Specifying Cooperation Environment Requirements using Formal and Graphical Techniques[1]

Marisol Sánchez-Alonso and Juan M. Murillo

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

Abstract. . Using formal languages to specify system requirements guarantees the correctness of systems specifications. However, having correct specifications does not guarantee such specification matching user requirements. To guarantee such matching, users are required to validate formal specifications. This is a difficult task because, usually, users are unaware of notations. This work focus on this problem, in particular the validation of formal specifications of complex coordinated systems. To make the user's validation easier, a new graphic technique to represent the dependencies in a coordinated environment is proposed. This graphic (and visual) technique increases users' understanding whilst lack of precisions is avoided. In fact, the proposed graphics correspond with visual representations of formal Maude specifications. Besides, taking advantage of the features of Maude, the system simulation is supported by the execution of Maude specifications. Thus, users are allowed to check whether the system produces the expected results.

1. Introduction

One of the first steps to be taken when developing software systems is to represent the system's requirements. Using formal languages in these steps allows the syntactic and semantic correctness of the system specification to be checked, and at the same time, ambiguities and lack of precision are avoided [1]. However, formal languages themselves do not guarantee that produced specifications are valid [2]. Consequently, one can have a correct specification that does not match the user requirements. So, in addition to a system specification process, a specification validation step is required.

The validation process confronts informal requirements stated by users with specifications representing the system's conceptual model. Some validation methods [3] include the simulation of the system by executing the formal specifications. Using that technique, users can observe the system's dynamic behavior in different situations, checking whether the specifications produce the expected results. However, validation presents some difficulties: The validation process requires the users' collaboration, requiring them to understand formal specifications to detect errors and misinterpretations. Nevertheless, formal techniques are not especially

[1] This work has been supported by the project CICYT under grant TIC 99-1083-C02-02.

comprehensible by users unaware of notation [2]. This problem grows when the system being built is a complex one, in which different components interact following some negotiation rules. In such cases, the formal specification of components becomes obscured on the formal specification of negotiation rules, making harder the validation task.

This paper focuses the above topics proposing a technique 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 [4] and a new kind of diagrams called Interelement Requirement Diagrams (IRD). Maude is used to specify components behavior. IRDs are used to specify the negotiation 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 (and finally, IRDs are translated to Maude specifications). Thus, the main advantages of the technique presented are:

1. Using IRDs designers specify the coordinated interactions between components independently from components specifications. Thus, IRDs make the formal specification of complex systems by focusing on how components interact and abstracting from specifications of components' internal behavior.
2. IRDs are a graphical and visual specification tool more suitable for system users than formal specification languages. By using IRDs, users can easily understand the specification of the negotiation rules that govern the interaction between components.
3. The use of IRDs does not introduce lack of precision. IRDs are graphic representations with a Maude specification, and this graphical interface makes the Maude specification understandable.
4. The final specification obtained in Maude can be executed allowing the system simulation. Thus, users are allowed to check whether the specified system produces the expected result. Moreover, IRDs are integrated in a specification tool that supports the progressive refinement of the system specification.

The paper structure is as follows: In section 2 related works are commented. Section 3 describes the formal context used to specify de Diagrams explained in section 4. In Section 5 the formal specification of diagrams are described. Section 6 explains future works and works in progress. Finally, section 7 presents conclusions and next, the references.

2. Related works

In last years new specification languages (or extensions of existing languages) have been appeared, combining formal techniques with OO paradigm, like, *Lotos* [5], Z++ [6], VDM++[7], ALBERT[8], TROLL[9], OASIS[10]; getting the advantages of both, but the difficulty in understanding too.

With the aim of making the software development easier a wide range of graphical tools combining both graphical and formal techniques, have been developed in recent years. For example Rhapsody [11] and Statemate [12] are commercial tools based on

the use of statecharts [13], appropriate to specify intra object behavior. The OO-Method [14] combines OASIS and UML [15], and the TROLL Workbench [16] and TROLL Tbench [17] tools combine TROLL and OTROLL (based on OMT). These tools can express in a detailed way the static and dynamic aspects of the system, but making use of different charts to express each one.

