Coordination models and Software Architectures in a Unified Software Development Process

Paola Inverardi[1] and Henry Muccini[1]

Dipartimento di Matematica
Universitá dell'Aquila
Via Vetoio, 1 - 67100 L'Aquila, Italy
Voice +39 0862 4331{27,89}
Fax +39 0862 433180
email{inverard, muccini}@univaq.it

Abstract Coordination models and Software Architectures (SAs) have been recognized as valid tools to manage complex distributed systems. Coordination models and languages provide a specification level description of processes interaction, separating control issues from the computational concerns. Software Architectures provide a high level description of software components interactions designing applications as sets of autonomous, decoupled components. In this work we are going to present how Coordination models and SA can be linked together and how they can be usefully integrated in a UML-based software development process. In particular, we will see that the dynamic description provided by the SA also allows for validating coordination policies. A real-world case study will be used to discuss, in detail, the approach previously introduced ...

1 Introduction

In recent years, there has been a growing interest in software architectures (SAs) and coordination models. Although they play different roles in the software development life cycle strong similarities and analogies in concepts and finalities seem to hold [8]. SAs represent the first design step in which a complete system model is provided. They focus on the overall organization of a large software system (the glue) using abstractions of individual components and explicity modeling their interaction. Their description encompass both static and dynamic aspects of the system under design, from topology to inter-components communication descriptions. Coordination models instead, come in at a later development stage in order to provide a precise system specification. Their focus however is similar to SA since coordination models are specialized to describe process interaction in a concurrent environment abstracting away the details of computation and focusing on the interactions [3]. As a matter of fact at the SA description level, many important design choices related to the way components interact, are already taken. Thus (see Figure 1) SA level information can influence the static and dynamic structure of the implemented system and drive/constrain the coordination model specification.

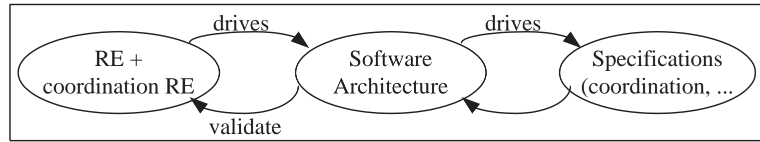

Figure 1. SA description drives Coordination models

In this work we are going to describe an extension to an our previous work [12] proposing a way to profitably and coherently relate these two system development phases. We cast our approach in a UML software development process [13]. Although UML original purpose was for detailed design, UML extension mechanisms makes it potentially applicable much more broadly. Recent works [13], [15] describe how UML Diagrams can correctly and completely describe each step in the software development life-cycle, from Requirements capturing to Design model description passing through a Software Architectural description [16], [6], [11]. Our approach allows us to use the SA model to validate coordination requirements and to drive the generation of a correct with respect to requirements coordination model. The approach we present is divided in the following steps: requirement level information is formalized using UML diagrams and guides software architectural items definition; the SA dynamic model is used to validate requirements; architectural level information drives coordination models specification.

The paper is organized as follows: section 2 introduces the Unified Software Development Process and summarizes the generation of an analysis model starting from Use Cases; subsection 2.1 describes a case study and shows the case study analysis model; section 3 introduces software architectures. The approach is analyzed in section 4 and applied to the case study in section 5. Section 6 contains conclusions and future works.

2 The Unified Software Development Process

A process defines *who* is doing *what*, *when* and *how* to reach a certain goal and serves as a guide for all participants in the software development. The Unified Process is the result of at least ten years of experiences in the development of software systems [13]. It is Use-Case driven, Architecture-Centric and, Iterative and Incremental. In this section we summarize only the process steps relevant for our approach: requirement capturing as Use Cases and Analysis model of Use Cases.

In the Unified Process, Use Cases drive the evolution of each development step. They capture functional requirements while Use Case Diagrams show how Use Cases are associated to Actors. *Use cases* represent a possible way of using the system while *actors* are who or what (humans or a subsystems) carry out use cases. Each user needs several different use cases, each representing the different ways he or she uses the system.

