Electronic Notes in Theoretical Computer Science 82 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 16 pages

CoordMaude: Simplifying Formal Coordination Specifications of Cooperation Environments [⋆]

Marisol Sánchez-Alonso
Pedro J. Clemente, Juan M. Murillo and Juan Hernández

QUERCUS Software Engineering Group, Computer Science Department, University of Extremadura, Spain
{marisol, jclemente, juanmamu, juanher}@unex.es

Abstract

Developing concurrent applications in cooperative environments is an arduous task. This is mainly due to the fact that it is very difficult to specify the synchronized interaction between the entities composing the system. Using coordination models makes this task easier. The latest trends in this area suggest that to manage the successful implementation of complex systems, coordination models must support some key features regarding the coordination constraints: their separated specification, their unanticipated evolution and their dynamic change. However, supporting these features is not only a technical challenge: it must be also guaranteed that the application of a separately specified coordination pattern to a set of encapsulated entities, or the change of the coordination constraints in an already running software system will not produce semantic errors. This is just the problem focused in this paper. In particular, a method for generating formal interpretable specifications reproducing coordinated environments is presented. The method is based on the Coordinated Roles coordination model and makes use of Maude as a formal language. The benefits obtained are: (i) easy specification using the coordination model syntax, (ii) automatic generation of the corresponding formal specification and (iii) simulation of system behaviour.

1 Introduction

The increasing number of requests for the development of complex software systems that support multiple services, and adopt new technologies, in distributed environments, has augmented the need for cooperation and coordination between all the entities concerning the system. The main task in the design of such applications is to specify the coordination constraints existing

⋆ This work has been supported by the project CICYT under grant TIC02-04309-C02-01 among the components. All this, joined to the last trends[1,2,3,4] promoting the reuse of components to attain an agile development process, has given an impulse to the appearance of coordination models and languages to facilitate this task. These models comply with the following goals:

- Providing enough expressiveness to specify all kind of coordination problems.
- Promoting the reusability of software components and coordination patterns independently.
- Giving support to Open Systems.
- Allowing the dynamic changes of coordination policies.

Coordination models can be classified depending on how the coordination constraints are expressed [2]. According to this classification the models can be:

- *Endogenous.* When the coordination is expressed inside the code of the elements and the communication is realized by means of a common tuple space. The tuple space avoids the need for the entities that are to be coordinated to know one another explicitly, facilitating the appearance and disappearance of elements in the system. However, these elements must insert the appropriate coordination primitives in their code, before being incorporated to the system. This makes it difficult to reuse the entities and the coordination pattern.
- *Exogenous.* When the coordination tasks are performed by an entity that is external to the entities to be coordinated, making a distinction between:
  - *No transparent.* Allowing the reusability of the coordination pattern, but making explicit the coordination from the entities to be coordinated, affecting their reusability.
  - *Transparent.* Promoting the coordinator entity performs the communication between the entities to be coordinated, without their explicit collaboration; resolving the problem of their reusability.

All the models that promote the separation between functional and coordination aspects provide the above goals with better or worse expressiveness. However, less attention has been paid to guarantee that the final behaviour obtained in the composed application is semantically coherent, that is:

(i) How can it be guaranteed that gluing together a coordination policy and a set of components (that have been coded separately) in an application will produce the expected behaviour?

(ii) Moreover, supposing that the expected behaviour is produced, how can it be guaranteed that adding new coordination constraints, or that changing the coordination constraints of an application will not produce conflicts with the current behaviour?

With the aim of avoiding these deficiencies, a method of generating formal interpretable specifications is presented in this paper. This method allows not only the specification and detection of inconsistencies when composing applications or when changing their coordination policies, but also the simulation of the global behaviour of such applications by means of executing the formal specifications.

The method is based on the formal specification language *Maude* [5] and a coordination model. The choice of *Maude* was motivated by the fact that it is interpretable, concurrent and able to support object definitions. The approach presented in this paper employs *Coordinated Roles* (CR from now on)[3]. However, the method could be easily adapted to any other exogenous coordination models [4,6].

The use of formal languages allows the checking of the syntactic and semantic correctness of the system and the simulation of the system behaviour executing the formal specifications. However, formal specifications can be large and complex [7], hindering understanding of the system representation and making it difficult to focus on the relevant features of the specification.

In order to facilitate the description of coordination constraints in cooperation systems, adopting the CR coordination model and taking advantage of the executable *Maude* formal language, we have developed *CoordMaude.*

CoordMaude is a set of *Maude* primitives allowing the use of CR syntax to describe the objects and the dependency relations of coordination environments, and generating the equivalent formal specification in *Maude.* This results in shorter and simpler specifications; it focuses on the coordination features and the abstraction of formalization details, while allowing the execution of the specifications generated to simulate the system behaviour.

