COFRE: Environment for Specifying Coordination Requirements using Formal and Graphical Techniques[1]

Marisol Sánchez-Alonso, Juan M. Murillo and Juan Hernández
QUERCUS Software Engineering Group, Computer Science Department,
University of Extremadura, Spain
Escuela Politécnica
Avenida de la Universidad S/N
10.071 Cáceres, España
Email: {marisol, juanmamu}@unex.es

Advances in computer science have enabled the development of more and more complex systems. One of the most powerful tools to manage these systems is coordination models and languages. However, a serious limitation of these models, with regard to their usability, is that they do not provide support to manage the coordination constraints from the early stages in the software life cycle. Indeed, while these models provide suitable support to structure the applications giving a separate treatment to coordination patterns and functional components at the implementation phase, they provide little support to deal with adequate treatment of such concerns in the requirements definition. Coordination models should be encompassing a methodology supporting the separation of concerns throughout the whole software development process. An example of such methodology is presented in this paper providing a graphic technique, a method of generating formal interpretable specifications for the reproduction of coordinated environments and an accordance checker to verify the system formal representation with regard to the graphic one. The method is based on the use of the formal language Maude *(as a simulation tool) and* Coordinated Roles *as a coordination model.*

ACM Classification: D.2.1 (Software – Software Engineering – Requirements/Specifications), I.6.5 (Computing Methodologies – Simulation and Modelling – Model Development), I.6.7 (Computing Methodologies – Simulation and Modelling – Simulation Support Systems)

1. INTRODUCTION

Advances in computer science have enabled the development of more and more complex systems, and one of the latest trends is component based software development (Heineman and Councill, 2001), where interactions and relations between components are established to configure the whole system. In this way, coordination models and languages (Frølund, 1996; Inverardi, Wolf and Yankelevich, 1997; Arbad, 1998; and Cruz and Ducasse, 1999) are powerful tools for the management of the interaction concerns between components.

However, the application of these models has focused on the last stages of the software life cycle. This is a serious limitation with regard to their usability, because they do not provide support

[1] This work has been supported by the project CICYT under grant TIC 02-04309-C02-01.

Manuscript received: 15 December 2003
Communicating Editor: Didar Zowghi

to manage the coordination constraints from the early stages in the software development process. Indeed, these models provide suitable support to structure the applications giving separate treatment to coordination patterns and functional components at the implementation phase, but they provide little support to deal with adequate separation in the requirements definition or architectural design phases. Thus, the treatment of coordination constraints is relegated to the implementation phase, incrementing the responsibilities of programmers. As a result, they have to accommodate the system specifications to the rules and tools provided by the coordination model, and this accommodation supposes changes that create a mismatch with the system implementation, having critical consequences over the system maintenance and being a possible cause of misinterpretations and incoherencies.

To deal with the specification of complex systems, coordination models should be encompassing a methodology supporting the separation of concerns throughout the software life cycle. This methodology can provide formal techniques to describe system specifications, thereby avoiding ambiguities, increasing strictness and making use of verification tools to demonstrate properties that the system must satisfy.

Using formal techniques to represent the system requirements in such methodologies allows practitioners to check the syntactic and semantic correctness of the system specifications and avoids ambiguities and lack of precision. However, one can have a correct specification that does not match the user requirements. Therefore, a specification validation step is required to confront requirements stated by users with specifications representing the system's conceptual model.

Some validation methods (Gravell and Henderson, 1996) include the simulation of the system by executing the formal specifications. Using that technique, users can observe the system's dynamic behaviour in different situations, checking whether the specifications produce the expected results. Moreover, the execution of specifications makes the reproduction of complex configurations easy, helping to discover deficiencies and situations not considered in the initial requirements.

With respect to the representation of system requirements, although formal techniques are more rigorous, graphic techniques are more easily understood by stakeholders. Therefore, it is advantageous to combine both. In this way, the former help to understand the interactions between system components, and the latter allow the concrete features that complete the graphic to be detailed.

On the other hand, the system representation by means of formal techniques in different detail levels allows us to verify whether these representations are coherent and consistent with the initial requirements definition.

COFRE (*Coordination Formal Requirements Environment*), is a set of tools offering all the above features, and provides a methodology to make the system specification easy based on formal and graphical techniques, while focusing this task on systems with important coordination constraints. This relegates the choice of a coordination model to the design phase while dealing with coordination constraints from the early stages. As a benefit, changing and reusing components and coordination patterns from requirements is a more systematic and pleasant task. Moreover, the simulation of the system is possible by executing the formal specifications, which simplifies the validation process, and the use of a verification tool allows for checking the accordance between specifications in different abstraction levels.

2. MOTIVATIONS

With the aim of making software development easier, a wide range of graphical tools combining both graphical and formal techniques have been developed in recent years. For example Rhapsody (Rhapsody, 1999) and Statemate in Harel and Naamad (1996) are commercial tools based on the use of statecharts defined in Harel (1987) and are appropriated to specify intra object behaviour. The OO-Method (Pastor *et al*, 1998) combines OASIS (Pastor *et al*, 1997) and UML (Rumbaugh, Jacobson and Booch, 1999), and the TROLL Workbench in Grau (2001) and TROLL Tbench (Kusch *et al*, 1995) tools combine TROLL (Jungclaus *et al*, 1996) and OTROLL (based on OMT). In Drew (1995) the combination of Causal Diagrams, a mathematic model to avoid the inherent ambiguity of the diagrams and a textual description to help in the understanding of the model for stakeholders is proposed. Causal Diagrams are oriented to describe how some attributes and calculus (that determine the system state) can be obtained starting from others. These tools can express in a detailed way the static and dynamic aspects of the system, making use of different charts to express each one and allowing for the description of the system representation until the detailed design or implementation stages.

