C C' : Cause .   vars El El' El'' El''' : TCEl .
vars InQ OutQ Out' : EvtQueue .
vars R R' G : TermList .
var State : DistributedState .
var VMEvt : VM-TCEvt .
vars RS RS' : ReactionState .

*** TUPLE-CENTRE-BEHAVIOR DEFINITION

crl [log] :
< TCc # ['input @ InQ, [C . C' . TRes]] | [Tu] |
['spec @ Sigma] | ['trig @ nilMT] | El | noReactionExec >
=>
< TCc # ['input @ InQ ] | [Tu] | ['spec @ Sigma] |
['trig @ Re] |  [C . C' . TRes] | El | noReactionExec >
if Re := triggeredReactions(TCc, [C . C' . TRes] ,Sigma) .

rl [start-reaction-req-undef] :
< TCc # ['trig @ ('reaction((Evt'),('g(G)),('r(R)))) | Re] |
[C . C' . 'undefined] | [Tu] | ['spec @ Sigma] | El |
noReactionExec >
=>
< TCc # ['trig @ Re] |  [C . C' . 'undefined] |
[Tu] | ['spec @ Sigma] | El |
< [R] | ['spec @ Sigma] | ['trig @ nilMT] | [Tu] |  [C . C' . 'undefined] >  > .

crl [start-reaction-req-def] :
< TCc # ['trig @ ('reaction((Evt'),('g(G)),('r(R)))) | Re] | [C . C' . TRes] |
[Tu] | ['spec @ Sigma] | El | noReactionExec >
=>
< TCc # ['trig @ Re] | [Tu] | ['spec @ Sigma] | El |
< [R] | ['spec @ Sigma] | ['trig @ nilMT] | [Tu] | [C . C' . TRes] > >
if TRes =/= 'undefined .

rl [r-exec-end] :
< TCc # ['trig @ Re] | [Tu] | ['spec @ Sigma] | El |
< [nil] | ['spec @ Sigma'] | ['trig @ Re'] | [Tu'] | El' > >
=>
< TCc # ['trig @ Re | Re'] | [Tu'] | ['spec @ Sigma'] | El |
noReactionExec > .
```

```
crl [service-link] :
< TCc # ['trig @ nilMT] | [Tu] | ['spec @ Sigma] | ['output @ OutQ] |
[C . [OpName'(T) . '@(Name, Node, 'tc) . TCId . Tm:Time] . 'undefined]|
El | noReactionExec > |
< '@(Name, Node, 'tc) # ['input @ InQ ] |  El'' >
=>
< TCc # ['trig @ Re] | [Tu'] | ['spec @ Sigma] |
['output @ OutQ, [C . [OpName'(T) . '@(Name, Node, 'tc) . TCId . Tm:Time] . Res]] |
El | noReactionExec > |
< '@(Name, Node, 'tc) #
   ['input @  [C . [OpName'(T) . '@(Name, Node, 'tc) . TCId . Tm:Time] . Res], InQ] |
   El'' >
if [(OpName'(T)) , Tu ] => [Tu' , Res , BL] /\
Re:=triggeredReactions(TCc,[C.[OpName'(T).'@(Name,Node,'tc).TCId.Tm:Time].Res],Sigma).

crl [service] :
< TCc # ['trig @ nilMT] | [Tu] | ['spec @ Sigma] | ['output @ OutQ] |
[C . [OpName'(T) . '@(Name, Node, 'agent) . TCId . Tim:Time] . 'undefined]| El |
noReactionExec >
=>
< TCc # ['trig @ Re] | [Tu'] | ['spec @ Sigma] |
['output @ OutQ, [C .[OpName'(T) .'@(Name, Node, 'agent) .TCId .Tim:Time] .Res] ] |
El | noReactionExec >
if [(OpName'(T)) , Tu ] => [Tu' , Res , BL] /\
Re := triggeredReactions(TCc, [C . [OpName'(T) . '@(Name, Node, 'agent) .
 TCId . Tim:Time] . Res], Sigma) .

rl [start-reaction-resp] :
< TCc # ['trig @ ('reaction((Evt')),('g(G)),('r(R)))) | Re] |
['output @ OutQ, VMEvt ] | [Tu] | ['spec @ Sigma] | El | noReactionExec >
=>
< TCc # ['trig @ Re] | ['output @ OutQ, VMEvt ] | [Tu] | ['spec @ Sigma] | El |
< [R] | ['spec @ Sigma] | ['trig @ nilMT] | [Tu] | VMEvt > > .