The paper structure is as follows: In section 2 the motivations underlying this work are presented by means of an example. The representation of *CR* in *Maude* is described in section 3. Section 4 presents the set of primitives constituting *CoordMaude* and its objectives. Related work is mentioned in section 5. Section 6 explains future work and work in progress. Finally, section 7 presents conclusions and next, the references.

2 Motivations

To illustrate the problem faced in this paper, an example is shown. The example is inspired by the case study described in [3]. This example presents a system to control the access to a car park. The car park has a barrier and a ticket machine. The desirable car park behaviour is as follows: when a sensor detects a car passing, it sends a message to the ticket machine, invoking the *Give* action to supply an entry ticket. This action produces a ticket and there is a sensor detecting if the ticket is collected. There is another sensor before the barrier. When the sensor detects the presence of a car, it sends a message to the barrier, invoking the *Rise* action. However, the barrier must not execute the *Rise* action if the ticket machine has not finished the *Give* action. It is necessary to impose this constraint, because if the ticket machine cannot execute the *Give* action (e.g., there is no paper), or a ticket is produced but not collected, a car could pass.

2.1 Coordinated Roles

In [3], CR is proposed as a coordination model inspired on *IWIM* model [6]. CR is an exogenous and transparent model based on the *Event Notification Protocols* (ENP) mechanism, which allows a coordinator component to ask for the occurrence of an event in another component. This process is transparent to the components to be coordinated. The notification can be asked for in a *synchronous* and in an *asynchronous* way. The events for which notification can be requested are the reception of a message (*RM event*), the beginning of the processing of a message (*BoP event*), the end of the processing of a message (EoP event) and the fact that the notifier has reached a particular abstract state (*SR event*).

Each coordination component imposes a coordination pattern. Each coordination pattern is structured as a set of roles. A role represents each of the characters that can be played in a coordination pattern. Behaviour components will have to adopt these roles in order to be coordinated. For each role, coordination components specify the set of events required to specify the desired coordination constraints.

There can be a functional component adopting several roles and roles adopted by other coordinators defining a hierarchy of coordinators in this way. The coordination components never make direct reference to the monitored components. The binding between coordinators and components to be coordinated is done at run-time via composition syntax.

2.2 Behavioural problems

The above example can be solved with *CR* coding a coordinator component that serializes the execution of any pair of operations. Then, when the instances of the *Ticket_Machine* object and the *Barrier* object and the coordinator have been created, the binding is made between the *Give* method and the first operation of the coordinator, and the *Rise* method and the second operation of the coordinator.

As the three components have been coded separately, with no references between them, it cannot be anticipated whether the constraints imposed by the coordinator will be coherent or not with the internal behaviour of the objects. So, the global behaviour of the application will be unpredictable. For example, it is supposed that in the car park application there is a third object that, when the application is launched, tests everyday if the ticket machine and the barrier work properly. With this purpose it sends *Give* and *Rise* messages, respectively, provided that this object has been coded in order to test first the barrier and then the ticket machine. Under these conditions, although nothing has changed in the *Ticket_Machine* or the *Barrier* components, a deadlock occurs when they are bound to the coordinator, because the coordinator will not allow the execution of *Rise* if *Give* has not been executed previously.

Moreover, even supposing that it has been demonstrated that the constraints of the coordinator are coherent with the behaviour of the objects and, thus, that the application works correctly, nothing can be anticipated if new coordination constraints are introduced or if the current coordination constraints are changed.

These problems are detected in all the exogenous coordination models and have been outlined by other authors [4, 8]. In the next section, the representation of *CR* in *Maude* is presented in order to validate the coordinated behaviour by executing the system specifications.

3 Maude specification of Coordinated Roles

In this section, the way in which the functional and coordination components of *CR* are formally specified in *Maude* is presented. The coordination mechanisms defined by the coordination model must also be specified in *Maude*. The *Maude* interpreter can execute the formal specification generated. In this way, the execution of the formal representation allows the simulation of the coordinated system, in order to be validated, detecting mismatches and inconsistencies. *Maude* also provides trace and debugger mechanismes.

First, in this section, the *Maude* formal language and the motivations of its use in this context are briefly outlined. Next, the correspondence between *CR* and *Maude* is shown by means of the example previously presented.

3.1 Maude specifications

Maude is an executable algebraic language based on rewriting logic. The language allows both functional and object-oriented specifications in a concurrent and non-deterministic way. *Maude* specifications can be executed by means of its rewrite engine, which facilitates its use for prototyping and for checking the specification behaviour[9].