However, initial requirements making use of these techniques are expressed in a global way that makes it difficult to adopt an architectural or design technique based on the separation of concerns (including coordination constraints). Particularly in coordination environments, the adoption of a coordination language requires the separation of the specification of the coordination behaviour from the functional one. But this task is delegated to designers or programmers starting from a conceptual model where these concerns are mixed. To avoid this problem, the development processes are made more agile and consistent. In this way, COFRE separates the functional concerns described for each component from concerns related to the interactions between them from requirements definition in developing process, adopting specific tools which allow for the representation of this separation from early stages.

In order to provide the validation process, several techniques have been proposed rendering the conceptual model more comprehensible for users. Most of these techniques consist of introducing graphic symbols or user's concept defined as Kung (1993), paraphrasing parts of the conceptual model in natural language in Dalianis (1992) or generating explanations from the specification (Gulla, 1996). At any rate, simulation by means of the model execution permits the observation and testing of the dynamic properties of the system. Often the formal specifications execution is named *animation*. Most of the animation techniques need the translation of the specification to a programming language in order to be executed (Grau, 2001; Kusch *et al*, 1995; Letelier, 1998), which can provoke a lack of precision and fidelity between both representations due to the different abstraction levels of the languages.

In coordination systems attention must be focused not only on validating each system component, but also on guaranteeing that the final behaviour obtained in the composed application is semantically coherent. Of particular importance is whether gluing together a coordination policy and a set of components (that can have been coded separately) in an application will produce the expected behaviour, and whether the addition or change of coordination constraints will not produce conflicts with the current behaviour.

In addition, a verification process is necessary for checking whether or not the interaction constraints agree between elements expressed in the initial requirements definition, and the detailed system specification where a specific coordination model has been applied.

In order to explain our proposal to improve the software developing process under these motivations, a single but illustrative example is presented. The example is an extension inspired in the Car Park study case presented in Sánchez *et al* (1999).

2.1 Car Park example

This example presents a system to control the access to a car park, which has the following elements: a traffic-light at the entry of the car park, a ticket-machine to provide the entry tickets, an optic-reader to control the entry of regular customers, and a barrier allowing the access to the car park. 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 *Collect* action. It is necessary to impose this constraint, because if the ticket machine cannot execute the *Give* action (i.e. there is no paper), or a ticket is produced but not collected, a car could enter. A separate provision is made for regular customers; they exhibit a sticker in their windscreen that can be read by the optic reader. In this way, when a car comes into the car park, the optic reader first detects if the car has the sticker of a regular customer; otherwise, the ticket machine produces a ticket. In both cases, the barrier is raised only if the sticker has been read or the produced ticket has been collected. In the entry, the traffic-light remains red while the car park is full; otherwise it is in green. When the traffic-light is in red, it will not allow new cars to go in. In order to control that, the corresponding actions in the optic-reader and the ticket-machine, reading the customer stickers or supplying entry tickets, are inhibited.

3. A PROPOSAL FOR DEVELOPING COORDINATION SYSTEMS

In this section, the above topics are focused on, proposing COFRE to make the specification and validation process easier for both software engineers and users. This proposal is based on the joint use of the formal language *Maude* (Clavel *et al*, 1999) and a new kind of diagrams called Interelement Requirement Diagrams (IRDs).

Maude is an executable algebraic language based on rewriting logic that 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 specifications behaviour (Sánchez *et al*, 2000). The language permits the definition of modules containing operations, equations and rewrite rules. The execution of specifications is performed by means of reducing terms in equations and rewrite rules.

The use of rewrite rules is particularly appropriate to represent state transitions in systems of concurrent objects, where a configuration formed by the object instances and the current messages present in the system determine the system state in each moment. *Maude* is a reflective language like the rewriting logic on which it is based, and provides specific definitions and operations to efficiently manage specification modules and terms specified in the language itself. This capability allows *Maude* to be used to develop tools that manage specifications written in *Maude* as well. The executability of the *Maude* specifications and the reflection make the use of *Maude* appropriate to represent the different abstraction level specifications of the system as well as to manipulate the specifications themselves, being the language in which part of the tools composing COFRE are implemented.

While *Maude* is used to specify component behaviour, IRDs are used to specify the cooperation rules (coordinated interactions) between components in a graphical way. The use of IRDs makes the system specifications more comprehensible but does not introduce lack of strictness. In fact, the artifacts from IRDs have a well-defined semantic in *Maude*.

The transition to the design stage is made by adopting the exogenous coordination model Coordinated Roles (CR) (Murillo *et al*, 1999). This coordination model is inspired by *IWIM* model defined in Arbad (1996) and 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 an *asynchronous* way. The events for which notification can be asked 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 component has reached a particular abstract state (*SR event*). Each coordination component imposes a coordination pattern structured as a set of roles. A role represents an abstract system component. The actions or the state defined in a role can constrain the execution of the actions defined in other roles. Consequently, each of the characters that can be played in a coordination pattern is represented by a role. 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 represent the desired coordination constraints. The binding between coordinators and components to be coordinated is done at run-time via composition syntax.

So, the artifacts from IRD diagrams have to be represented in terms of CR definitions. These definitions also have a representation in Maude language (Sánchez-Alonso and Murillo, 2002), in a sufficiently detailed specification that allows the execution representing different system configurations. In this way, the system behaviour can be simulated and observed, contributing to the system validation. Moreover, the specification adopting the coordination model can be verified with respect to the initial formal representation of the IRD to determine their accordance making use of the accordance checker, developed for this purpose. Figure 1 shows a schematic representation of the method.