Our proposal intends to express inter object behavior abstracting from the internal behavior of the system's components, in a unique diagram describing the important static and dynamic features. This unique diagram provides a global view very appropriate for users and designers to understand the system description.

In addition, 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 [18], paraphrasing parts of the conceptual model in natural language [19] or generating explanations from the specification [20], but simulation by means of the model execution is the technique that better 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 to be executed [16,17,21]. That can provoke lack of precision and fidelity between both representations due to the different abstraction levels of the languages [22]. The use of a formal language like Maude allowing the execution of formal specification avoids that problem.

3. Context

In this section, first, an overview describing the main steps of our proposal is presented. Next, the *Maude* formal language and the motivations of its use in this context are briefly outlined. Finally the objectives of IRDs are described to introduce the next section.

The aim of this work is to integrate a set of techniques and tools to make the description, the development and the validation of software systems with important coordination constraints easier. All this focuses on the separation of the coordination aspect promoting the reutilization of software components.

Figure 1 shows the main steps of the environment: 1) the main system components, their external interface and the negotiation rules of the system are expressed using IRDs. 2) The IRD has a Maude representation allowing to check for the agreement of a detailed Maude specification with regard to the original requirements expressed in the IRD. This representation is the entry to 3) the system specification in Maude. 4) The behavior simulation of the system can be tested and validated in each refinement iteration of the specifications as well as being in accordance with the formal representation of IRD by means of 5) the checker.

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 environment. *Maude* specifications can be executed by means of its rewrite engine, which facilitates its use for prototyping and for checking the specifications behavior [23].

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, rewriting rules can also be defined. *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 parameterized modules by means of views and theories.

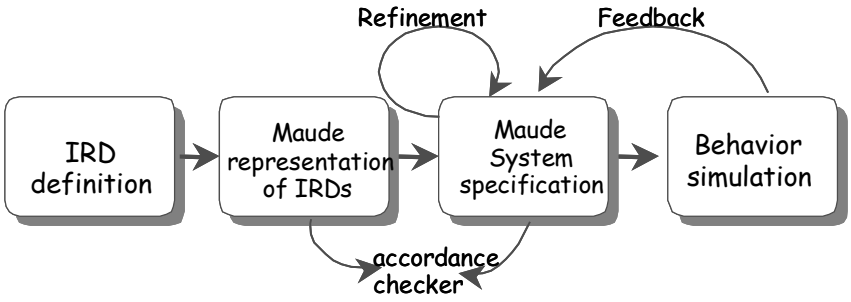


Figure 1. Main steps of the proposal

The clarity of the language, its wide range of application, its executability and its reflection facilitates by modules provided in the environment, have been decisive to select *Maude* as formal language in our context.

Finally, *IRDs* can be used to represent the negotiation rules in a separate way, by means of interrelations between objects in a system, independently of the coordination model adopted in the design phase. In this work, the first aim was to translate the restrictions imposed by the coordination models to early phases in software life cycle. Exogenous coordination models promoting the separation of functional and coordination aspects were considered. In particular, attention was focused on *Coordinated Roles (CR)*[24] because of its special adaptation. That model, based on IWIM [25], makes use of Event Notification Protocols to coordinate the different components of a system in a transparent way. In fact, the considered events in this model *(reception of a message*, *beginning of the processing of a message*, *end of the processing of a message* and *state researched*) have inspired the considered events in the relation constraints of *IRDs*.

4. Interelement Requirement Diagram

In the early phases of the software development process, the scope, objectives and constraints of the future system have to be described. 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. A diagram construction, in this way, would be the previous step to define a formal model of the system and it could be very useful to identify the candidate elements to be reused. With the aim of achieving an easy graphic representation to express initial system requirements we propose a new kind of diagrams named Interelement Requirement Diagrams (*IRD*).
The aim under *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 graphic where the following topics are expressed: main system elements, their global behavior, how they are related and how they answer to specific stimulus, and how different features in their context are expressed. The same representation contains the system's static (the element's observable structure and external interface) and dynamic (relations describing the system behavior) aspects. Static aspects are expressed by means of nodes named elements, their observable external actions (operations 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 if the action invocations are attended to.

Next, an example is introduced, in order to clarify the characteristics and components of an IRD.