*** REACTION-EXECUTION DEFINITION
crl [r-exec-ok] :
< [(OpName(T)), R] | [Tu] | ['spec @ Sigma] | ['trig @ Re'] |
    [C . [Evt . AgId . TCId . Tim:Time] . TRes] | El >
=>
< [R'] | [Tu'] | ['spec @ Sigma] | ['trig @ Re' | Re] |
    [C . [Evt . AgId . TCId . Tim:Time] . TRes] | El >
if  [(OpName(T)) , Tu ] => [Tu' , Res , BL] /\
R' := unifyReactions(BL,R) /\
Re := triggeredReactions(TCId,[C. [OpName(T) .TCId .TCId .Tim:Time] .TRes],Sigma) .

crl [r-exec-fail] :
< [(OpName(T)), R] | [Tu] | ['trig @ Re'] | El >
=>
< [nil] | [Tu] | ['trig @ nilMT] | El >
if
[(OpName(T)) , Tu ] => X[Tu , 'nil , empty] .

crl [r-exec-pred] :
< [(OpName(T)), R] | VMEvt | El >
=>
< [R'] | VMEvt | El >
if R' := execObs(OpName(T), VMEvt, R) /\
   R' =/= 'noSolution .

crl [r-link-operation] :
< TCc # < El | [('?(TCct,Evt')),(R) ] | ['spec @ Sigma] | ['trig @ Re'] |
      [C . [Evt . AgId . TCId . Tim:Time] . TRes] > | El' > |
< TCct # ['input @ InQ] | El'' >
=>
< TCc # < El | [R] | ['spec @ Sigma] | ['trig @ Re' | Re] |
      [C . [Evt . AgId . TCId . Tim:Time] . TRes] >
| El' > |
< TCct # ['input @ InQ , [ C . [Evt' . TCc . TCct . Tim:Time] . TRes ]] | El'' >
if Re := triggeredReactions(TCc,[C . [Evt' . TCc . TCct . Tim:Time] . TRes], Sigma) .

endm
```

Fig. 3. Definitions in `RESPECT-SEMANTICS` module.

- `r-exec-ok`, modelling the successful execution of an operation predicate.
- `r-exec-fail`, modelling the execution with failure of an operation predicate.

- `r-exec-pred`, modelling the execution of an observation predicate.
- `r-exec-end`, modelling the end of the execution of an entire reaction.
- `r-link-operation`, modelling the end of the execution of a link operation, that is, an operation whose target is another tuple centre.

4 Case Studies of Coordination

In the following sections, we consider two coordination case studies in order to test and validate the behaviour of the A&A ReSpecT executable specification. More precisely, we focused on the *in-all* and *distributed-dining-philosophers* problems, showing for both a solution based on A&A ReSpecT and simulated by using our Maude executable specification.

4.1 In-All

The goal of this case study is to define a new primitive able to collect and remove all the tuples matching a given tuple template. The behaviour of such a primitive was defined in the A&A ReSpecT program shown in Figure 4. This program allows for the removal of all the tuples matching `'g('a)`.

The first reaction is triggered during the response phase of an `'in('g('a))` operation, as defined in the corresponding guard section. Then, the `'out('h('b))` reaction goal is executed, leading to the triggering of reactions 2 and 3. Tuple `'h('b)`, emitted upon the execution of the previous `out` operation, can be viewed as a *starting signal* for the tuple centre to remove all the `'g('a)` tuples. Reaction 2 succeeds if the tuple centre still contains one or more `'g('a)` tuples. In this case, a tuple `'h('b)` is kept in the tuple space, and reactions 2 and 3 are triggered once again. This allows the tuple centre to remove the remaining `'g('a)` tuples. Oppositely, reaction 3 succeeds if the tuple centre does not contain `'g('a)` tuples.

The program was tested by using the tuple-set represented in Figure 4 with the `termSet` constant, which models a multiset of tuples composed of 6 `'g('a)` tuples. Then we exploited the Maude command `search`, in order to explore the whole reachable state space and check if the behaviour of the *in-all* specification was as expected, leading to the removal of all the `'g('a)` tuples.

We ran the `search` command as follows:

```
search
< '@('tc1,'node, 'tc) #
  ['in @ Event] | [termSet] |
  ['spec @ reactionSet] | ['trig @ nilMT] |
  ['out @ Out:EvtQueue] |
  noReactionExec
>
=>!
D:DistributedState .
```