3.1 Interelement Requirements Diagram

In the system requirements description, it is interesting to represent the system elements and their dependencies using an initial graphic schema, independent of later design decisions. Users must help to make that representation, collaborating in the discovery and the comprehension of the relationships between the different components in the system. With the aim of achieving an easy graphic representation to express an initial system requirement, thereby facilitating the posterior application of a coordination model, we propose a new kind of diagram named Interelement Requirement Diagrams (*IRD*). IRD definition is the first step in COFRE method (see Figure 1). This

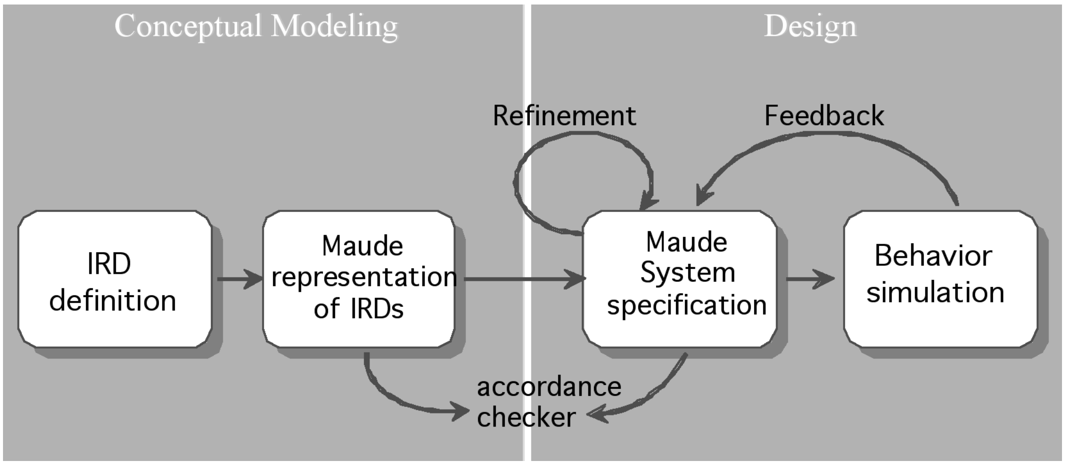
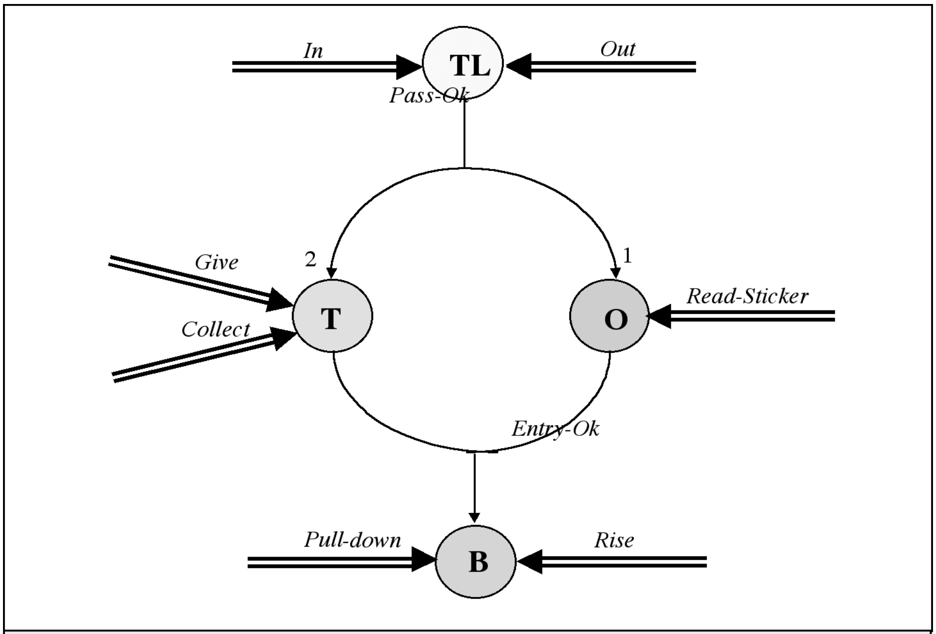


Figure 1: Schematic representation of COFRE method steps

task consists of representing the physical elements of the system and their dependencies, identified starting from user explanations.

The aim of *IRDs* is to represent the system's main features in the requirements definition when the system's specific objects and their classes have not yet been determined. This representation consists of a unique graph where the following topics are expressed: main system elements, how they are related, their global behaviour, how they answer to a specific stimulus, and how different features in their context are expressed. The system's static (the element's observable structure and external interface) and dynamic (relations describing the system behaviour) aspects are contained in the same



Element TL **is** :: traffic-light
State Pass **: Bool (init** false **)**
Values Capacity: Integer **(init** 150 **)**, Green: Bool **(init** true **)**
Actions In **{** Capacity**--** **;** Pass = **false** ; **If** capacity **>=** 0 **then** Pass **= true** **;**
If Capacity <= 0 **then** Green **= false }**
Out **{** Capacity**++;** Pass **= true ; If** capacity **>=** 1 **then** Green**= true }**

Element T **is** :: ticket-machine
Actions Give
Collect

Element B **is** :: barrier
Actions Rise
Pull-down

Element O **is** :: optic-reader
Action Read-Sticker

Relation Pass-Ok
Constraint async State Pass **== true then** (Read-Sticker **else** Give))

Relation Entry-Ok
Constraint (sync End Collect **or sync End** Read-Sticker**) then** Rise

Figure 2: IRD of the car park example.

representation. Static aspects are expressed by means of nodes named elements, their observable external actions (operations that the element can perform) and the values needed to perform these actions. Dynamic aspects are expressed by means of relations between components, represented by single arrows, where some conditions can decide whether the action invocations are attended to or not.