4.1. Museum example

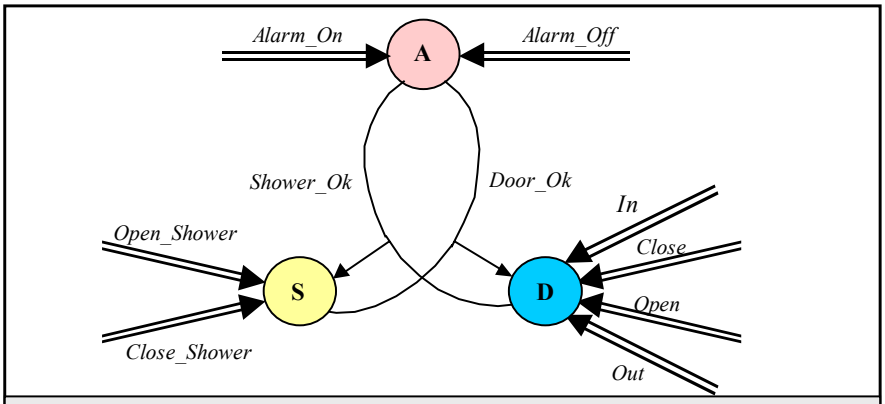
The example presents a fire control system for a museum showroom. The showroom has a smoke detector connected to several elements: an alarm, a shower and an access door to the room. When sensors detect smoke, send messages to the elements, invoking the actions to switch on the alarm, to open the shower and to close the door. However, all these actions must be coordinated to avoid people may be trapped in the showroom or the shower opens before the showroom has been evacuated.

The system alarm can perform the actions *Alarm_On* and *Alarm_Off*. The smoke detector will invoke these actions when the smoke is detected and stops being detected respectively. The shower can do the actions *Open_Shower* and *Close_Shower*. These actions will be also invoked when there is smoke or not, respectively. The door behavior is a little more complex. The door detects when people come in or out in the showroom by means of sensors. The sensors send messages invoking the action *In* or *Out* that increases or decreases respectively the number of persons in the room. The door can also do the actions *Close* and *Open* that are invoked when there is or not smoke. The *Close* action can be executed if there are no people in the showroom and the door is opened. Otherwise the *Close* action will not have effect. *Close* or *Open* actions modify the door state if they can be performed.

When the smoke sensor activates the alarm, and there are no people in the showroom, the door will be closed. Then the shower will be switched on. Just when the smoke stops being detected, the shower and the alarm will be switched off and after the two actions occur, the door will be open automatically.

Figure2 shows the example IRD. Each element is represented as a node in the graph. Thus, the museum IRD has three elements: (A) the alarm, (D) the showroom door and (S) the shower. An element belongs to a specific class of elements (i.e. the D

element belongs to the *Door* class of element). For each class of elements its observable structure is described when the first element of the class is specified. In this definition the following features can be expressed: a state, a list of values and a list of actions. In this way, when a new elements belonging to a specified class are defined, it is only necessary to express which is their class.



Element D **is** ::Door.
State closed **:** Bool
Value inside
Actions In { *inside* ++}
Out { *inside* --}
Close {if *closed* == false and *inside*= = 0 then *closed* = true}
Open {if *closed* == true then *closed* = false}

Element A **is** :: Alarm **Actions** Alarm_On Alarm_Off	**Element** S **is** :: Shower **Actions** Open_Shower Close_Shower

Relation Shower_Ok
Constraint State closed == false **and End** Alarm_On **then** Open_Shower **mode** synchronous

Relation Door_Ok
Constraint End Alarm_Off **and End** Close_Shower **then** Open **mode** synchronous

Figure2. IRD of the Museum example.

The state is defined if the system behavior description refers to it. In the example, only the Door element specifies a state that is referred to in the *Shower_Ok* relation. Values represent variables or attributes of the elements that can be operated or changed by actions in the element; and they cannot be referred out of the element definition. The list of actions represents the operations that the element can perform and can be externally invoked (represented as a double arrow to the element). If the actions modify the state or the values specified as part of its internal behavior, that feature must be specified making use of sentences. The sentences change the state and/or values defined in an element and can be performed if a condition is satisfied. The condition can ask about the content of the state and values. So, in the example all the actions in the Door description define sentences modifying its value and its state.