Maude is divided into two levels: *Core Maude* and *Full Maude. Core Maude* contains the basic syntax of the language allowing the definition of functional and system modules. Operations and equations can be defined in both kinds of modules. In system modules, rewrite rules can also be defined. The execution of specifications is performed by means of reducing terms by equations and rewrite rules. *Full Maude* is developed on *Core Maude*, and extends *Maude* with the necessary syntax to define object-oriented modules. In these modules the rewrite rules are interpreted as state transition rules of the object classes defined in them. *Full Maude* also provides the use of parameterised modules. The parameterised modules take other modules as arguments. The structure of the parameters is described by theories, representing the schema which the modules as arguments must adapt to. To bind theories with specific modules to be passed as arguments, views are defined. This mechanism is used to represent the correspondence of *CR* in *Maude*.

An important capability provided by the language is reflection[10]. Reflection allows the handling of terms and language constructions as arguments in operations, and the increase in the syntax of the languages defining new operations over language terms. In fact, *Full Maude* is defined in relation to *Core Maude* making use of this capability. The definition of *CoordMaude*, that is presented in the next section, has also been accomplished making use of this capability. The clarity of the language, its wide range of application, its executability and its reflection capability have been decisive to select *Maude* as the formal language in our method.

3.2 Representing Coordinated Roles in Maude

In the introduction, it is mentioned that the approach can be adapted to the use of any other exogenous coordination model. In order to achieve this, some general rules must be observed:

- Both functional and coordination components are represented by specification modules. In particular, specification modules representing coordination components are parameterized, in which the parameters represent the coordinated components.
- The use of theories will allows the coordinators to access the features of coordinated components without making explicit reference to them. In this way, the reusability of the coordinator modules is guaranteed.
- Views will be used to make the binding between the coordinator parameters and the components to be coordinated.
- In order to implement the coordination constraints, rewriting rules are used in the specification modules representing coordinator components. These rewriting rules specify the dependencies between the operations performed by the system.

In particular, the parallelism between the coordination mechanisms of *CR* and the mechanisms of *Maude* used to specify them are shown in figure 1.

The objects to be coordinated are represented in *Maude* as object modules, and the coordinators are represented by means of parameterised modules, where the parameters are defined by means of theories. The theories, in the *Maude* representation, take the function of the roles in *CR*, that is, the syntactical schema to be satisfied by the object modules representing the objects to coordinate. The compositional syntax of *CR* allows the binding of each object to coordinate with the roles declared in the coordinator. That is represented in *Maude*, making use of views. Views associate specific object modules with the abstract schema represented by the theories.

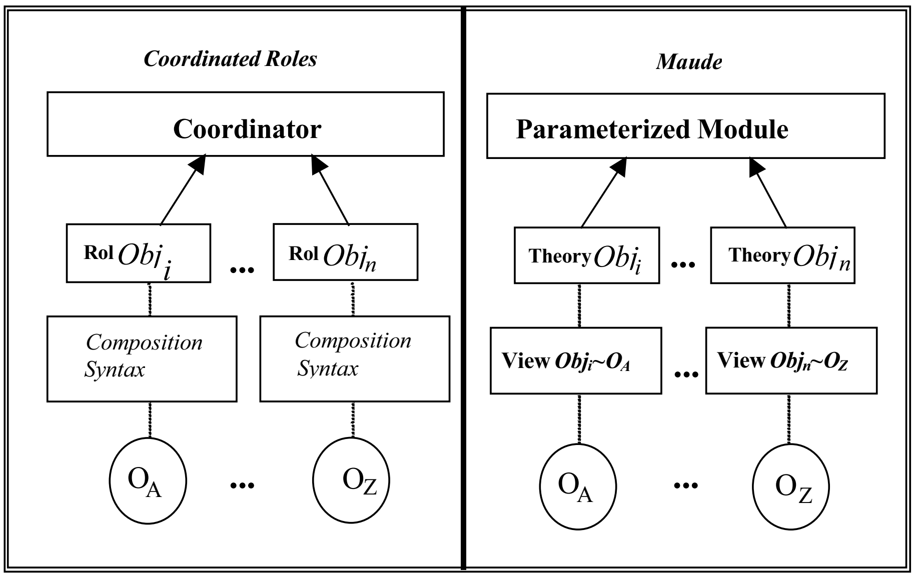

Fig. 1. Correspondence between CR and Maude

To show how the different coordination mechanisms of CR can be specified in *Maude*, the car park example is represented. Figure 2 shows the specification of the *Ticket-machine* and *Barrier objects*, represented as object modules, when

- The classes and messages of the respective objects are declared.
- The *total-requestE, ACK-receivedE, total-requestB* and *ACK-receivedB* attributes are included in the class definition to represent the synchronization mechanisms of the event notification.
- The conditional rules represent the behavior of both objects when receive the Give and Rise messages respectively.

