

Checking consistency between architectural models using SPIN

Paola Inverardi & Henry Muccini & Patrizio Pelliccione
Dipartimento di Matematica
Università dell'Aquila - Via Vetoio, 1
67100 L'Aquila, Italy
{inverard, muccini, pellicci}@univaq.it

Abstract

Requirements and Software Architectures are strictly related but only a little attention has been paid to their integration. What we propose in this paper is an approach to i) trace coordination requirements from their definition to the low level specification and ii) validate the architectural dynamic model with respect to these coordination requirements.

1. Introduction

Analyzing current Software Engineering techniques we can note that Requirements Engineering and Software Architectures have been recognized, both by academia and software industries, as a means to improve the dependability of large complex software products, while reducing development times and costs.

Requirement Engineering (RE) is the most developed part of Software Engineering in last years to optimize the study of first stages in the software life cycle. RE applies different theories and methodologies, by means of specific software systems, to formalize the identification, collection and organization of the System requirements. On the other side, new tools and architectural languages have been proposed to specify the Software Architecture (SA) of large scale systems. Software Architecture description represents the first, in the development cycle, complete system description. They provide at the high level of components and connectors both a description of the static structure of the system and a model of its dynamic behavior.

Even if RE formalisms and Software architectural languages have assumed a great interest, it seems that only a little attention has been paid to their integration, increasing the risk of inconsistencies in system development and evolution. Requirements captured in the first stage of system evolution are not always traced to the architectural picture; Software Architectural models, even more used to drive the

design step, are not proven to be correct with respect to the expected behaviors.

In this paper we report our experiences on software architecture models analysis and requirements understanding, based on two previous works: in [9] we put in evidence how coordination requirements can be captured at the architectural level and we give an idea of how the SA model could be proven to be consistent with respect to the requirements; in [11] we present an approach to verify the consistency between statecharts and scenarios models representing the SA dynamics and the coordination constrains respectively.

Putting together these works our aim is to provide an approach able to i) trace coordination requirements from their definition to the low-level specification and ii) validate the architectural dynamic model with respect to these coordination requirements.

In Section 2 we summarize the approach we used in [9] to capture coordination requirements, in Section 3 we give an overview on the approach we presented in [11] to validate statecharts and scenario models. Section 4 presents conclusions and ongoing works.

2. Coordination Requirements and Software Architectures

Software Architectures (SAs) and Coordination models play different roles in the software development life cycle: SAs represent the first design step in which a complete system model is provided, modeling components interactions and encompassing both static and dynamics aspects; Coordination models instead, come in at a later development stage in order to manage the interaction among concurrent programs or activities.

However they work in different domains, strong similarities and analogies in concepts and finalities seem to hold [5], since Coordination models and SA are specialized to *describe process interaction (in a concurrent environment), abstracting away the details of computation and*

focusing on the interactions [2, 16]. 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. Moreover, the SA description can also be an useful tool to better understand system requirements. Following these necessities, our aim is to be able to capture coordination policies at the requirement level, model these policies at the SA level and to use the SA description to drive the generation of a coordination model. Moreover, we are interested in validating our architectural model with respect to the coordination requirements to check the architectural correctness with respect to the expected behaviors.

To gain these goals, an UML-based development process [12] is used: coordination policies are captured at the Requirement level, using Use Case and Interaction Diagrams (Step1), and they are used to *drive* the SA description (Step2). The SA can be *validated* with respect to Coordination requirements (Step3) and can drive the generation of a Coordination model (Step4). In the following we present a summary of the approach, described in [9].

Step1: UML and Requirements

The UML [15] approach to identifying system requirements is mainly based on Use Case Diagrams; *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.

To achieve a more precise understanding of the requirements and structure them for reuse and maintenance an analysis model can be described [12] using *analysis classes* and interacting analysis objects. Analysis classes describe how a specific use case is realized in terms of “abstract” cooperating classes and always fit one of three basic stereotypes: boundary, control or entity;

- boundary classes represents 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 [12].

Each use case can be modeled by analysis classes. Each class may participate and play roles in several use case realizations. A class diagram (of analysis classes) can be drawn to indicate which use case realizations a class participate and plays roles in. This diagram gives an high-level *static* description of the “modules” implementing the use cases

but it does not give information on *how the system evolves* in terms of use cases interactions.