Figure 2 shows the IRD corresponding to the car park example, described in the previous section. System components are represented as nodes in the diagram named elements. For each element, values, state, actions and activities can be defined (grey side in Figure 2). The state and actions can be asked for from relations between elements while activities and values are private for elements. The values represent features of the element and actions and activities represent tasks that an element can perform. The difference is that actions are invoked by means of a message passing and activities are invoked internally. The definition of a kind of elements is specified the first time that an element of this class is represented; the rest of the time it is only necessary to specify the name of the kind of element.

The interactions between elements and their constraints are expressed in terms of relations. A relation can ask for events over actions or the state of the source objects. These events allow the actions restricted in target elements to be executed if they have been previously invoked.

In Figure 2, the state of the traffic-light imposes a constraint over the ticket-machine and the optic-reader in order to prevent cars from passing if there are no vacancies. This constraint is represented by means of the *Pass-Ok* relation with the traffic-light as source and the ticket-machine and the optic reader as targets. When the *Pass* state is true the *Read-sticker* action or the *Give* action can be performed. As the *Give* action only has to be produced if the car passing is not a customer, the priority of the actions is expressed by a number in the relation next to the target element in the diagram, and by means of an *else* operator in the definition. The other relation, *Entry-Ok*, determines that the invocation to the Rise action in the barrier element has not been attended to until the *End* of the *Read-sticker* action or the *Collect* action from the source elements has been produced.

3.1.1 Formal representation of IRDs

An *IRD* represents system requirements in a semiformal way providing a visual representation of the system and a description of the constraints over the relations between the system elements. That representation is easier to understand by users and designers but it may be inaccurate, incomplete or inconsistent with the design specification In order to take advantage of this representation and avoid the disadvantages, the semantics of each element in the IRD has been defined using the *Maude* formal language (second step in Figure 1).

Each element in an *IRD* is represented by means of an object module in *Maude*, where a class is defined as a subclass of *Elmt* (defined in DefELEMENT) with its own attributes and the actions defined as messages.

Returning to the Car Park example, the module ELMT_ticket-machine represents the ticket-machine class of element.

```
(omod ELMT_ticket-machine is
   protecting DefELEMENT .
   class ticket-machine .
   subclass ticket-machine < Elmt .
   msg Give : Oid -> Msg .
   msg Collect : Oid -> Msg .
endom)
```

The ticket-machine is an *Elmt* subclass. The two actions *Give* and *Collect* are represented as messages. No attributes and no State are defined in this element. In the same way, the *ELMT_barrier* and *ELMT_optic-reader* are defined with the messages representing their corresponding actions.

The *traffic-light* element definition in the *IRD* has two actions, *In* and *Out*, that are formally represented as messages, but in this case the actions define sentences. When sentences are defined in an action, they are represented in *Maude* as rewriting rules.

```
(omod ELMT_traffic-light is
   protecting DefELEMENT .
   class traffic-light | State : Ste('Pass , nil ),
                         ValState : [ true ; false ] , Capacity : Int ,
                         Green : Bool .
   subclass traffic-light < Elmt .
   msgs In Out : Oid -> Msg .
   var TL : Oid .
   var C : Int .
   rl[In1] : In(TL) < TL : traffic-light | Capacity : C >
      => < TL : traffic-light | State : Ste ('Pass, false) ,
                                Capacity : (C - 1) > .
   crl[In2] : < TL : traffic-light | State : Ste ('Pass, false),
                                     Capacity : C  >
      => < TL : traffic-light | State : Ste ('Pass, true) >  if C >= 0 .
   crl[In3] : < TL : traffic-light | Capacity : C , Green : true >
      => < TL : Traffic-light |Green : false >  if C <= 0 .
   rl[Out1] : Out(TL) < TL : traffic-light | Capacity : C >
    => < TL : traffic-light | State : Ste ('Pass, true ) ,
                              Capacity : ( C + 1 ) > .
   crl[Out2] : < TL : traffic-light | Capacity : C, Green : false >
       => < TL : traffic-light | Green : true >  if C >= 1 .
endom)
```

Another object module represents the complete *IRD* in *Maude*, specifying the instance of the *IRD* class where the set of elements and the set of relations are defined. In the example, the *IRD_Car-Park* module represents the concrete instances of the elements defined above and their relations.

```
(omod IRD_Car-Park is
     protecting DefIRD .
     protecting DefRELATION .
```

```
          protecting ELMT_ticket-machine .
          protecting ELMT-barrier .
          protecting ELMT-traffic-light .
          protecting ELMT_optic-reader .
          op init : -> Configuration .
          subsort Qid < Oid .
          eq init =
            < 'T : ticket-machine | State : null , ValState : nil >
            < 'B : barrier | State : null , ValState : nil >
            < 'TL : traffic-light | State : Ste('Pass, false),
                                    ValState : false true ,
                                    Capacity : 150 , Green : true >
          < 'O : optic-reader | State : null , ValState : nil >
          < 'Pass-Ok : Relation | From : 'TL , To : 'T 'O ,
                  Constraint : Event sync 'TL . 'Pass == true
                      < Action(Read-Sticker('O)) else Action(Give('T)) >
          < 'Entry-Ok : Relation | From : 'T 'O , To : 'B ,
                  Constraint : < Event(sync, EoP, Collect('T)) or
                                 Event(sync, EoP , Read-Sticker('O)) >
                        Action (Rise('B)) >
          < 'Car-Park : IRD | ElmtSet : 'T 'B 'TL 'O ,
                              RelSet : 'Pass-Ok 'Entry-Ok > .
   endom )
```