As a bridge between Requirements and Design Models an Analysis Model is generated to *realize* Use Cases. An Analysis model is a stereotyped [10], [15] UML class diagram; it is composed of Analysis classes describing *how* a specific use case is realized in terms of "abstract" cooperating classes and associations between them. Analysis classes (in the following also simple called *classes*) always fit one of three basic stereotypes: **boundary**, **control** or **entity** (their means will be described in the approach section). To create this model a set of use cases is selected and the analysis classes that participate in his realization are identified. We typically begin by looking at a few use cases and gradually build a stable model. As a result, each class may participate and play roles in several use-case realizations.

So far, only the structure of objects implementing Use Cases has been found; nothing has been said on the interaction patterns. To understand how use cases are used by actors to evolve in the system global behavior, Interaction Diagrams can be used. In particular, Collaboration Diagrams show how the focus moves from object to object.

In Section 5 we will show how an analysis model of the case study presented in the following section could be realized.

2.1 The TRMCS case study: The Requirements and the Analysis Model

The Teleservice and Remote Medical Care System (TRMCS) [4] is a system providing monitoring and assistance to users with specific needs, like disabled or elderly people. A typical service is to send relevant information to a local phone-center so that the family, medical or technical assistance can be timely notified of critical circumstances. The TRMCS is being developed at Parco Scientifico e Tecnologico d'Abruzzo, and currently a Java prototype is running and undergoes SA based integration testing.

The basic **functional** and **non functional** requirements on the system were to allow enabled users to *send help messages*, guarantee the *termination* of the service (hw and sw fault-freeness), to guarantee the *continuity* of the service (24 hours a day, for every day), to *optimize* the reply time and to reduce the service *cost*. Working on these basic requirements, we have identified the main needed functionalities, that are:

- Alarm management;
- Ack management;
- Check that User's signaler correctly works;
- Log files for Alarm and Check messages;
- User's management capabilities;
- Hardware fault tolerance management;
- Check User's rights to the service;
- Administrative features.

and the *coordination policies* required:

1. The system can be used concurrently by n users;
2. An Alarm message sent from User has to be followed by an acknowledgment message;
3. k Check and p Alarm messages have to be concurrently managed.

The next step is to identify actors that carry on the various functions taking place in the system:

- User that represents the disabled or elderly people and sends messages and waits for acknowledgments;
- Router, an intermediate communication level that serves the purpose of dispatching User messages to the Server;
- Server, that provides the service requested by the User.

In the following figure (Fig. 2) we show the UML Use Case Diagram that represents Actors and their functionalities.

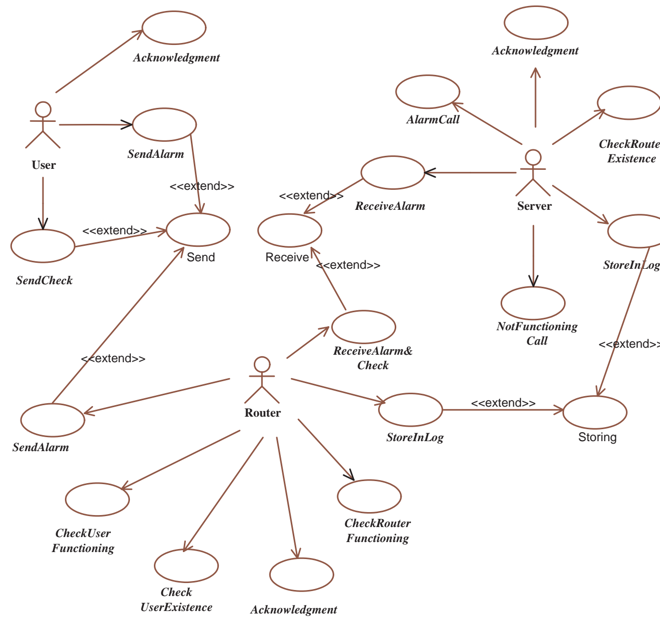

Figure 2. TRMCS Use-Case Diagram

The analysis model realizing the Use Cases shown in Fig. 2 is shown in Figure 3. User(s), Router and Server are the system actors, CheckRequestUI and AlarmRequestUI are the boundary classes, CheckHandler, Alarm and Ack Handler and ErrorHandler are the control classes while UserDbase, RouterDbase and LogFile are entity classes.