```
mod IN-ALL is
 including RESPECT-SEMANTICS .

 op termSet : -> MultiTerm .
 eq termSet = ('g('a)) | ('g('a)) | ('g('a)) | ('g('a)) | ('g('a)) | ('g('a)) .

 op reactionSet : -> MultiTerm .
 eq reactionSet =
    'reaction( 'in('g('a)),
            'g('response, 'from-agent),
            'r('out('h('b)))
    )
    |
    'reaction( 'out('h('b)),
            'g('from-tc),
            'r( ('in('g('a)),
                ('out('h ('b))),
                ('in('h('b)))))
    )
    |
    'reaction('out('h('b)),
            'g('from-tc),
            'r( ('no('g('a))),
                ('in('h('b)))))
    ) .

endm
```

Fig. 4. Definition of the *in-all* program.

where symbol `=>!` means that we are interested in final states (i.e. non-further-rewritable states), and `Event` is a `VM-TCEvent` defined as

```
[ StartCause:Cause.
 ['in('g('a)).'@('a,'node,'agent).'@('t,'node,'tc).T:Time].
 'undefined]
```

representing the event caused by an operation `'in('g('a))` emitted by agent `'a` on tuple centre `'t` at time `T`.

The result of the execution of `search` is shown in Figure 5. The execution trace reported in Figure 5 makes it clear that the tuple centre can reach only one final state for the execution of *in-all*, proving that the program features a terminating behaviour. Furthermore, this final state demonstrates that the *in-all* program guarantees the removal of all the `'g('a)` tuples, since the final tuple-set is `[nilMT]`, where `nilMT` represents the empty multiset of tuples.

4.2 Distributed Dining Philosophers

A first ReSpecT-based solution to the *dining philosophers* problem has already been presented in [12]. Here, we focus on an extension called *distributed dining philosophers*. The main difference from the original Dining Philophers is concerned with the distributed nature of the problem: indeed, in this extended version both the philosophers and the resources (seats and table) are distributed. An A&A ReSpecT solution to this new problem has already been shown in [11].

We used the A&A ReSpecT executable specification to model, run and verify the correctness of this solution: the `DINING-PHILO` module, extending `RESPECT-SEMANTICS`, is shown in Figure 6. As clearly visible from this Figure, `DINING-PHILO` defines two distinct sets of reactions: one specifying the behaviour

```
Solution 1 (state 243)
states: 244  rewrites: 2835 in 90ms cpu (479ms real) (31500 rewrites/second)
D:DistributedState -->
 < '@('tc1,'node,'tc) # noReactionExec | [nilMT] |
 ['spec @ reactionSet] | ['trig @ nilMT] | ['input @ emptyQ] |
 ['output @
    [C:Cause.['in('g('a)).'@('a,'node,'agent).'@('t,'node,'tc).T:Time].'g('a)]
 ]
 >

 No more solutions.


Maude> show search graph .

state 0, TCState: < '@('tc1,'node,'tc) # noReactionExec |
 [('g('a)) | ('g('a)) | ('g('a)) | ('g('a)) | ('g('a)) | ('g('a))] |
 ['spec @ reactionSet] | ['trig @ nilMT] |
 ['input @
  [C:Cause.['in('g('a)).'@('a,'node,'agent).'@('t,'node,'tc).Tim:Time].'undefined]] |
 ['output @ Out:EvtQueue] >
arc 0 ===> state 1 (...)

state 1, TCState: < '@('tc1,'node,'tc) # noReactionExec |
 [('g('a)) | ('g('a)) | ('g('a)) | ('g('a)) | ('g('a)) | ('g('a))] |
 ['spec @ reactionSet] | ['trig @ nilMT] |
 ['input @ emptyQ] | ['output @ Out:EvtQueue] |
 [C:Cause.['in('g('a)).@('a,'node,'agent).'@('t,'node,'tc).Tim:Time].'undefined]>
arc 0 ===> state 2 (...)

state 2, TCState: < '@('tc1,'node,'tc) # noReactionExec |
 [('g('a)) | ('g('a)) | ('g('a)) | ('g('a)) | ('g('a))] | ['spec @ reactionSet] |
 ['trig @ 'reaction(('in('g('a))),('g('response,'from-agent)),('r('out('h('b))))))] |
 ['input @ emptyQ] |
 ['output @ Out:EvtQueue,
 [C:Cause.['in('g('a)).'@('a,'node,'agent).'@('t,'node,'tc).Tim:Time].'g('a)]]>
arc 0 ===> state 3 (...)