The dependencies between elements are expressed by means of relations. In the example there are two dependency relations: *Shower_Ok* defining the sequences of actions to perform when there is smoke, and *Door_Ok*, defining the system behavior when the smoke stops is no longer detected. Both relations need the conditions imposed by their two origins to be satisfied. In the case of the *Shower_Ok* relation, the *Open_Shower* action will be allowed if the *Alarm_On* action has ended and the door state is closed. In the *Door_Ok* relation the *Open* action is allowed if the *Open_Shower* and the *Alarm_Off* actions have been performed. So, in a relation the following can be expressed: the conditions that must be satisfied in the origin and/or final element(s), the events triggered, the actions to perform in the final element(s) and the mode (synchronous or asynchronous) in which each action is invoked. Events constrain the actions to be performed in the destination elements of a relation and can be triggered from different elements (origins of the dependency relation). In the *Shower_Ok* relation, the events state changes in the *Door* element, and the end of processing of the *Alarm_On* action in the Alarm element constrains the execution of the *Open-Shower* action in the Shower element. Thus, in this relation there are two origins D and A, and one destination element S.

5. Formal representation of IRD

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 the advantages of this representation and avoid the disadvantages, the semantic of each element in the IRD has been defined using the *Maude* formal language. To facilitate that representation, several object modules have been defined to be included. Some of then are shown below.

```
(omod DefELEMENT is

      protecting MACHINE-INT .

      protecting QID .

      sorts DefState ListVal Val NameState Atrib .

      subsorts Val < ListVal .

      subsorts Qid < Oid NameState .

      subsorts MachineInt Bool < Val Atrib .

      class Elmt | State : DefState .

      op null : -> DefState .
```

```
    op Ste : NameState ListVal -> DefState .

    op <_;_> : ListVal Val -> ListVal .

    op <_:_> : MachineInt MachineInt -> ListVal .

    ops <_:> <:_> : MachineInt -> ListVal .

endom)
```

The DefELEMENT module defines the *Elmt* class, and all elements in an *IRD* are instances of this class. The attribute *State* can be null when an element has no state declared. Otherwise it will have a name and a value or list of possible values. A *sort* named *Attrib* is a generic type that can be use to define any attribute in a specific element if has no a predefined *sort*. In this module, two predefined *Maude* modules are imported, MACHINE-INT and QID, to use the definitions and operations of integer numbers and quoted identifiers respectively.

```
(omod DefRELATION is

  protecting DefSET .

  protecting DefELEMENT .

  sorts DefCons ListCons ListAct NotEvent Mode .

  class Relation | From : DefSet, To : DefSet,

                   Constraint : DefCons .

  op __ : ListCons ListAct -> DefCons .

  ops RM BoP EoP : -> NotEvent .

  op Event : NotEvent Msg -> ListCons .

  op Event_._ : Oid NameState -> ListCons .

  op Event_._==_ : Oid NameState Val -> ListCons .

  op Event_._=/=_ : Oid NameState Val -> ListCons .

  op Eventnot_._: Oid NameState -> ListCons .

  op Eventnot_._==_ : Oid NameState Val -> ListCons .

  op Eventnot_._=/=_ : Oid NameState Val -> ListCons .

  op <_or_> <_and_> : ListCons ListCons -> ListCons .
```

```
   ops sync async: -> Mode .

   op Action : Msg Mode -> ListAct .

   ops <_and_> <_or_>  : ListAct ListAct -> ListAct .

   ops <_orthen_> <_;_>: ListAct ListAct -> ListAct .

   ops <_else_> <_xor_>: ListAct ListAct -> ListAct .

endom)
```

DefRELATION module is used to define constraint relations in an *IRD*. It imports the above DefSET module and declares the Relation class. All constrains relations of an *IRD* are instances of this class . The attributes *From* and *To* indicate the origin and the end element or set of elements of the relation, and the *Constraint* attribute indicates the conditions imposed by the relation. The operations show how to express the different event notification modes and the priority and/or the messages necessary to perform the actions imposed by the relation.