It is necessary to import all the above element definitions and a set of auxiliary modules. The *init* operation results in a system configuration whose equation creates instances of each element in the car park *IRD*. The two constraint relations in *IRD* are declared now. The '*Pass-Ok* relation indicates the origin of the relation in the *From* attribute. In this case the origin is the *'TL* traffic light element. The final elements in the relation *'TM* and *'O* are indicated by the *To* attribute. The *Constraint* attribute expresses the necessary conditions to perform, first, the *Read-Sticker* action in the *'O* optic reader, if an invocation to this message exists (the customer sticker has been detected on the windscreen); otherwise it performs the *Give* action. One condition must be satisfied to execute those actions; the *'Pass* state of the *'TL* traffic light element must be true. The processing mode has to be synchronous: the traffic light will be locked until the *Read-Sticker* or the *Give* action can be processed.

The *'Entry-Ok* relation acts in the same way; it has two original elements and a final element represented in the attributes *From* and *To* respectively. Its constraint has two events separated by an or operator, describing that one of these events must occur to allow the execution of the *Rise* action in the barrier element in a synchronous mode. These events are the end of the processing of the *Read-Sticker* action in the optic reader element or the end of the processing of the *Collect* action in the ticket-machine element. Finally, *'Car-Park* object, which represents the *IRD*, is defined, including the set of elements and the set of relations.

In this way, the interaction constraints imposed by the relations between elements are specified separately of the elements, providing several advantages:

1. This representation allows one to gradually broach the system specification difficulty from requirements, focusing on the interaction rules of the system abstracting of the component specifications.
2. The adoption of a coordination model in the design stage is direct and more intuitive.
3. It makes the reusability of the *IRDs* easier, changing the constraints imposed by the dependency relations without modifying the element specifications.
4. The formal representation and posterior steps (shown in Figure 1) are generated automatically from *IRDs*, making easier the development process and reducing the corresponding efforts and costs.

3.2 Maude System Representation

Thanks to the separation of the coordination rules from the system elements, the correspondence between the formal specification of IRDs and a coordination model is simple. In COFRE, the coordination model applied in this step is *Coordinated Roles*, but other models could be adopted following a similar correspondence. Each element definition is corresponded with a CR class definition and each relation is corresponded with a coordinator class, where actions or states which constrain and those which are constrained adopt the corresponding roles in the coordinator. The roles are specified considering the constraints imposed by the events to be notified. All actions constrained are specified depending on a synchronous BoP notification that can be produced when the events constraining this action have occurred.

In order to validate the coordinated behaviour, we propose the execution of the specifications making use of the Maude language. This makes it necessary to establish the parallelism between the coordination mechanisms of *CR* and the mechanisms of *Maude* used to specify them (see Figure 3).

The classes of objects (corresponding with elements in IRDs) to be coordinated are represented in *Maude* as object modules. Each class defines the messages and rules describing the activities and actions of the elements represented and the state and the values represented as attributes of the class. Moreover, three other attributes are defined in all the classes to perform the coordination tasks:

- *UnLock:* To indicate the method that the object is allowed to execute.
- *Faction:* To inform about the method that has just finished. Initially, or when the information of the last method has been processed, its value is none.
- *actList:* Represents the object log, recording the sequence of methods executed and the state changes that occurred in the object.

Objects are not allowed to modify the value of *UnLock*, and cannot set *Faction* to the value *none*. These actions are controlled by special objects named *notificators*. There is a notificator for each method that can be invoked in an object, and for each state to be notified. Notificators intercept the messages invoking methods of an object. Each notificator associated to a method has a list of the different kinds of notifications to be performed. When a message is intercepted by the corresponding notificator, this performs all the notifications associated to the method invoked and determines whether the method can be allowed. If the method is allowed, the notificator sets the method to the *Unlock* attribute of the corresponding object. Depending on the kind of notification to make, the notifications can be performed before or after the execution of the method or just when the message is intercepted. Special notifications are produced when objects change their state. In this last case no methods are allowed, only the notification of state reached are performed.