...

state 243, TCState: < '@('tc1,'node,'tc) # noReactionExec | [nilMT] |
['spec @ reactionSet] | ['trig @ nilMT] | ['input @ emptyQ] |
['output @ Out:EvtQueue,
[C:Cause.['in('g('a)).'@('a,'node,'agent).'@('t,'node,'tc).Tim:Time].'g('a)]] >
```

Fig. 5. Execution trace of the `search` command on the *in-all* program.

of the *table* tuple centre, the other defining the behaviour of each *seat* tuple centre. Adopting this program, we ran the `search` command on the following tuple-centre multiset:

```
< '@(('seat('1,'2)),'node1, 'tc) # ['spec @ reactionSeat] |
    ['philo('thinking)] |
    ['trig @ nilMT]  |
    [ 'input @ ['out('wanna-eat).-.-.-] ]  |
    ['output @ emptyQ]  | noReactionExec >
 |
< '@(('seat('2,'3)),'node2, 'tc) # ['spec @ reactionSeat] |
    ['philo('thinking)] |
    ['trig @ nilMT]  |
    [ 'input @ ['out('wanna-eat).-.-.-] ]  |
    ['output @ emptyQ]  | noReactionExec >
 |
< '@(('seat('3,'1)),'node3, 'tc) # ['spec @ reactionSeat] |
```

```
    ['philo('thinking)] |
    ['trig @ nilMT]  |
    [ 'input @ ['out('wanna-eat).-.-.-] ]  |
    ['output @ emptyQ]  | noReactionExec >
 |
 < '@('table,'node4, 'tc) #  ['spec @ reactionTable]  |
    [('chop('1)) | ('chop('2)) | ('chop('3))] |
    ['trig @ nilMT] | noReactionExec |
    ['input @ emptyQ]
    ['output @ emptyQ] >
 =>!
 D:DistributedState .
```

The above syntax describes a set of tuple centres representing a table, three seats and three philosophers who all request to eat . The three `'chop('n)` tuples within the `'table` tuple centre represent three chops available to the three philosopher agents seated around the table. To eat, each philosopher has to get two chops: the A&A ReSpecT specification shown in Figure 6 guarantees the avoidance of deadlocks. If we consider a table with N chops and N philosophers, a deadlock occurs when the N philosophers own a chop each. Accordingly, in the instance considered here, a deadlock occurs when the three philosophers own a chop each.

Furthermore, the test previously shown, starting from an initial state where the three philosophers are thinking, simulates a situation in which all the philosophers emit an eating request at the same time: the `'seat('1,'2)`, `'seat('2,'3)` and `'seat('2,'3)` philosophers express the desire to eat at the same time.

Executing `search`, we obtained the results presented in Figure 7. Results make it clear that the *distributed dining philosophers* program behaves as expected. Indeed, the `search` execution shows only three possible solutions: *(i)* one featuring the acquisition of chops `'1` and `'2` by the `'seat('1,'2)` philosopher, *(ii)* one featuring the `'seat('2,'3)` philosopher getting the lock on chops `'2` and `'3`, and *(iii)* the last featuring the `'seat('1,'3)` philosopher getting the lock on chops `'1` and `'3`.

Since the execution of `search` explores the whole space of the reachable states, it is straightforward to test the correctness of specific properties of an A&A ReSpecT program, verifying not only all the admissible solutions, but also the termination property of the program itself.

5 Conclusion

In this paper we demonstrate how to bridge the gap between formal definition and actual implementation of programming languages by exploiting Maude to model the A&A ReSpecT coordination language. The resulting specification represents *de facto* an executable prototype of A&A ReSpecT. The A&A ReSpecT executable specification is used to generate an execution machine for two A&A ReSpecT programs that address two coordination problems: the *in-all* and *distributed-dining-philosophers* problems. The results obtained by the execution of such programs

```
mod DINING-PHILO is including RESPECT-SEMANTICS .