Next, in DefIRD, the class IRD is used to define an *IRD*. It has two attributes representing the set of elements and the set of relations in the *IRD*.

```
(omod DefIRD is

   protecting DefSET .

   class IRD | ElmtSet : DefSet , RelSet : DefSet .

endom)
```

5.1. Formal representation of the Museum example

Performing to the Museum example, the different modules composing the formal specification of the system are defined.. The module ELMT_Alarm represents the Alarm class of element.

```
(omod ELMT_Alarm is

     protecting DefELEMENT .

     class Alarm .

     subclass Alarm < Elmt .

     msgs Alarm_On Alarm_Off : Oid -> Msg .

endom)
```

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. Alarm is an *Elmt* subclass. The two actions Alarm_On and Alarm_Off are represented as messages. No attributes and State are defined in this element.

In the same way, the ELMT_Shower module is defined. with the *Open_Shower* and the *Close_Shower* messages representing the corresponding actions.

```
(omod ELMT_Shower is

     protecting DefELEMENT .

     class Shower .

     subclass Shower < Elmt .

     msgs Open_Shower Close_Shower : Oid -> Msg .


endom)
```

The *Door* element definition in the *IRD* has four actions, the *closed* state typed Bool and a value named *inside*. The actions are formally represented as messages, but in this case the actions define sentences. When sentences are defined in an action, those are represented in *Maude* as rewriting rules. Consequently, there is a rewriting rule for each action. The left side in the rule represents the action invoked and the configuration of the elements to apply the rule. The right side in the rule represents the changes of the element after applying it. Conditional sentences are represented by conditional rules.

```
(omod ELMT_Door is

     protecting DefELEMENT .

     class Door | inside : Atrib .

     subclass Door < Elmt .

     msgs In Out : Oid -> Msg .

     msgs Open Close : Oid -> Msg .

     var D : Oid .

     var A : Atrib .

     rl[in] : In(D) < D : Door | inside : A >

     => < D : Door | inside : (A + 1) > .
```

```
    rl[out] : Out(D) < D : Door | inside : A >
    => < D : Door | inside : (A - 1) > .
    rl[close] : Close(D) < D : Door |
          State : Ste (•closed , false) , inside : 0 >
    => < D : Door | State : Ste (•closed , true) > .
    rl[open] : Out(D)
    < D : Door | State : Ste (•closed , true) >
    => < D : Door | State : Ste (•closed , false ) > .
endom)
```

So, the *in* labeled rule is applied when an *In* message is invoked; in this case the *inside* value is incremented by one. In the same way, the *out* labeled rule acts on the contrary, when an *Out* message is invoked. *Close* and *open* rules are applied only if the corresponding actions are invoked and the conditions imposed by the state and value of the *Door* are satisfied. To apply the *close* rule it is necessary the *inside* value is 0 and the *closed* state is false. Only in this case the state is switched to true (the door will be closed). On the other hand, the *open* rule will be applied when the action is invoked, and if the *closed* state of the door is true, changing the state value to false.

Another object module represents the complete *IRD* in *Maude,* representing the instances of the *IRD* class where the set elements and the set of relations are defined. In the example, the *IRD_Museum* module represents the concrete instances of the elements above defined and their relations.

```
(omod IRD_Museum is
   protecting DefIRD .
   protecting DefRELATION .
   protecting CONFIGURATION .
   protecting ELMT_Shower .
   protecting ELMT_Door .
   protecting ELMT_Alarm .
   subsort Qid < Oid .
   op init : -> Configuration .
   eq  init  = < 'S : Shower | State : null >
```

```
      < 'A : Alarm | State : null >
      < 'D : Door | State : Ste ('closed , false)
                  , inside : 0 >
      < 'Shower_Ok :  Relation | From : 'A 'D , To : 'S ,
            Constraint: < Event 'D. 'closed and Event
            (EoP , Alarm_On('A)) >
            Action (Open_Shower ('S) , sync ) >
      < 'Door_Ok : Relation | From : 'A ' S , To : 'D ,
            Constraint: < Event (EoP , Alarm_Off('A) and
            Event ( EoP , Close_Shower('S)) >
            Action (Open ('D) , sync ) >
      < 'Museum : DRI | ElmtSet : 'A 'D 'S ,
            RelSet : 'Door_Ok 'Shower_Ok > .
endom)
```