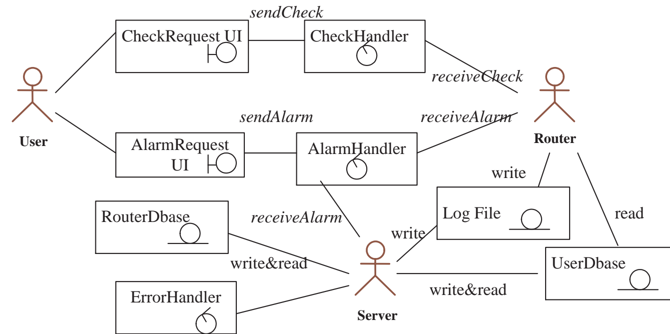

Figure 3. TRMCS Analysis model

Putting attention only to the main system functionalities (those shown in Fig. 2), the dynamic interaction can be described as below:

the system USER can (non deterministically) choose to SEND AN ALARM message or a CHECK MESSAGE to a component named ROUTER. The Router is always WAITING FOR ALARMS and CHECKS. If an alarm message is received by Router, it checks if the user is eligible for the service then STOREs the alarm request in a log file. If the alarm request is correctly processed by the Server component (to which the alarm is forwarded) an ACK MESSAGE is sent back to the User. If a check message is received, by Router, it is STORED in a log file; in the case an ERROR OCCURs, this has to be handled by the SERVER.

Interaction diagrams can be drawn on the Analysis model objects to represent possible execution scenarios (Figures 4, 5).

Figure 4 shows what happens when a User sends either an alarm or a check and the operation is succesfully managed.

Figure 5 shows what happens when two users are competing to send an Alarm (5.a) and when a user sends an Alarm and another one sends a check msg (5.b).

These scenarios also implicitly embodies the coordination policies; Figure 5 shows a possible system behavior when n=2 (look at coordination policies number 1); Figure 4 is correct with respect to coordination policy 2 and Figure 5 implements policy 3.

3 Software Architecture

In recent years the study of software architecture (SA) has emerged as an autonomous discipline requiring its own concepts, formalisms, methods and tools [11], [5], [17]. The originality of the SA approach is to focus on the overall organization of a large software system (the glue) using abstractions of individual components. This approach makes it possible to design and apply tractable methods for the development, analysis, validation, and maintenance of large software systems. Software architectures represent the overall system structure by modeling