If the condition is satisfied and the object state matches the left side of the rule, the transition will occur. Then the new object state will be represented by the right side of the rule.

In figure 3, the theories *OBJECT1* and *OBJECT2* expressing the abstract schema that the *Ticket-machine* and the *Barrier* must satisfy respectively are defined. These theories correspond with the roles defined in the coordinator.

The coordinators in *CR* must specify the event notifications required regarding the objects adopting their roles. By means of rewrite rules, the different event notification protocols are specified in *Maude.* These rules ask about and modify the attributes declared in the objects to represent such a mechanism.

In particular, *Receipt of* a *Message* event (RM) needs three attributes to have been declared for the object to be coordinated: *message_received*, *total_request* and *ACK_received.* The *Message_received* attribute informs the coordinators that the object has received a message. The *Total_request* attribute indicates the number of coordinators requesting the event notification,

```
(omod TICKET-MACHINE is...
  class ticket-machine | action : Qid, total-requestE : MachineInt,
                         ACK-receivedE : MachineInt, total-requestB : MachineInt,
                         ACK-receivedB : MachineInt .
  msg give : Oid -> Msg .
  ...
  crl[Give] : (give(M) )
      < M : ticket-machine | action : 'nil,  total-requestE : N, ACK-receivedE : T >
  => < M : ticket-machine | action : 'give, ACK-receivedE : 0 > if N = = T .
endom)

(omod BARRIER is ...
  class barrier | action : Qid, total-requestB : MachineInt, ACK-receivedB : MachineInt .
  msg Rise : Oid -> Msg .
  ...
  crl[Rise] : (rise(B) )
          < B : barrier | action : 'nil, total-requestB : N, ACK-receivedB : T >
  =>  < B : barrier | action : 'rise, ACK-receivedB : 0 >  if N = = T .
endom)
```

Fig. 2. Object module specifications of Car Park example

```
(oth OBJECT1 is...
  class Object1 | element : Qid, total-requestE : MachineInt,
                            ACK-receivedE : MachineInt, total-requestB : MachineInt,
                            ACK-receivedB : MachineInt .
  msg MObj : Oid -> Msg .
endoth)

(oth OBJECT2 is...
  class Object2 | element : Qid, total-requestB : MachineInt,
                   ACK-receivedB : MachineInt .
  msg MObj2 : Oid -> Msg .
endoth)
```

Fig. 3. Theory specifications of Car Park example

and the *ACK_received* attribute counts the number of coordinators that have processed the event notification.

To specify *Beginning of processing event (BoP)*, *total_request* and *ACK_received* attributes are required with the same purpose described above.

Three attributes: *total_request, ACK_received* and action are required to specify *End of processing event (EoP)*: Action attribute is used to register the last message processed by the object. The rest of the attributes have the same function explained in the two previous events.

Last, the *State Reached event (SR)* requires that the coordinated object declare an attribute informing when the desirable state is obtained.

The coordinator in the example is represented as a parameterised module in figure 4. The coordinator named *SERIALIZER* has two parameters (X and Y), whose types are the *OBJECT1* and *OBJECT2* theories, respectively. The purpose of this coordinator is to serialize the execution of the operation of object X and the execution of an operation in object Y. With this purpose,

```
(omod  SERIALIZER [ X :: OBJECT1, Y :: OBJECT2 ] is ...
  class serial [X, Y ] | notificated : Bool, oper-executed? : Bool, list : QidList .
  vars ...
  crl [EoP1] : < S : serial[X, Y] | notificated : false, oper-executed? : false >
        < M : Object1.X | element : Q , ACK-receivedE : N >
  =>      < S : serial[X, Y] | notificated : true  >
  < M : Object1.X | ACK-receivedE :  N + 1 >  if Q =/= 'nil .
  crl [EoP2] :  < S : serial[X, Y] | notificated: true,  oper-executed? : false >
  < M : Object1.X | total-requestE : N,  ACK-receivedE : T >
  => < S : serial[X, Y] |  oper-executed? : true >
  < M : Object1.X | ACK-receivedE : 0 >    if N == T .
  crl [Trace1] : < S : serial[X, Y] | oper-executed? : true , list : QL >
  < M : Object1.X | element : Q >
  => < S : serial[X, Y] | notificated : false , list : QL Q >
  < M : Object1.X | element : 'nil >  if Q =/= 'nil .
  rl [BoP] :  < B : Object2.Y | element : 'nil ,  ACK-receivedB : N >
  < S : serial[X, Y] | oper-executed? : true , notificated : false >
  => < S : serial[X, Y] | oper-executed? : false  >
  < B : Object2.Y |  ACK-receivedB : N + 1 > .
  crl [Trace2] :   < B : Object2.Y | element : Q >
  < S : serial[X, Y] | list : QL >
  => < B : Object2.Y | element : 'nil >
  < S : serial[X, Y] | list : QL Q >       if Q =/= 'nil .
endom)
```