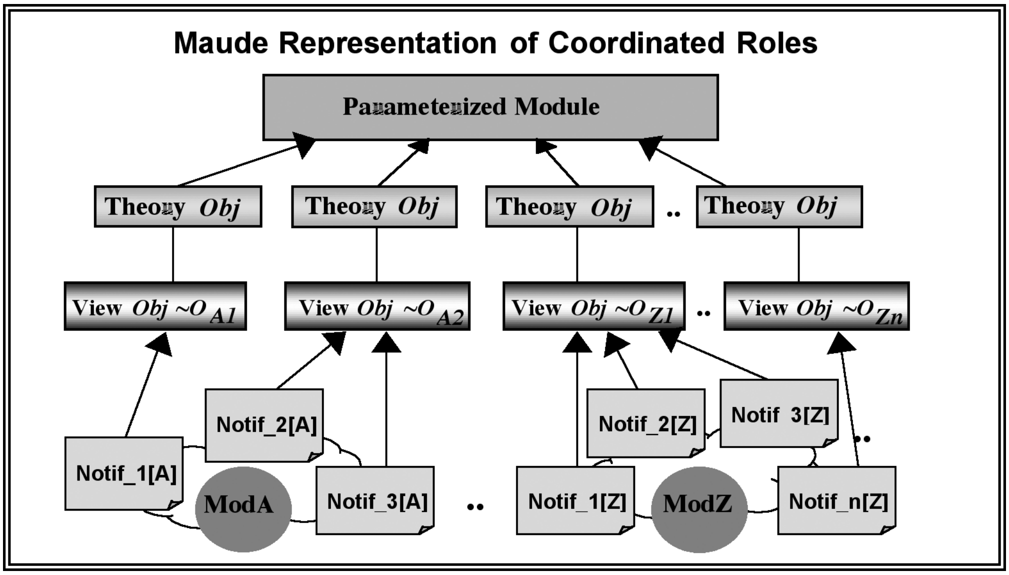


Figure 3: Schematic representation of CR in Maude.

The coordinators are represented by means of parameterized modules, where the parameters are defined using the same theory OBJT (see Figure 3). This theory represents the common attributes in all objects that are necessary to perform the coordination. The parameters represent the different roles defined in the coordinator. To make the binding between objects and the coordinator parameters, views are defined.

The definitions of the kinds of notifications used in a coordination module have to be imported making reference to the parameter (role) which is linked. The coordinator specifies the coordination constraints by means of rewriting rules, processing the event notifications received.

3.3 Behaviour Simulation

In order to generate an object configuration representing the system and a specific events occurrence, an object module is specified. This test module imports all the modules defining coordinators and classes of elements. In the coordinators, formal parameters are substituted for the views establishing the binding between a role and a specific object class. In addition, the instances of the coordinators, classes and a test sequence of messages are defined. The example module is executed and the configuration resulting can be checked with the expected configuration to determine if the result agrees with the expected coordinated behaviour. By changing the configuration, different situations and message sequences can be checked.

However, the Maude representation of *CR* is very complex and difficult for designers to understand because it introduces mechanisms to represent the coordination model that make the specification obscure. On the other hand *CR* syntax is very comprehensible, and for this reason the method provides a set of Maude specifications named *CoordMaude* defined in Sánchez-Alonso *et al* (2003), allowing the generation of the Maude specification of *CR* starting with the specification in CR. In this way, *CoordMaude* avoids the complexity of the specifications and allows the execution to simulate the system behaviour, contributing to the validation process.

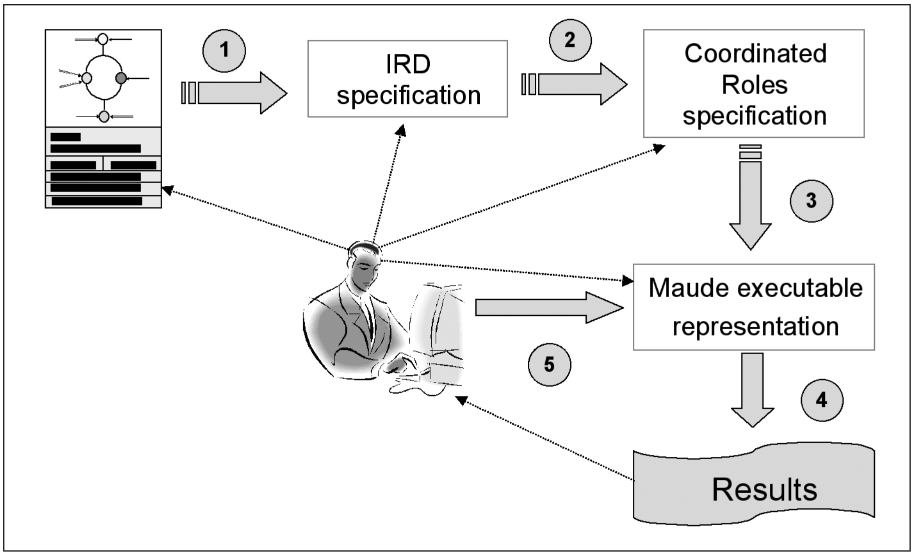

1
IRD specification
2
Coordinated Roles specification
3
Maude executable representation
4
5
Results

Figure 4: Documentation generated from IRDs

3.4 Feedback and Refinements

Figure 4 shows the different representations generated by the environment. The translation in each step is made automatically, but practitioners can interact directly in each of the representations and continue with the next steps starting from that point. Step 1 in Figure 4, generates the formal representation of an IRD system diagram, and step 2 generates the corresponding representation in CR syntax. The successive refinements of the detailed system specifications providing details about the internal functionality of the system components can be defined starting from this representation. CR representation is translated (step 3) and executed by *CoordMaude* (step 4) to facilitate the validation task, and different system configurations and events can be tested to simulate system coordinated behaviour (step 5). The results are fed back to practitioners to determine whether the system behaviour is what was expected or not. In this last case, they could apply the necessary changes to solve the problems detected, starting from any of the system representations.

3.5 Accordance Checker

With the aim of checking the accordance between the different abstraction levels resulting in the development stages, a verification process is required. It is particularly necessary to check whether all the initial requirements are maintained and also whether detailed specifications introduce constraints or requirements that are not present in the initial requirements model. Although this should not occur in the automatic process along all the steps in COFRE, the refinement process (allowing the manual modification of specifications) could introduce changes affecting the system configuration and the components interactions.