The sequence of actions in a use case begins when an actor invokes the use case by sending some form of message to the system; in the analysis class, the boundary class (i.e., the interface) receives the communication request, sends the request to the control class that coordinates the various activities and lets the involved objects interact to realize the use case. Interaction Diagrams can model the chronological sequences of interactions but only as a sort of *coordination specification constrains*.

Step2: From UML Diagrams To SA Model

The idea is to define a mapping between Analysis model and SA topology, considering that each analysis class represents an abstract view of the system and is involved in conceptual relationships: actors in the Analysis class diagrams represent a suitable abstraction of active components in the SA description; control classes can identify coordination components; control classes attributes can identify communication channels; entity classes can be mapped into databases. Other classes can be hidden at the SA level or mapped to other components.

We can now start modeling the system dynamics: analyzing the Interaction diagrams built in the previous step, we can try to understand how architectural components dynamically interact and we can express the system behavior through an architectural language. We do not define a formal mapping among UML scenarios and the architectural description of the dynamics; what we say is that they drive the architectural modelization and help the software architect to identify components interactions.

Step3: Validating interactions via SA dynamic model

We make the assumption that from an architectural description (in some architectural language) a Labeled Transition System (LTS) can be derived, whose node and arc labels represent respectively states and transitions relevant in the context of the SA dynamics.

Each LTS complete path describes a possible execution scenario so that all LTS complete paths denote the set of all possible system behaviors. The LTS model is intended to be the coordination model, defined at the architectural level, and representing the *implemented* system behavior; Interaction Diagrams, manually built over the SA components and reflecting the Coordination Requirements previously captured represent the coordination specification (i.e., the *expected* behavior). To guarantee the SA model correctness with respect to the selected requirements, we need to validate the SA LTS by model checking it on the Interaction Diagrams, i.e., we need to validate if the implemented behavior is correct with respect to the expected one.

Step4: From SA model To IWIM Coordination Model

In the last step (as drawn in the right part of Figure 2) we

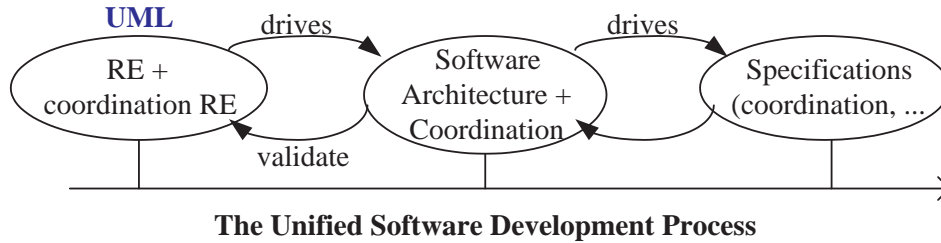


Figure 1. Requirements, Software Architectures and Coordination models

use the SA description to drive the specification of coordination model in the IWIM formalism.

The IWIM [1] model for coordination is described by *processes, ports, channels* and *events*. A *process* is a black box operating unit. It can 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 information exchange; each communicating process owns 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.

SA items are comparable with IWIM items: SA components and IWIM processes are black box units; an SA component is the high level description of an IWIM process; the SA description is higher level since a single SA component can be realized by several IWIM processes. Following these considerations, it is amenable to realize a mapping between the SA description level to the Coordination:

- the SA coordination component becomes a manager process while others become worker process;
- the SA channel (and port) semantics is close to the IWIM model: each SA channel can be mapped in one of the five IWIM channels semantics;
- the IWIM events are comparable with transactions in the LTS model of SA dynamics.

Figure 2 refines Figure 1 and graphically depicts the approach steps. In [10] a full description of the approach can be found with its application to a case study.

3. Consistency Checking between models

Despite the high level of abstraction very often SA are too complex to be managed. A way to tackle system complexity consists of representing the system through several

view points [13, 8, 4]; as a direct consequence, different models are used to represent the different views. In practical contexts, statecharts and scenarios are the most used tools to model the system dynamics.