Fig. 4. Parameterised module representing coordinator in *Car Park* example

SERIALIZER asks about the asynchronous *EoP* event notification of the operation in *X* and the synchronous *BoP* event notification of the operation in *Y*. The execution of the operation in *Y* is only allowed if the notification of the operation execution in *X* has been received. The rewrite rules labeled as *EoP1* and *EoP2* specify the required protocol to the asynchronous notification of *EoP* event. When this event is notified, in *SERIALIZER* module, the operation-executed? attribute is modified. The protocol for the synchronous notification of a *BoP* event is specified in the rewrite rule labeled *BoP*. *Trace1* and *Trace2* rules are used to keep in the list attribute the sequence of operations performed by the system.

The use of theories and views is the mechanism that establishes the binding between the modules representing the coordinated objects, and the parameterised module representing the coordinator. In this way, the coordinator module does not refer to the coordinated objects explicitly, getting transparency in the coordination and making the reusability of the coordinated objects and the coordination pattern easier.

Figure 5 shows *Ticket-dispatch* and *Barrier* views of the example. The first one establishes the binding between the elements specified in *TICKET-MACHINE* module and the elements specified in *OBJECT1* theory. The second one makes the same with the elements in *BARRIER* module and *OBJECT2* theory.

```
(omod PROOF1 is
  protecting SERIALIZER[Ticket-dispatch , Barrier] .
  op init : -> Configuration .
  var T : Oid .
  eq init =  < 'S : serial[Ticket-dispatch , Barrier] | notificated : false ,
                                                 oper-executed? : false , list : nil >
           < 'B : barrier | action : 'nil , total-requestB : 1 , ACK-receivedB : 0 >
           < 'M : tick-machine |  action : 'nil ,  total-requestE : 1 , ACK-receivedE : 0,
                        total-requestB : 0 , ACK-receivedB : 0 >
   ( give ('M) give ('M) rise('B) rise('B) give ('M) rise ('B) give ('M) rise ('B) ) .
 endom)

(rewrite init . )

Introduced module: PROOF1
rewrites: 239 in 120ms cpu (122ms real) (1991 rewrites/second)
rewrite in PROOF1 : init .
result Configuration :
 < 'B : barrier |  accion : 'nil , total-requestB : 1 , ACK-receivedB : 0 >
 < 'M : tick-machine |   action : 'nil ,  total-requestE : 1 , ACK-receivedE : 0,
                        total-requestB : 0 , ACK-receivedB : 0 >
 < 'S : serial[Ticket-dispatch , Barrier] | notificated : false , oper-executed? : false , list ::
( 'give 'rise 'give 'rise 'give  'rise 'give 'rise ) >
```

Fig. 5. Views of Car Park example

An instantiation example is presented in *PROOF1* module. In this module, shown in figure 6, a *coordinator object of serial class* and specific objects of *ticket-machine class* and *barrier class* are created. Init operation sets all the objects to initial configuration and defines a set of messages to test the coordinated behaviour of the system. The simulation of the system behaviour is performed by rewriting the *Init* operation. The result shows the system configuration after the execution of the messages and the list of messages processed. Different test cases can be simulated modifying the set and sequence of the messages to be performed by the system, in this way the system behaviour can be checked to detect inconsistencies and mismatches. If there are messages that cannot be processed, they will show in the result, before the system configuration.

3.3 Modifying coordination constraints

New components and coordination constraints can be added to the system maintaining the above behaviour for the rest of system components. In order to illustrate this feature, a new object and a new coordination constraint are added to the *Car Park* example. The new constraint dictates that no cars can access if the *Car Park* is full. To force this new constraint a traffic light is added to the entry. The traffic light will not allow new cars to pass if the car park is full. Moreover, if the traffic light is red new tickets will not be produced.