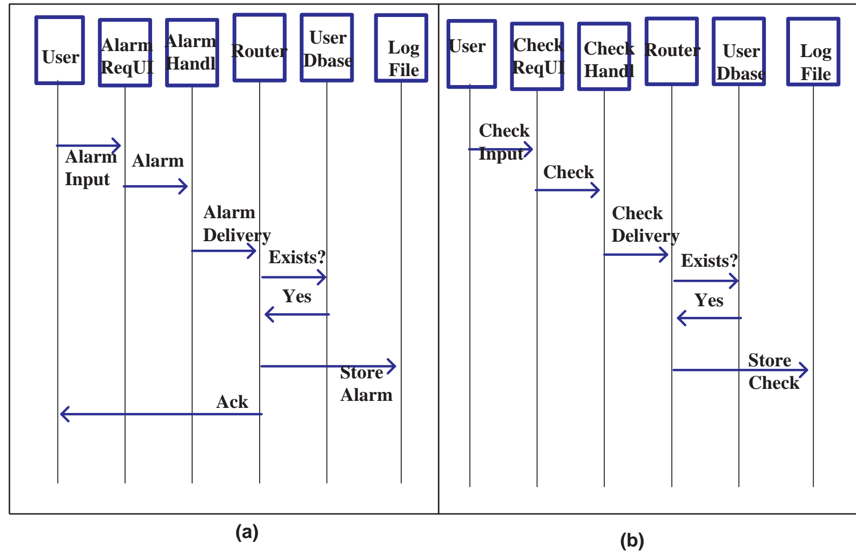

Figure 4. a) Alarm scenario; b) Check scenario

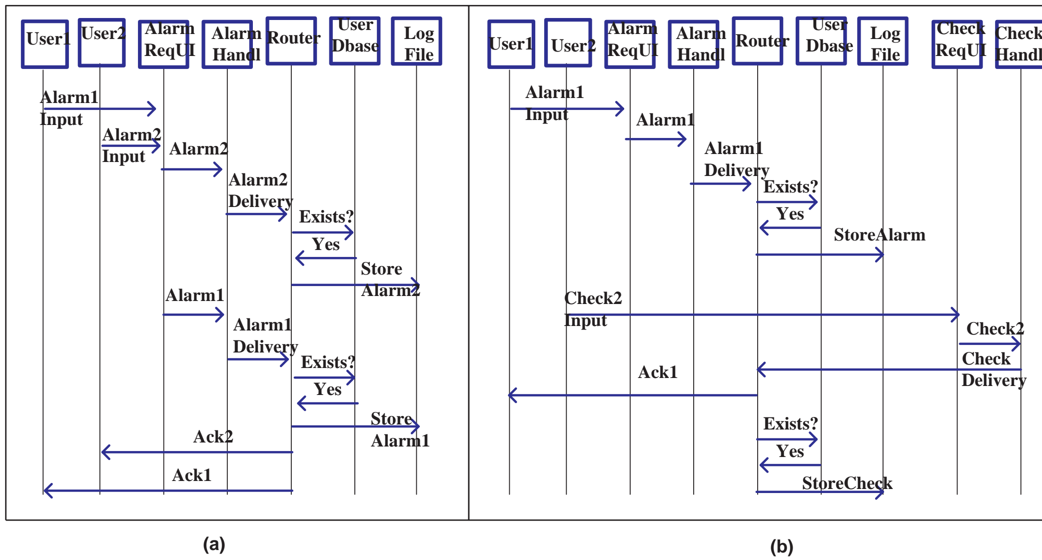

Figure 5. a) Alarm from two users; b) Alarm and Check from different users

individual components and their interactions. Thus, they naturally provide the right level of abstraction to express the logical coordination structure of complex distributed software.

SAs support the formal modeling of a system allowing for both a topological (static) description and a behavioral (dynamic) one. The behavioral description provides means for an early in the life cycle validation of design choices, while the topological one can be used to carry on the development steps in a refinement (top-down) fashion, or to identify architectural components in a bottom-up fashion.

For these reasons, SAs have recently been considered, both by academia and the software industry, as a means to improve the dependability of large complex software products while reducing development times and costs. Software architectures are seen by many as the most promising approach to tackle the cumbersome problem of *scaling* in software engineering, because they provide the means to make large applications manageable through suitable abstractions.

Our approach assumes the existence of a formal definition of the Software Architecture encompassing both the static and the dynamic descriptions [9], [1], [14], [7]. We assume that from an architectural description a Labelled Transition System (LTS) (see definition 1) can be derived, whose node and arc labels represent respectively states and transitions relevant in the context of the SA dynamics. We also assume that states contain information about the single state of components and that labels on arcs denote relevant system state transitions.