Although very expressive this approach has two drawbacks with respect to analysis and validation. The first one deals with system specification *incompleteness*: statecharts and scenarios only partially model the system components and interactions. The second is a problem of *view consistency*: several views in fact, are not independent or orthogonal and can erroneously describe different systems. In [11] we proposed an approach to complete statecharts models of the system architecture behavior using interaction scenarios. On the other side we wanted to *validate the obtained model with respect to component-interaction requirements* expressed by the scenarios. To a certain extent this amounts also at validating the two dynamic models.

In this section we will analyze only the validation purpose, with the aim to verify that a model of the system dynamics (statecharts) corresponds to the coordination requirements (expressed by scenarios).

To implement this approach we make the assumption that the SA we want to validate is expressed by an Architectural Description Language who describes the components behavior and the inter-component interactions using statecharts and scenarios respectively. The components statecharts are translated into a Promela specification in the first step; in the second step, the system scenarios, are expressed by Linear time Temporal Logic (LTL) formulae. Finally, the SPIN [17] model checker runs on these specifications to check if the system behavior expressed by the scenarios are well implemented by the model generated by SPIN and based on the Promela [17] specification. This provides a first validation of the architectural model that can be used to perform analysis with respect to safety and liveness properties on the SA dynamics. Moreover, the standard SPIN model check may be run on the statechart model to identify deadlocks, constrains violations, livelocks and some other properties [17].

Figure 3 summarizes the approach.

We now summarize, in an informal way, the three steps the approach is composed of. Technical descriptions can be

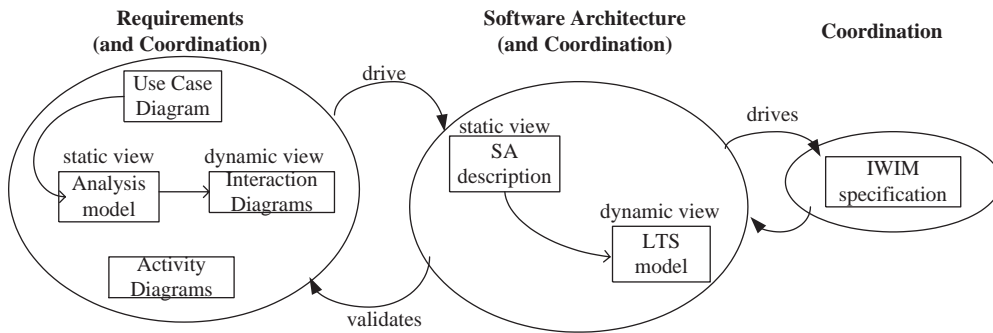


Figure 2. Coordination Requirements and SA: The Approach

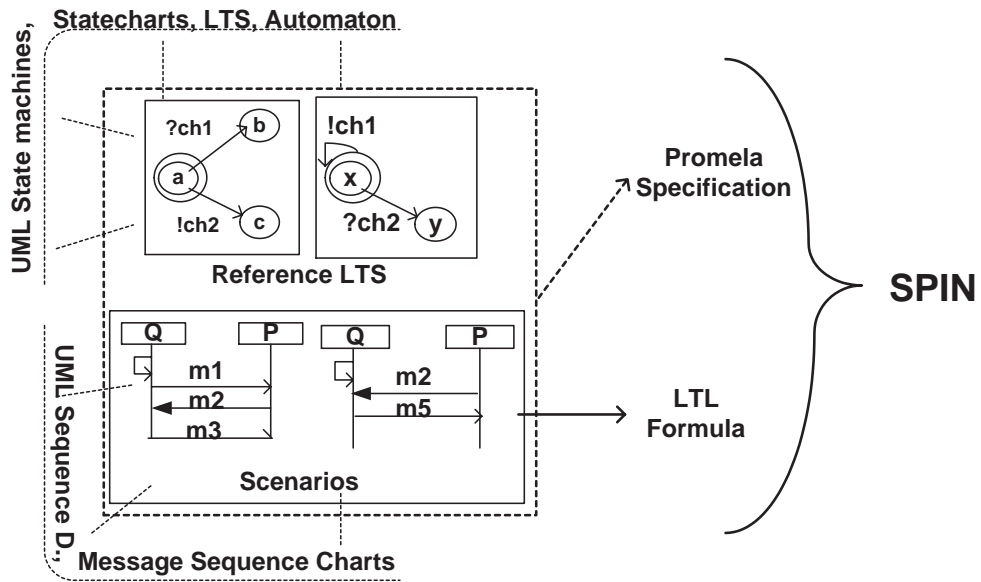


Figure 3. Consistency Checking between models: The Approach

found in [11].