In order to control that tickets are not given if the traffic light is red, the new constraint must be added without interfering with the original behaviour

```
    (view Ticket-dispatch from OBJECT1 to TICKET-MACHINE is
        class Object1 to tick-machine .
        attr element . Object to action .
        attr ACK-receivedB . Object to ACK-receivedB .
        attr total-requestB . Object to total-requestB .
        attr ACK-receivedE . Object to ACK-receivedE .
        attr total-requestE . Object to total-requestE .
        msg MObj to give .
     endv)

    (view Barrier from OBJECT2 to BARRIER is
        class Object2 to barrier .
        attr element . Object to action .
        attr ACK-receivedB . Object to ACK-receivedB .
        attr total-requestB . Object to total-requestB .
        msg MObj2 to rise .
endv)
```

Fig. 6. Instantiation test in Car Park example

```
(omod SERIALIZER2[X :: OBJECT1, Y :: OBJECT2, Z :: IN-CONTROLER]  is
  protecting SERIALIZER[X, Y] .
  class serial2[X, Y, Z] | lock : Bool  .
  subclass serial2[X, Y, Z] < serial[X, Y] . ...
  rl [SR1] :       < S : serial2[X, Y, Z] |  lock : true >
  … (rules are not shown)
endom)
```

Fig. 7. Coordinator specification in Car Park with traffic light example

of the system. For this purpose, a new coordinator is specified. This coordinator must inherit the behaviour of the previous coordinator and must impose the new coordination constraint. The new coordinator represented as *SERIALIZER2* parameterised module is shown in figure 7 (the object module corresponding to the traffic light and the theory and the view associated to it are not shown, for brevity's sake).

The new coordinator has a parameter representing the *Z* object of *IN-CONTROLER* type. *IN-CONTROLER* is the theory representing the syntactical schema that the traffic light specification (in this case) must adopt.

In *SERIALIZER2*, *serial2* class is defined as a *serial* subclass from *SERIALIZER* module. This new class declares an attribute named *lock*, to indicate whether the *Car Park* cannot accept any car (where traffic light sets red and the last car has crossed the barrier). Now, the coordinator allows the operation associated to *X* object, depending on the value of *stop* attribute of *Z* object. New rules detect when the *stop* attribute modifies its value allowing the execution of the operation associated to *X*.

In figure 8, *PROOF2* module is shown. This module specifies a possible initial configuration of the system and a set of operations. The test represented

```
(omod PROOF2 is ...
 protecting SERIALIZER2[Ticket-dispatch, Barrier, C-park-TL] .
 op init2 : -> Configuration .
 var T : Oid .
 eq init2 =  < 'S : serial2[Ticket-dispatch, Barrier, C-park-TL] | lock : false,
                         notificated : false ,  oper-executed? : false , list : nil >
        < 'B : barrier | action : 'nil , total-requestB : 1 , ACK-receivedB : 0 >
        < 'M : tick-machine | action : 'nil , total-requestE : 1 , ACK-receivedE : 0,
                              total-requestB : 1 , ACK-receivedB : 0 >
        < 'P : traffic-light | action : 'nil , stop : false , capacity : 0 , max : 4  >
      (in('P) give('M) rise('B)) in('P) give('M) rise('B) out ('P) in('P) in('P) ...
endom)
```

Fig. 8. Test specification to Car Park with Traffic Light example

here creates a new coordinator instance, of *serial2* class, and the instances of *traffic-light, ticket-machine* and *barrier* classes. The interaction between the ticket machine and the barrier maintains the behaviour specified in *SERIALIZER*, and the interaction between the traffic light and the ticket machine is controlled, by the new rules specified in *SERIALIZER2.* Now, the operations accepted by the system are: in and out from the traffic light, *give* from the ticket machine and *rise* from the barrier.

4 CoordMaude

In the above section, the *CR* representation in *Maude* has been described. The example used is very simple but expressive enough to show how complex and obscure the system specification can become. The objects to be coordinated and the coordinators must be specified. Moreover, each one of the events to be notified must be expressed by means of rewrite rules, and theories and views have to be created for each coordinated object. The specification obtained is long and complex, and developers can be lost among all these mechanisms used to establish the correspondence between *CR* and *Maude.* With the aim to simplify this process and make the system specification easier to understand, we have defined *CoordMaude.*