Definition 1. A Labelled Transition System is a quintuple $(\mathcal{S}, \mathcal{L}, S_0, \mathcal{S}_\mathcal{F}, \mathcal{T})$, where $\mathcal{S}$ is the set of states, $\mathcal{L}$ is the set of labels, $S_0 \in \mathcal{S}$ is the initial state, $\mathcal{S}_\mathcal{F} \subseteq \mathcal{S}$ is the set of final states and $\mathcal{T} = \{\stackrel{l}{\longrightarrow} \subseteq \mathcal{S} \times \mathcal{S} \mid l \in \mathcal{L}\}$ is the transition relation.

The initial LTS state corresponds to the initial, static configuration of the system. Transition rules applied to the initial configuration define how the system dynamically evolves from its initial configuration, the label on an arc usually represents the event associated to the transition denoted by that arc. The set of final solutions represents the different possible states of the system in which the computation is considered to have completed. We also need the definition of a *complete path* over an LTS:

Definition 2. Let

$$p = S_0 \stackrel{l_1}{\longrightarrow} S_1 \stackrel{l_2}{\longrightarrow} S_2 \stackrel{l_3}{\longrightarrow} \ldots \stackrel{l_n}{\longrightarrow} S_n$$

be a path in an LTS. p is complete if S_0 is the initial state and S_n is a final one.

4 The Approach

In section 2 we have shown how an high level class diagram can be generated by Use Cases. In section 3, we have described SAs main concepts and how they

can be used. In this section we are going to explain how a Software Architecture description can be related to the use of UML Diagrams and how this model can be subsequently used to drive a coordination model specification of the system (Figure 6). We also describe how SA model can be used to validate coordination policies. In Section 5 we will apply our approach to the TRMCS case study.

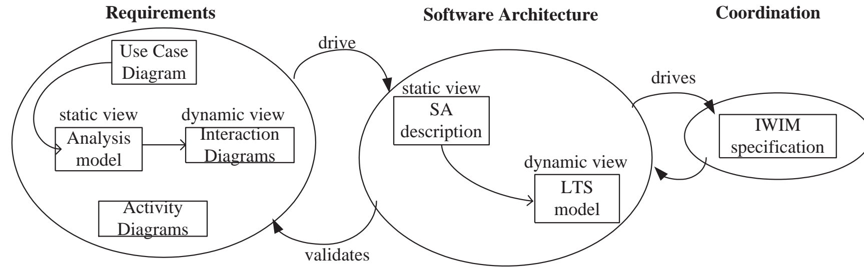

Figure 6. From Requirements to Coordination models

4.1 From UML Diagrams To SA Model

To describe the SA of a software system we need, first of all, to identify the Components and the Connectors.

The Analysis model generated in section 2 and implemented in 2.1 can be a good starting point; each analysis class represents an abstract view of the system and is involved in conceptual relationships. Moreover, analysis classes always fit one of three basic stereotypes with the following semantics:

- boundary classes represent abstractions of windows, forms, communication interfaces;
- entity classes reflect logical data structure;
- control classes represent *coordination, sequencing, transactions and control of other objects* and are often used to encapsulate control related to a specific use case.

This model can be usefully used to model the topological SA description but must be supported by other information to describe the dynamic interactions architectural model: in particular, we have to know how many processes are involved in the system, how they can synchronize. By the use of *Activity Diagrams* we can partially fix these problems putting in evidence how system functionalities can be grouped in several activities and how they can be sequenced.

The underlying idea to turn the Analysis model and the Activity Diagrams into a SA description is the following:

- actors become components;

- control classes become *coordination* components; they will manage the coordination between several components and coordination components;
- each activity put in the Activity Diagram has to be reflected in the SA; they can be used to specialize components or connectors.

4.2 Validating interactions via SA dynamic model

Given the SA topological description, the software architect describes the system dynamics in terms of component and connectors. In our approach the SA dynamics is modeled by a Labelled Transition System (LTS) (as discussed in section 3). Each LTS complete path describes a possible execution scenario so that all LTS complete paths denote the set of all possible system behaviors. We can validate Analysis model SD by model checking them on the SA LTS. The main problem validating Analysis SDs and SA model is that they work on a different set of objects: Analysis model SDs objects are analysis level objects while SA objects SA works on components.