  ops reactionSeat reactionTable : -> MultiTerm .
  eq reactionSeat = (
        'reaction( 'out('wanna-eat),
                   'g('request, 'from-agent),
                   'r( 'in('philo('thinking)),
                       'out('philo('waiting-eat)),
                       'event-target('@(('seat(v('C1),v('C2))),v('C3), 'tc)))
                       '?( ('@('table,'node4, 'tc)), ('in('chops(v('C1),v('C2)))))
        )   |
        'reaction( 'out('wanna-eat),
                   'g('response, 'from-agent),
                   'r( 'in('wanna-eat))
        )  |
        'reaction(  'out('wanna-think),
                   'g('response, 'from-agent)
                   'r( 'in('wanna-think)))
        )   |
        'reaction( 'in('chops(v('C1),v('C2))),
                   'g('response, 'inter, 'endo),
                    'r('in('philo('waiting-eat)),
                       'out('philo('eating)),
                       'out('chops(v('C1),v('C2))))
        )  |
       'reaction( 'out('chops(v('C1),v('C2))),
                  'g('response, 'inter, 'endo),
                  'r( 'in('philo('waiting-think)),
                      'out('philo('thinking)))
        )  |
        'reaction( 'out('wanna-think),
                   'g('request, 'from-agent),
                   'r( 'in('philo('eating)),
                       'out('philo('waiting-think)),
                       'event-target('@(('seat(v('C1),v('C2))),v('C3), 'tc)),
                       'in('chops(v('C1),v('C2))),
                       '?( ('@('table,'node4, 'tc)),('out('chops(v('C1),v('C2)))))))
        ) .
  eq reactionTable = (
        'reaction( 'out('chops(v('C1),v('C2))),
          'g('response, 'from-tc),
          'r( 'in('chops(v('C1),v('C2))), 'out('chop(v('C1))), 'out('chop(v('C2))) )
         )  |
        'reaction( 'in('chops(v('C1),v('C2))),
                   'g('request, 'from-tc),
                   'r( 'out('required(v('C1),v('C2))))
          )  |
        'reaction( 'in('chops(v('C1),v('C2))),
                   'g('response, 'from-tc),
                    'r( 'in('required(v('C1),v('C2))))
          )  |
        'reaction( 'out('required(v('C1),v('C2))),
          'g('request, 'intra, 'endo),
          'r( 'in('chop(v('C1))), 'in('chop(v('C2))), 'out('chops(v('C1),v('C2))) )
           )  |
         'reaction( 'out('chop(v('C))),
                   'g('response, 'intra, 'endo),
                    'r( 'rd('required(v('C),v('C2))),
                          'in('chop(v('C))), 'in('chop(v('C2))),
                          'out('chops(v('C),v('C2))))
          )  |
         'reaction( 'out('chop(v('C))),
                    'g('response, 'intra, 'endo)
                     'r( 'rd('required(v('C1),v('C))),
                          'in('chop(v('C1))), 'in('chop(v('C))),
                           'out('chops(v('C1),v('C))))
          ) .

 endm
```

Fig. 6. Definition of the *distributed dining philosophers* program.

allow for the validation of the behaviour of the implemented coordination policies.

As already shown in previous works, MAUDE proves itself to be a suitable tool for the rapid prototyping of programming languages, including those not belonging

```
Solution 1 (state 12915)
states: 13209  rewrites: 2842405 in 32870ms cpu (34607ms real)(86474 rewrites/second)
D:DistributedState -->
< '@('table,'node4,'tc) #  [('chop('3))|('required('2,'3))|('required('3,'1))] |... >
  |
< '@(('seat('1,'2)),'node1,'tc) # [('chops('1,'2)) | ('philo('eating))] | ... >
  |
< '@(('seat('2,'3)),'node2,'tc) # ['philo('waiting-eat)] | ... >
  |
< '@(('seat('3,'1)),'node3,'tc) # ['philo('waiting-eat)] | ... >


Solution 2 (state 13018)
states: 13209  rewrites: 2847117 in 33100ms cpu (35370ms real)(86015 rewrites/second)
D:DistributedState -->
< '@('table,'node4,'tc) #  [('chop('1))|('required('1,'2))|('required('3,'1))] |... >
  |
< '@(('seat('1,'2)),'node2,'tc) # ['philo('waiting-eat)] | ... >
  |
< '@(('seat('2,'3)),'node1,'tc) # [('chops('2,'3)) | ('philo('eating))] | ... >
  |
< '@(('seat('3,'1)),'node3,'tc) # ['philo('waiting-eat)] | ... >