CoordMaude is a set of primitives developed in *Maude*, allowing the specification of the system by means of *CR* syntax. The primitives have been defined making use of the reflection capability of *Maude* [10]. The mechanisme has two step: the specification step and the instantiation step. In the first, the objects to be coordinated, the coordinators and their roles are specified using the *CR* syntax. The result of applying the primitives provided by *CoordMaude* is the representation of the system specification containing: the object modules corresponding to the coordinated objects, the parameterised modules corresponding to the coordinators with the events to be notified represented as rewrite rules, the theories corresponding to the roles defined in the coordinators, and the views representing the compositional syntax of *CR.*

```
Class 'Ticket-Machine          methods [ 'Give ]
            Oper 'Give = return 'Give ; .
Class 'Barrier         methods [ 'Rise ]
            Oper 'Rise = return 'Rise ; .
Coord 'Serializer
            Def Rol  'Obj1 methods 'MObj1 ; 'Obj2 methods 'MObj2 ;
            Def Event Asyn EoP Event 'MObj1_Terminated for 'MObj1
                                                    Exec 'Termination_MObj1 ;
                        Syn BoP Event 'Request_MObj2 for 'MObj2
                                                    Constr 'MObj1_Processed? ;
            Oper  'Termination_MObj1 = 'MObj1_Processed = 1 ;
            Oper   'MObj1_Processed? =( 'Aux = 'MOjb1_Processed ;
                     ( 'MObj1_Processed = 0 ; return 'Aux ; ) )  .
```

Fig. 9. CoordMaude specification of car park example

```
'TM1 = ActiveObject ( 'Ticket-Machine ) ,
'B1 = ActiveObject ( 'Barrier )
( 'CarPark . AddObject ( 'TM1 to 'Obj1 with ('Give as 'MObj1 ) )
  'CarPark. AddObject ( 'B1 to 'Obj2 with ( ( 'Rise as 'MObj2 ) ) ) ) .
```

Fig. 10. Instantiation of car park example in CoordMaude

All these modules are automatically generated. The system behaviour can be simulated then, providing a configuration of object instances and a set of operations to be executed, which will be reinterpreted.

In figure 9, the initial *Car Park* example is taken again, to show the *Coord-Maude* specification of the objects to be coordinated, representing the ticket machine and the barrier and the *Serializer* coordinator.

The interpretation of the Car Park *CoordMaude* specification generates:

- The object modules corresponding to the ticket machine and the barrier, where the attributes needed for the coordination (explained in section **??**) are added to each object class automatically.
- The parameterised module representing the serializer, coordinating the above objects.
- The theories corresponding to the parameters in the parameterised module, representing the two roles imposed by the coordinator.

The second step of the mechanisme used by *CoordMaude* specifies the instances of active objects and coordinators constituting the system configuration. The views to bind the theories to the corresponding object modules are generated in the instantiation step. An example of the system configuration is shown in figure 10.

The *CoordMaude* example greatly simplifies the system specifications, adopting *CR* syntax. This facilitates the understanding of the specification, avoiding handling the mechanisms provided to represent the system features formally and reducing the system specification significantly..

Another advantage of this representation is that allows the changing of coordination policies easily. Moreover, the specification can be executed, by means of the *Maude* interpreter, simulating the system behaviour as the previous representation.

5 Related work

Although the approach explained in this work has chosen *Maude* as a formal language, there are other similar algebraic specification languages. OBJ3[11] has very similar features. In fact, *Maude* contains OBJ3 as a sublanguage and can interpret OBJ3 modules. CafeOBJ[12] also has a lot of similarities to Maude. Both of them support object oriented specification and rewriting logic. However, *Maude* provides a good framework to use the reflection capability that is the basis on which our approach is developed.

Some research work[4,8] has also detected the problem that this paper deals with. The problem was observed in [4], although no adequate solution has been suggested. In [8], the critical challenge faced by software developers when trying to ascertain whether system components are correctly integrated is highlighted, providing a method mainly focused on early detection of deadlock situations.

The work presented in [13] must also be mentioned. In this work, active systems are specified using LOTOS and SDL. The goal is to generate prototypes of Java programs from specification modules. As is explained in the next section our research group has automatic code generation as a common goal.

Other approaches [14,15,16] also propose the animation or specification execution to validate system behaviour, but need the translation of the specification to a programming language to be executed. That can provoke a lack of precision and fidelity between both representations, due to the different abstraction levels of the languages [17]. The use of a formal language like *Maude* allowing the execution of formal specification avoids that problem.

6 Future work

The method presented is part of an environment that we are developing to specify cooperation systems with important coordination constraints, from early stages in the software development process. Another part of this environment is a technique to describe, in a graphical way, the elements composing the system and their dependency relations, which we have called IRDs (Interelement Requirements Diagrams) [18]. These diagrams have a representation in Maude. To guarantee that the detailed system specification is coherent with the requirements expressed in the *IRD* of the system, an accordance checker tool is developed in Maude, making use again of its reflection capability. To facilitate the interaction between all parts in the environment being developed and the developer's task , our next objective is to generate automatically *CoordMaude* specifications of the system from *IRDs*. In this way, the changes in the composition and /or coordination policies of the system will be able to be expressed dynamically in the different abstraction levels of the system specification. The whole environment is completed with the generation of Java code.