Considering that we have knowledge on how analysis objects are mapped to SA components (as briefly discussed in section 4.1) the idea is to validate the SDs looking at objects that are both in the two representations intersection as we will better see in the example.

4.3 From SA model To IWIM Coordination Model

The IWIM [2] model for coordination is described by *processes*, *ports*, *channels* and events.

A process is a black box operating unit. It ca be regarded as a *worker* process or a *manager* process. The first one can execute elaborations but it is not responsible for communication while the latter coordinates communications among worker and manager processes.

A port is used for the exchange of information; each communicating process own at least one port.

A channel represents the interconnection between a producer process port to a consumer process port. There are five different alternatives for a channel; one is for synchronous communication while the others are useful for asynchronous one.

Events are broadcast in the environment and could be picked up by a process.

We can note that some items are comparable with SA items. Processes and components are black box; a component is the high level description of a IWIM process where a coordination component and a component will became, respectively, a manager and a worker process. SA description is high level because a single SA component can be duplicated in several IWIM processes (look at the application example in section 5). The channel and port semantics in the IWIM model and in a SA model are really close. Each SA channel can be mapped in one of the five IWIM channels semantics. Finally, the IWIM event are comparable with transactions in the LTS model of SA dynamics.

Following these considerations, it is amenable to realize a mapping between the SA description level to the Coordination. In section 5 we will see how the TRMCS SA model could be mapped into an IWIM model. The formalization of this mapping will be of interest for future works.

4.4 Summarizing the approach

Here are listed the steps followed by our approach:

1. - Use Case Diagrams are captured from functional Requirements;
   - Use Cases are realized by the analysis classes;
   - interaction diagrams are drawn to describe the analysis model scenarios;
   - activity diagrams are used to highlight system activities and synchronization;
2. using Analysis level objects and activity diagrams information, SA components and connectors can be selected;
3. - a Labelled Transition System describes the system dynamics in terms of components interactions
   - interaction diagrams and LTS can be compared for a consistency checking;
   - an IWIM model construction can be driven by SA description.

5 Applying the approach to the TRMCS case study

An analysis model can be usefully used to put in evidence items implementing the system functionalities but an activity diagram can be more adequate to look at the system processes. In Figure 7 we describe the Trmcs activity diagram that can be described as follows:

User is composed of three activities: he can send an alarm, send a check or wait for an ack (after an alarm has been sent). The Router is also composed of three activities: he waits for a check an alarm and an ack msg. The Server receives alarms and sends acks and waits for possible User no functioning. It is important to note the needs of a Timer to synchronize events.

Based on the Trmcs analysis model (section 2.1) and on the activity diagram just depicted, we can select the items to be used in the SA description. In particular:

- User, Router and Server analysis classes become SA components. We have to manage multiple instances of User components;
- Check Handler becomes the Check Coordinator component and Alarm plus Ack Handler become Alarm Coordinator component;
- A Timer Coordinator component is introduced as suggested by the activity diagram;
- RouterDbase, UserDbase and Log File are hidden because not relevant in this architectural view;