It is necessary to import all the above element definitions and the auxiliary DefIRD and DefRELATION modules. CONFIGURATION is a predefined *Maude* module to represent a specific object configuration. The *Init* operation results in a system configuration, the associated equation creates instances of each element in the Museum *IRD*. The *'A* alarm and the *'S* shower elements have null state attribute because they have no state declared. The *'D* door element has a typical initial configuration when the Museum is opened with the *closed* state set *false* and the *inside* value set *0* (there are nobody in the showroom). The two constraint relations in *IRD* are declared now. The *'Shower_Ok* relation indicates the origin of the relation in the *From* attribute. In this case there are two origins: the *'A* and the *'D* elements. The *'S* final element of the relation is indicated by the *To* attribute. The *Constraint* attribute expresses the necessary conditions to perform the *Open_Shower* action. Two conditions must be satisfied to execute that action; the closed state of the *'D* door element must be true and the end of the processing event over *Alarm_On* message to *'A* alarm element must have happened. The processing mode has to be synchronous: the door and the alarm will be locked until the *Open_Shower* action can be processed. The *'Door_Ok* relation acts in the same way; it has two origin elements and a final element represented in the attributes. Its constraint also has two events that must occur to allow the execution of the *Open* action in the shower element in synchronous mode. These events are the end of processing the *Alarm_Off* action in the alarm element and the end of the processing of the *Close_Shower* in the shower element.

Finally the '*Museum* object of *IRD* class is defined with the set of the door, shower and alarm elements and the set of both relations.

In this way, the coordination constraints imposed by the relations between elements are specified separately of the elements. So, this provides several advantages. On the one hand, it is making the reusability of the IRDs easier, changing the constraints imposed by the dependency relations without modifying the element specifications. On the other hand, this representation allows one to focus on the negotiation rules of the system abstracting of the component specifications.

6. Future works

Currently, we are developing the tool supporting the creation of *IRDs* and a checker that determines whether the system's refined specifications in Maude reflect only all the elements and their relations expressed in the original *IRD*. The checker has to guarantee that all features in the specification correspond to features in the *IRD,* and all features in the *IRD* are presented in the specification through the successive refinements and changes in the development process.

The relations between the system specification and the corresponding *IRD* have been defined considering that in the detailed specification internal operations and values that are not represented in the *IRD* can appear. In such case, the new features only must reflect internal behavior and must not affect interelement relations or their constraints.

The checker is being developed using Maude. The META-LEVEL predefined module in Maude, facilitates the use of the reflective properties of the language, simplifying the work with terms and modules in the same language.

The next objective is the generation of executable specifications from the system's *IRD* with the aim to validate the system's behavior. In order to generate specifications reflecting the intrinsic characteristics of cooperation environments, which are easier to understand by designers, we consider more appropriate the generation of specifications making use of CoordMaude (a set of primitives *Maude* that we are developed allowing to use the syntax of *Coordinated Roles*[20] to generate *Maude* specifications in a simple, clear and short way).

7. Conclusions

This work explains a technique to represent both visually and formally, the dependency requirements between different elements in a system. Interelement Requirements Diagrams facilitate the descriptions and representations of relations between elements in a transparent manner with regards to the internal behavior of each element. The advantages of this representation are: 1) simplicity in the construction of systems by means of components composing, because the coordination dependencies are specified separately from the components 2) the changes in dependency policies can be easily expressed 3) usefulness in the representation of open and distributed systems where the elements configuration, the

system and their relations are variable and 4) a unique and single representation of the system, expressing static and dynamic aspects.

The correspondence between an *IRD* and their representation using an executable algebraic language provides a means to formally specify the *IRD* artifacts. So, it can be used to validate the global system behavior executing those formal specifications from a particular system configuration and simulating a set of event occurrences. Moreover, that representation can be used to verify whether later specification refinements are in accordance with the initial requirements represented by the *IRD*. And a representation better oriented to designers can be generated, making use of the coordination model based on the separation of the functional and coordination aspects.