7 Conclusions

This work explains a method to formally represent applications in cooperation domains. *Coordinated Roles* is the coordination model adopted, because it is an exogenous, transparent model, promoting the separation of functional and coordination aspects. This allows the functional components and the coordinated pattern to be reused, without a loss in expressiveness.

Maude is the formal language used to represent the coordination model that may allow the execution of specifications and provide in this way a technique to validate the system specifications.

The system behaviour can be simulated then, defining different objects configuring the system and different sequences of operations. This is especially important in systems when the configuration of objects is variable along time, and the dynamic change of coordination policies is required.

However, the handling of formal specifications is not a single task and these specifications become long and complex. To avoid this disadvantage and still maintain the specification execution advantage of using *Maude* language, *CoordMaude* has been defined. This set of primitives, developed in the same language that profits from the reflection capability of *Maude*, allows the use of the simpler notation of *CR*, generating the whole *Maude* specification automatically. This resulted in better understanding of the specifications and greater easiness to make changes, together with the capability to execute the specifications.

References

[1] Frolund. Coordinating Distributed Objects. An Actor-Based Approach to Synchronization. The MIT Press. 1996.

[2] F. Arbab. What Do You Mean Coordination? Bulletin of the Dutch Association for Theoretical Computer Science (NVTI). March'98.

[3] J.M. Murillo, J. Hernández, F. Sánchez, L.A. Álvarez. Coordinated Roles: Promoting Re-usability of Coordinated Active Objects using Event Notification Protocols. 3rd Int. Conf. Coordination'99. LNCS 1594. Springer-Verlag. 1999.

[4] J.C. Cruz, S. Ducasse. A Group Based Approach for Coordinating Active Objects. 3rd Int. Conf. Coordination'99. LNCS 1594. Springer-Verlag. 1999.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. Quesada. Maude: Specification and Programming in Rewriting Logic. Computer Science Laboratory. SRI International. March 1999.

[6] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. 1st Int. Conf. Coordination'96. LNCS 1061. Springer-Verlag. 1996.

[7] A. Gravell and P. Henderson. Executing Formal Specifications Need Not Be Harmful. Software Engineering Journal vol. 11 n^o 2, 1996.

[8] P. Inverardi, A. Wolf and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. 2nd Int. Conf. Coordination'97. LNCS 1282. Springer-Verlag. 1997.

[9] M. Sánchez-Alonso, J. L. Herrero, J. M. Murillo, J. Hernández. Guaranteeing Coherent Software System when Composing Coordinating Systems. 4th Int. Conf. COORDINATION'2000. LNCS 1906. 2000.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel Computation in Maude. In Proc. 2nd Int. Workshop on Rewriting Logic and its Applications, ENCS Elsevier Sciences, 1998.

[11] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi and J. P. Jouannaud. Introducing OBJ. In Software Engineering with algebraic specification in action, edited with Grant Malcolm, Kluwer, 2000.

[12] R. Diaconescu and K. Futatsugi CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification, by, Volume 6 of AMAST series in Computing, World Scientific, 1998.

[13] P. Poizat, C. Choppy and J. C. Royer. From Informal Requirements to COOP: a Concurrent Automata Aproach. FM'99 World Congress on Formal Methods in the Development of Computing Systems. LNCS 1709. Springer-Verlag. 1999.

[14] A.Grau. Computer-Aided Validation of Formal Conceptual Models PhD. Thesis. Technischen Universität Braunschweig. March 2001.

[15] J.Kusch, P.Hartel, T.Hartmann and G.Saake. Gaining a Uniform View of Different Integration Aspects in a Prototyping Environment. 6th Int. Conf. on Database and Expert Systems Applications. LNCS 978 Springer-Verlag 1995.

[16] P. Letelier. Animación Automática de Especificaciones OASIS utilizando Programación Lógica Concurrente. PhD. Thesis, UPV Valencia, 1999.

[17] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executableΓ. In Software Engineering Journal Vol. 4 n^{o}6 pags: 320-338, 1989.

[18] M. Sánchez-Alonso and J. M. Murillo. Specifying Cooperation Environment Requirements using Formal and Graphical Techniques. 5th. Workshop on Requirements Engineering WER'2002, Valencia (Spain), November, 2002.