In this work, the specific goal is to guarantee that the elements and coordination constraints expressed in the IRD diagrams are in accordance with the objects and coordinators definitions (in the *Maude* representation of the system), adopting the CR model. This means that all the interactions and their features present in an IRD diagram must be present in the specification of the system adopting CR. However, the lower level specification can contain relative features detailing internal behaviour in the objects corresponding to the elements in the IRD diagrams that are not present in IRDs representation. This is due to the different abstraction level between both representations and occurs throughout all the refinements in the system, and this case must not be considered a lack of accordance.

The accordance checker tool included in COFRE analyzes both representations detecting a lack of accordance. This tool is developed in *Maude* making use of the reflective capability, and to facilitate the checking process, the specifications syntax needs to be very restricted. The checker starts from the two Maude system specifications: the IRD formal representation and the CR representation. First, the checks are made with regard to the elements present in the system IRD. All of them must appear as objects in the system specification. The opposite does not have to be true, because other objects can appear in the specification to consider design or implementation details. The condition is that these additional objects do not participate in the system coordination constraints. For each element, all its features are compared with the corresponding object features of the detailed specification. Second, the tool examines the relations between elements in the IRD and compares them with the coordination patterns described in the detailed specification. In this case each relation is matched to a coordination pattern and no constrains can be imposed in either representation that does not appear in the other. If there is a lack of accordance between the system IRD and the detailed specification, the tool returns an error message announcing the origin of the error. It can have warning messages if there are few differences between both representations. The verification tool also informs about the detailed objects and features present in the specification representation and they are not present in the corresponding IRD.

The tool can be used to check the specification resulting after each refinement iteration with the requirements specification reflected in the system IRD.

4. FUTURE WORK

At present, COFRE is being applied to a project for the development of domotic systems. CR syntax is well known by practitioners who have noticed the facility to obtain the CR specification from IRD graphic representation. The simulation of different configurations to validate the system coordinated behaviour is a useful tool in their opinion but they have had to be trained in *Maude* to interpret the results of the simulation process. They have noticed the need to improve the interface of this tool to make transparent the mechanisms to represent CR in *Maude* when the simulation is performed. With regard to the accordance verification tool, practitioners have expressed their satisfaction in detecting unnoticed mistakes. The modifications and additions to the specification in the refinement iterations have been made from CR representation.

As practitioners have suggested, the simulation step must be improved, developing a tool that facilitates the simulation process and the resulting interpretation.

The generation of *Java* code starting from detailed specifications is another future work to consider.

One of the aims of this work is to consider the correspondence between the formal representation of the system and UML. The Classes Diagram generation is our starting point.

5. CONCLUSIONS

The system requirements representation must be done making use of a model that facilitates the comprehension and the development of the posterior phases. Particularly in coordination systems, the adoption of a coordination model must be facilitated. Moreover, the system behaviour needs to be validated from the early stages, and the accordance between different system representations verified.

For this purpose, COFRE has been developed, containing a set of techniques and tools to simplify the specification and development of systems with important coordination constraints. Thus, the main advantages of the set of techniques composing COFRE are:

1. By using IRDs, the coordinated interactions between components can be specified independently from component functionality. Thus, IRDs make the formal specification of complex systems easier by focusing on how components interact and abstracting from specifications of components' internal behaviour.
2. IRDs are more suitable graphical and visual specification tools than formal specification languages. By using IRDs, stakeholders can easily understand the specification of the rules that govern the interaction between components.
3. The formal representation of IRDs in Maude avoids the ambiguity that can be produced from the graphical representation, adding precision to the model. On the other hand the graphic representation makes the understanding of the formal specification of the system easy.
4. The adoption of a specific coordination model is facilitated from early stages in the development process due to the separation of functional and coordination concerns.
5. The features and event notification protocols are represented in Maude to make use of the rewrite engine of the language and with the aim to simulate the system behaviour with different configurations. Thus, stakeholders are allowed to check whether the specified system produces the expected results, contributing to the validation process.
6. The execution of the system representation adopting *CR* syntax is possible by using *CoordMaude*. This tool extends *Maude* accepting specifications made on *CR* and generating all the mechanisms needed to represent and execute the coordination model.
7. Formal representation of IRDs can be checked with the specifications resulting from the application of the coordination model to determine whether the accordance between both representations is maintained. The accordance checker detects errors in the representation of interactions between components or constraints that do not exist in the initial requirements descriptions.

REFERENCES

ARBAB, F. (1996): The IWIM model for coordination of concurrent activities. CIANCARINI, P., HANKIN C. (Eds.). *First International conference Coordination'96*. LNCS 1061, 34–56. Springer-Verlag.

ARBAB, F. (1998): What do you mean coordination? *Bulletin of the Dutch Association for Theoretical Computer Science* (NVTI). March.

CRUZ, J. C. and DUCASSE, S. (1999): A group based approach for coordinating active objects. CIANCARINI P. and WOLF A. L. (Eds.) *Third International conference Coordination'99*. LNCS 1594: 355–371. Springer-Verlag.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET N., MESEGUER, J. and QUESADA, J. (1999): Maude: Specification and programming in rewriting logic. Computer Science Laboratory. SRI International. March.

DALIANIS, H. (1992): A method for validating a conceptual model by natural language discourse generation. In LOUCOPOULOS P. (ed.) Proc. of *Int. Conf. On Advanced System s Engineering (CAISE'92)* Springer, LNCS 593: 425–444.

DREW, D. R. (1995): System dynamics: Modelling and applications. *ENGR 5104 Study Notes*, VPI&SU, Blacksburg, VA.