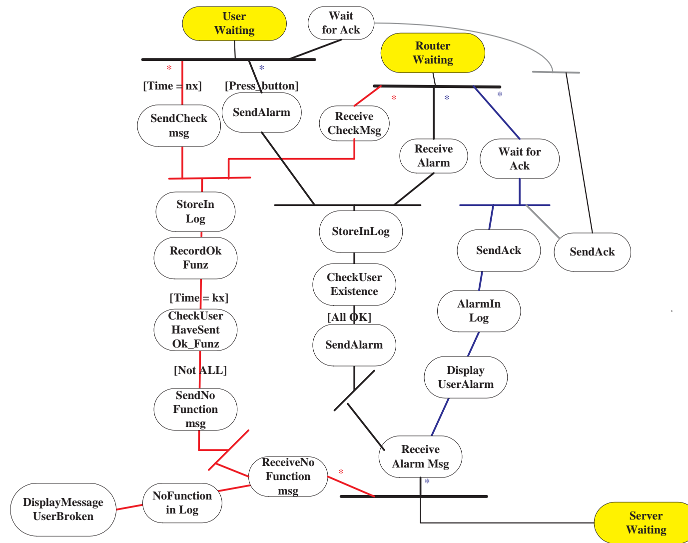

Figure 7. TRMCS Activity Diagram

– Activity diagram information will be used to model the component interaction.

Based on the previous described mapping, the TRMCS topological description is obtained and shown in Fig. 8.

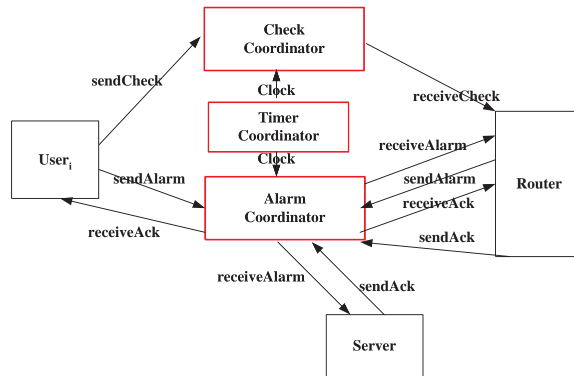

Figure 8. TRMCS Components and Connectors

Considering the case when two users are working on the system, we have build the LTS model of the system dynamics. A portion of it is shown in Figure 9. The whole LTS is around 500 states. Note that arc labels 0, 1, ..., 21 correspond,

respectively, to T_0, T_1, ..., T_{21} (the labels of the TRMCS reaction rules),"0" denotes the initial state and box states denote pointers to states elsewhere shown in the picture (to make the graph more readable). Double arrows denote the points in which the figure cuts LTS paths.

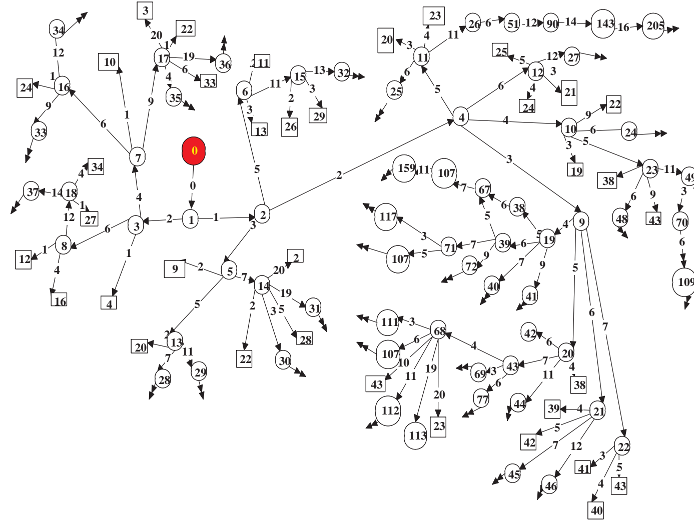

Figure 9. TRMCS LTS

Each LTS complete path represents a possible combination of Alarms, Checks, Clocks and Acks messages exchanged by the system components. An Analysis level scenario is validated by the LTS model if and only if correspondent components [1] follows the same pattern. For example, Figure 10 does not validate Figure 4.a because a finale state is reached and Alarm2 has not received an ack msg. Check and CheckDelivery are not relevant to validate the selected scenario.