First Step: statecharts expressed in Promela

The statecharts models, describing the components behavior, are translated into a Promela specification through an incremental style which in five steps identifies Promela constants, variables and proctypes. The mapping will structure the Promela specification reflecting the separation among components and connectors. It is important to notice that the Promela specification we write follows the [14, 7] mapping directions but contains variables and data structures needed for the subsequent analysis.

Second Step: scenarios expressed in LTL

Scenarios semantics is translated into LTL formulae; this operation set the order in the events that builds the scenario. Each scenario represents an expected system behavior; our aim is to use SPIN to model-check if the architectural model (expressed in Promela) conforms to the selected scenarios. It is important to note that the LTL formulae are written making reference to the variables introduced in the Promela specification, that store information on the system execution.

Third Step: running SPIN

The last step consist on the check of the system using the model-checker Spin. In the case that an LTL formula (representing an expected system behavior) is not verified on the Promela model (representing the architectural behavior), an architectural inconsistency is found and the erroneous behavior is drawn.

In [11] a detailed description of the approach can be found with its application to a case study.

4. Conclusions and Ongoing Works

To bridge the gap between Requirements and Software Architectures we propose to integrate two different approaches: in the first one, defined in Section 2, the software designer captures the Requirements using an UML modelization and drives the high- and low-level specification description in a way to trace coordination requirements during the software life-cycle. It also recognizes the necessity to be able to validate the architectural model with respect to some requirements. In the second approach, the designer receives the tool to perform the validation: coordination requirements are expressed using scenarios (translated in LTL formulae), the architectural model of dynamics is represented using the Promela formalism and SPIN is run to model-check the conformance of the implemented behavior with respect to the expected one.

At the state of the art, we applied both the approaches to a common case study, the Teleservice and Remote Medical Care System (TRMCS) [3, 9]: this system provides monitoring and assistance to users with specific needs, like disabled or elderly people and a typical service is to send rel-

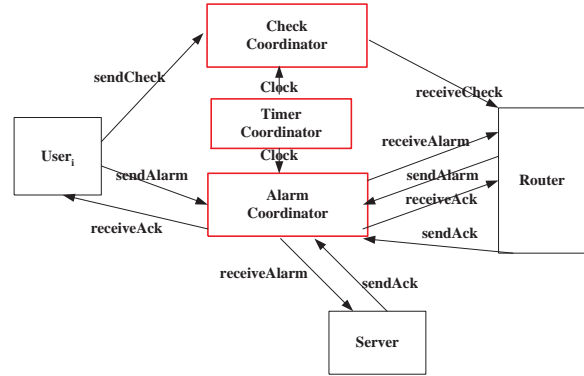


Figure 4. The TRMCS SA topology

evant information to a local phone-center so that the family, medical or technical assistance can be timely notified of critical circumstances.

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* and the *coordination constrains* are the following:

1. An Alarm message sent from User has to be followed by an acknowledgment message;
2. An User can send Alarms and Checks whenever he wants;
3. Checks and Alarms messages, sent by different Users, must be concurrently managed.

Following the approach described in Section 2, in [10] we captured system requirements using Use Case diagrams, we built an analysis model of these Use cases and identified the SA components and connectors (Figure 4). We then described the system behavior using the FSP [6] Process Algebra generating the transition system of the architectural behavior and finally identified some coordination scenarios of interest (Figures 5.a, b, c represent possible dynamic scenarios conforming to the first, second and third described coordination constrains respectively).

Following the approach described in Section 3, we translated the TRMCS SA components statecharts in Promela and the coordination scenarios (like those in Figure 5) in LTL formulae (as shown in [11]).