FRØLUND. (1996): Coordinating distributed objects. An actor-based approach to synchronization. The MIT Press.

GRAVELL, A. and HENDERSON, P. (1996): Executing formal specifications need not be harmful. *Software Engineering Journal,* 11(2): 104–110.

GRAU, A. (2001): Computer-aided validation of formal conceptual models Ph.D. thesis. Technischen Universität Braunschweig.

GULLA, J. A. (1996): A genera explanation component for conceptual modelling in CASE environments. *ACM Transactions on Information Systems,* 14(2): 297–329.

HAREL, D. (1987): Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8: 231–274.

HAREL, D. and NAAMAD, A. (1996): The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4): 293–333.

INVERARDI, P., WOLF, A. and YANKELEVICH, D. (1997): Checking assumptions in component dynamics at the architectural level. GARLAN D., LE MÉTAYER, D. (Eds.). *Second International conference Coordination'97*. LNCS 1282. Springer-Verlag.

JUNGCLAUS, R., SAAKE, G., HARTMANN, T. and SERNADAS, C. (1996): TROLL. A language for object-oriented specification of information systems. *ACM Transactions on Information Systems,* April, 14(2): 175–211.

KUSCH, J., HARTEL, P., HARTMANN, T. AND SAAKE, G. (1995): Gaining a uniform view of different integration aspects in a prototyping environment. In *Proc. 6th Int. Conference on Database and Expert Systems Applications (Dexa'95)* Springer-Verlag LNCS 978: 37–47.

KUNG, D. (1993): The behaviour network model for conceptual information modelling. *Information Systems*, 18(1): 1–21.

LETELIER, P., SÁNCHEZ, P. and RAMOS, I. (1998): Prototyping a requirements specification through and automatically generated concurrent logic program. *Practical Aspects of Declarative Languages*, LNCS 1551: 31–45.

MURILLO, J.M., HERNÁNDEZ, J., SÁNCHEZ, F. and ÁLVAREZ, L.A. (1999): Coordinated roles: Promoting re-usability of coordinated active objects using event notification protocols. CIANCARINI P. and WOLF A. L. (Eds.) *Third International conference Coordination'99*. LNCS 1594: 53–68. Springer-Verlag.

PASTOR, O., INSFRÁN, E., PELECHANO, V,. ROMERO, J. and MESEGUER, J. (1997): OO-METHOD: An OO software production environment combining conventional and formal methods. In CAiSE'97, *Int. Conf. on Advenced Information Systems*, 145–158.

PASTOR, O., PELECHANO, V., INSFRÁN, E. and GÓMEZ, J. (1998): From object-ooriented conceptual modeling to automated programming in Java. In LING, T.W., RAM S. and LEE M. L. (eds.) *Proc. of the 17th Int. Conf. On Conceptual Modeling (ER'98)* LNCS 1507: 183–196.

RHAPSODY (1999): Rhapsody. I-Logix Inc., Andover, MA.

RUMBAUGH, J., JACOBSON, I. and BOOCH, G. (1999): The unified modeling language reference manual. Addison-Wesley.

SÁNCHEZ-ALONSO, M., HERRERO, J.L., MURILLO, J.M. and HERNÁNDEZ, J. (2000): Guaranteeing coherent software system when composing coordinating systems. PORTO A. and ROMAN C. (Eds.). *Fourth Int. Conference COORDINATION'2000*. LNCS 1906: 341–346.

SÁNCHEZ-ALONSO, M. and MURILLO, J.M. (2002): Specifying cooperation environment requirements using formal and graphical techniques. In Proc. *5th. Workshop on Requirements Engineering WER'200*.

SÁNCHEZ-ALONSO, M., CLEMENTE, P.J., MURILLO, J.M. and HERNÁNDEZ, J. (2003): CoordMaude: Simplifying formal coordination specifications of cooperation environments. *Second Workshop on Languages Description Tools and Applications (LDTA'03)*. ENTCS, 82.

BIOGRAPHICAL NOTES

Marisol Sãnchez-Alonso

Marisol Sãnchez-Alonso is an associate professor of languages and systems at the Extremadura University, Spain. She teaches software engineering in the Polytechnic Institute and Computer Science in the Faculty of Law at this University. As a researcher she is a member of the Quercus Software Engineering Group and her research interests are focused on the formal modelling of aspect oriented software development, particularly on coordination systems and on techniques for the behaviour simulation of objects cooperation environments. She has presented congress communications and published papers and books related to information systems development, information security and simulation of objects cooperation environments using formal methods.

Juan Manuel Murillo

Juan Manuel Murillo is associate professor at the computer science department of the University of Extremadura in Spain, at which he is a member of the Quercus Software Engineering Group. He received his MSc in computer science in the Polytechnic University of Catalonia, Spain, and his PhD in computer science at the University of Extremadura, Spain. His main research interests include software coordination and adaptation and aspect oriented software development. After the proposition coordinated roles, a coordination model separating the interaction concern, his interest now is in separating the coordination concern from early stages of the software life cycle.

Juan Hernãndez

Juan Hernãndez is associate professor of languages and systems and the Head of the Quercus Software Engineering Group of the Extremadura University, Spain, where he also currently leads the computer science department. He has served on several committees (steering, program and organizing) for national and international conferences on software engineering and object-orientation. He received a BSc in mathematics from the University of Extremadura, and a PhD in computer science from the Technical University of Madrid. His research interests are focused on aspect-oriented software development for open and distributed systems, including component-oriented programming and web services. He is involved in several research projects as responsible and senior researcher related to these subjects. He is a member of both the ACM and the IEEE Computer Society.