Moreover, the IWIM basic items can be found enriching the SA components with the Activity diagram information:

User component is decomposed in UserSendAlarm, UserSendCheck and User-ReceiveAck IWIM worker processes;

Router component is decomposed in RouterWaitAlarm, RouterWaitCheck, RouterWaitAck, RouterSendAlarm, RouterSendAck workers;

Server is decomposed in ServerWaitAlarm and ServerSendAck workers;

Alarm, Check and Time Handler become manager processes;

sendCheck, sendAlarm, sendAck and clock connectors become output ports;

receiveCheck, receiveAlarm and receiveAck connectors become input ports;

[1] User, Router and Server (analysis level) classes correspond to User, Router and Server components, Check Handler and Alarm Handler correspond to Check and Alarm Coordinator components.

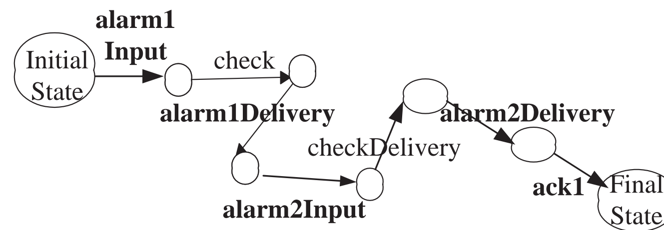

Figure 10. An LTS complete path

Check, Alarm, Ack and Clock are the channel interconnecting send and receive ports.

6 Conclusions and Future Works

In this work we have put in evidence how requirements, software architectures and coordination models can be related and how high level information can be used to drive the specification of coordination models. We have based our approach on a UML-based software development process, extending standard concepts (analysis model) to drive SA and coordination model specification.

As future works, we are going to better understand (and to formalize) how SA topological description can be used to drive Coordination items selection and how high level coordination information can be used to better understand coordination model specification.

References

1. Allen, R., Garlan, D.: A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
2. Arbab, F.: Coordination of massively concurrent activities. CWI Report CS-R9565 (1995).
3. Arbab, F.: What Do You Mean, Coordination? In the March '98 Issue of the *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*. Available at: <http://www.cwi.nl/ farhad/>.
4. Balsamo, S., Inverardi, P., Mangano, C., Russo, F.: Performance Evaluation of a Software Architecture: A Case Study. *IEEE Proc. IWSSD-9*, Ise-Shima, Japan (April 1998).
5. Bass, L., Clements, P., Kazman, R.: Analyzing Development Qualities at the Architectural Level. Software Architecture in Practice. SEI Series in Software Engineering.
6. Booch, G.: Software Architecture and the UML.
   Slides available at <http://www.rational.com/uml/index.jtmpl>.
7. Compare, D., Inverardi, P., Wolf, A. L.: Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming* (33)2 (1999) pp. 101-131.

8. Coordination '99: *Proc. 3rd Int'l Conf. on Coordination Languages and Models*, LNCS 1594, Springer Verlag, April 1999.
9. Darwin, an Architectural Description Language.
   Web site: <http://www-dse.doc.ic.ac.uk/research/darwin/darwin.html>.
10. Eriksson, H.E., Penker, M.: UML Toolkit. John Wiley & Sons, Inc. publishing (1998).
11. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley, (1999).
12. Inverardi, P., Muccini, H.: A Coordination Process Based on UML and a Software Architectural Description To appear on *Proc. of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, session: Coordination Models, Languages, Systems and Applications, June 26 - 29, 2000 Monte Carlo Resort, Las Vegas, Nevada, USA.
13. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison Wesley, Object Technology Series (1999).
14. Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., Mann, W.: Specification and Analysis of System Architecture Using Rapide. *IEEE TSE, Special Issue on Software Architecture*, 21 (4):336-355, April 1995.
15. Rational Corporation: Uml Resource Center. UML documentation, version 1.3. Available from >http://www.rational.com/uml/index.jtmpl>.
16. Robbins, J.E., Medvidovic, N., Redmiles, D.F., Rosenblum, D.S.: Integrating Architecture Description Languages with a Standard Design Method. *Proc. 20th Int'l Conf. on Software Engineering* Apr. 1998, pp. 209-218.
17. Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, New Jersey (1996).