Solution 3 (state 13138)
states: 13209  rewrites: 2945119 in 34150ms cpu (36480ms real)(86015 rewrites/second)
D:DistributedState -->
< '@('table,'node4,'tc) #  [('chop('2))|('required('1,'2))|('required('2,'3))] |... >
  |
< '@(('seat('1,'2)),'node2,'tc) # ['philo('waiting-eat)] | ... >
  |
< '@(('seat('2,'3)),'node1,'tc) # ['philo('waiting-eat)] | ... >
  |
< '@(('seat('3,'1)),'node3,'tc) # [('chops('3,'1)) | ('philo('eating))] | ... >
```

Fig. 7. Result of the execution of the `search` command on the *distributed dining philosophers* program.

to traditional paradigms—like imperative, functional and logic ones. Indeed, even though A&A ReSpecT is defined by a logic-like syntax, it also features a different, event-driven computational model based on the concept of *reaction*.

We intend to further explore the use of Maude as a tool to prototype the operational semantics of programming languages. In particular, we plan to keep on experimenting with A&A ReSpecT, by applying model-checking techniques in order to prove linear temporal logic (LTL) properties on our executable specification. This will also allow us to verify the satisfaction of *safety* and *liveness* properties that are crucial in the development of A&A ReSpecT coordination policies.

References

[1] Busi, N., R. Gorrieri and G. Zavattaro, *Three semantics of the output operation for asynchronous communication*, in: D. Garlan and D. Le Métayer, editors, *Coordination Languages and Models*, LNCS **1282**, Springer-Verlag, 1997 pp. 205–219.

[2] Busi, N., R. Gorrieri and G. Zavattaro, *A process algebraic view of Linda coordination primitives*, Theoretical Computer Science **192** (1998), pp. 167–199.

[3] Busi, N. and G. Zavattaro, *On the serializability of transactions in JavaSpaces*, Electronic Notes in Theoretical Computer Science **54** (2001), pp. 92–105.

[4] Cabri, G., L. Leonardi and F. Zambonelli, *MARS: A programmable coordination architecture for mobile agents*, IEEE Internet Computing **4** (2000), pp. 26–35.

[5] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, "Maude Manual," Department of Computer Science, University of Illinois at Urbana-Champaign, 2.2 edition (2005). URL http://maude.cs.uiuc.edu

[6] Denti, E., A. Natali and A. Omicini, *On the expressive power of a language for programming coordination media*, in: *1998 ACM Symposium on Applied Computing (SAC'98)* (1998), pp. 169–177, special Track on Coordination Models, Languages and Applications.

[7] Gelernter, D., *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[8] Gelernter, D., *Multiple tuple spaces in Linda*, in: *Parallel Architectures and Languages Europe (PARLE'89)*, LNCS **366** (1989), pp. 20–27.

[9] Martí-Oliet, N. and J. Meseguer, *Rewriting logic: roadmap and bibliography*, Theoretical Computer Science **285** (2002), pp. 121–154.

[10] Mousavi, M. R., M. Sirjani and F. Arbab, *Formal semantics and analysis of component connectors in Reo*, Electronic Notes in Theoretical Computer Science **154** (2006), pp. 83–99.

[11] Omicini, A., *Formal* ReSpecT *in the A&A perspective*, Electronic Notes in Theoretical Computer Sciences **175** (2007), pp. 97–117, 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.

[12] Omicini, A. and E. Denti, *Formal* ReSpecT, Electronic Notes in Theoretical Computer Science **48** (2001), pp. 179–196, declarative Programming – Selected Papers from AGP 2000, La Habana, Cuba, 4–6 December 2000.

[13] Omicini, A. and E. Denti, *From tuple spaces to tuple centres*, Science of Computer Programming **41** (2001), pp. 277–294.

[14] Omicini, A. and F. Zambonelli, *Coordination for Internet application development*, Journal of Autonomous Agents and Multi-Agent Systems **2** (1999), pp. 251–269.

[15] Papadopoulos, G. A. and F. Arbab, *Configuration and dynamic reconfiguration of components using the coordination paradigm*, Future Generation Computer Systems **17** (2001), pp. 1023–1038.

[16] Peltz, C., *Web Services orchestration and choreography*, IEEE Computer **36** (2003), pp. 46–52.

[17] Plotkin, G., *A structural approach to operational semantics*, Technical Report DAIMI FN-19, Department of Computer Science, AArhus University, Denmark (1991).

[18] ReSpecT *home page*.
URL http://respect.alice.unibo.it