8. References

1. C. Rolland and C. Cauvet. "*Trends and Perspectives in Conceptual Modelling"*. In P. Loucopoulos and R. Zicari (eds.), Conceptual Modeling, Databases and CASE, Chapter 1, John Wiley and Sons, Inc. , 1992.
2. R. Kneuper. *"Limits of Formal Methods"* Formal Aspects of Computing. Vol. 9 , pags: 379-394, 1997.
3. A. Gravell and P. Henderson. Executing Formal Specifications Need Not Be Harmful. Software Engineering Journal vol. 11 nº 2, 1996.
4. 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.
5. E. Cusack, S. Rudkin and C. Smith. An Object-Oriented Interpretation of LOTOS. In S.Vuong (ed.) Formal Description Techniques II (FORTE'89), Amsterdam, 1990.
6. K. Lano. Z++: an Object-Oriented Extension to Z. In J. Nicholls, ed., Z Users Workshop: Proc. of 4th Annual Z User Meeting. Springer-Verlag 1991
7. E. H. Dürr and J. Katwijk. VDM++. A Formal Specification Language for Object-Oriented Design. In Proc. of TOOLS7 , Technology of Object-Oriented Languages and Systems. Prentice-Hall,1992.
8. P. Du Bois. The Albert II Language. On the Design and the Use of a Formal Specification Language for Requirements Analysis. PhD thesis, University of Namur, Belgium, 1995.
9. R. Jungclaus, G. Saake, T. Hartmann and C. Sernadas. TROLL. A Language for Object-Oriented Specification of Information Systems ACM Transactions on Information Systems vol.14 nº2, April 1996, pags 175-211.
10. P. Letelier, I. Ramos P. Sánchez and O. Pastor. OASIS v3.0: Un Enfoque Formal para el Modelado Conceptual Orientado a Objeto. Universidad Politécnica de Valencia. 1998.
11. Rhapsody. I-Logix Inc., Andover, MA, 1999.
12. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. ACM Transactions on Software Engineering and Methodology, vol. 5 nº 4, pags. 293-333. 1996.
13. D. Harel. Statecharts: A visual formalism for complex systems. Science of Computer Programming, vol 8. pags:231-274, 1987.
14. O. Pastor, V. Pelechano, E. Insfrán and J. Gómez. From Object-Oriented Conceptual Modeling to Automated Programming in Java. In T.W. Ling, S. Ram and M. L. Lee (eds.) Proc. of the 17th Int. Conf. On Conceptual Modeling (ER'98) LNCS 1507, 1998.N. E.
15. J.Rumbaugh, I. Jacobson, G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

16. A.Grau. Computer-Aided Validation of Formal Conceptual Models PhD. Thesis. Technischen Universität Braunschweig. March 2001.
17. J.Kusch, P.Hartel, T.Hartmann and G.Saake. 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 1995.
18. D. Kung. "The Behavior Network Model for Conceptual Information Modelling". Information Systems, vol. 18, nº 1 pags:1-21, 1993.
19. H. Dalianis. "*A Method for Validating a Conceptual Model by Natural Language Discourse Generation*". In P. Loucopoulos (ed.) Proc. of Int. Conf. On Advanced System s Engineering (CAISE'92) Springer, LNCS 593, pags: 425-444, 1992.
20. J. A. Gulla. "*A Genera Explanation Component for Conceptual Modelling in CASE Environments*". ACM Transactions on Information Systems vol. 14, nº 2, pags:297-329, 1996.
21. P. Letelier. "Animación Automática de Especificaciones OASIS utilizando Programación Lógica Concurrente". Tesis Doctoral, Universidad Politécnica de Valencia, 1999.
22. I.J. Hayes and C.B. Jones. "*Specifications are not (necessarily) executable".* In Software Engineering Journal Vol. 4 nº6 pags:320-338, 1989.
23. Marisol Sánchez, José L. Herrero, Juan M. Murillo, Juan Hernández. Guaranteeing Coherent Software System when Composing Coordinating Systems. A. Porto and C. Roman (Eds.). Fourth Int. Conference COORDINATION'2000. LNCS 1906. 2000.
24. J. M. Murillo. Coordinated Roles: un modelo de coordinación de objetos activos. PhD. Thesis. University of Extremadura, 2001.
25. F. Arbab. The IWIM Model for Coordination of Concurrent Activities. P. Ciancarini, C. Hankin (Eds.). First International conference Coordination'96. LNCS 1061. 1996.