The work we are now developing is to model-check the Promela model with respect to the selected scenarios: just to report an initial result, model-checking the scenario in Figure 5.b we found an incoherence between the requirements and the SA model. As we said before, the scenario in Figure

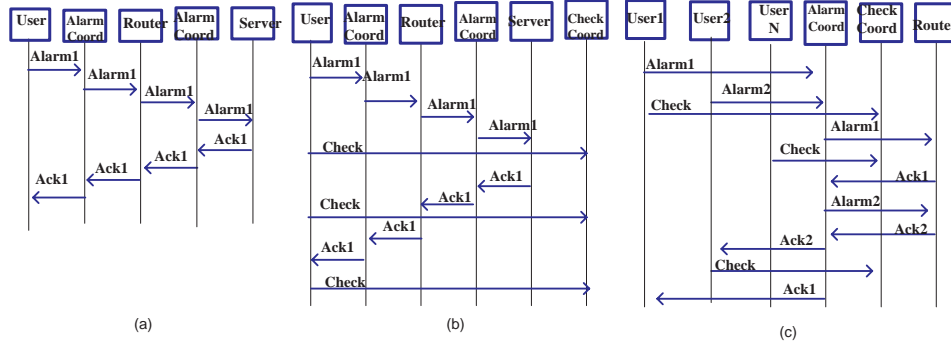


Figure 5. Coordination Requirements at the SA level of abstraction

5.b is a possible modelization of the second coordination requirement in which “An User can send Alarms and Checks whenever he wants”; running the SPIN verification process and analyzing the output trails we was informed that (in the TRMCS architectural model) a second Check msg can be delivered *if and only if* the first Check has been previously forwarded to the Router component. It contradicts the scenarios and the coordination requirement. This result only represents an initial report but, anywhere, it identifies a certain sort of unexpected bottleneck in the SA model and let us increase our expectations on this approach.

As an ongoing work, we are developing a tool to automatically translate scenarios in LTL formulae and statecharts in Promela following the mapping defined in [11]. In this way, the software architect does not need to know Promela or LTL: she draws statecharts and scenarios and the tool automatically generate the inputs for the SPIN model checker.

An idea for future works is to consider enriched statecharts and scenarios to prove that the architectural model correctly behaves with respect to the expected behavior and that quantitative or temporal requirements are met by the specification.

Acknowledgments

The authors would like to acknowledge the Italian M.U.R.S.T. national project SALADIN that partly supported this work and the anonymous reviewers for their constructive comments.

References

- [1] F. Arbab. Coordination of massively concurrent activities. CWI Report CS-R9565 (1995).
- [2] F. Arbab. What Do You Mean, Coordination? *In the March '98 Issue of the Bulletin of the Dutch Assoc. for Theor. Comp. Sc. (NVTI)*, Available at: <<http://www.cwi.nl/farhad/>>.
- [3] S. Balsamo, P. Inverardi, C. Mangano, and F. Russo. Performance Evaluation of a Software Architecture: A Case Study. *In IEEE Proc. IWSSD-9*, pp. 116-125, Japan, 1998.
- [4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series, Addison-Wesley, 1998.
- [5] Coordination'99. *Proc. 3rd Int'l Conf. on Coordination Languages and Models*, LNCS 1594, Springer Verlag, 1999.
- [6] FSP. Finite State Process. On-line at: <<http://www.dse.doc.ic.ac.uk/~jnm/book/ltsa/Appendix-A.html>>.
- [7] S. Gnesi, D. Latella, and M. Massink. Model Checking UML Statecharts Diagrams using JACK. *In Proc. Fourth IEEE International Symposium on High Assurance Systems Engineering*, IEEE Press, 1999.
- [8] C. Hofmeister, R. L. Nord, and D. Soni. *Applied Software Architecture*. Addison Wesley, 1999.
- [9] P. Inverardi and H. Muccini. Coordination models and Software Architectures in a Unified Software Development Process. *In the Coordination 2000 Proceedings*, On-line at: <<http://www.dm.univaq.it/~muccini/Page2.html>>.
- [10] P. Inverardi and H. Muccini. Coordination models and Software Architectures in a Unified Software Development Process. *Internal Report. University of L'Aquila*, On-line at: <<http://www.dm.univaq.it/~muccini/Page2.html>>.
- [11] P. Inverardi, H. Muccini, and P. Pelliccione. Checking consistency between architectural models using SPIN. On-line at: <<http://www.dm.univaq.it/~muccini/Page2.html>>.
- [12] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, Object Technology Series, 1999.
- [13] P. Kruchten. Architectural Blueprints - The “4+1” View Model of Software Architecture. *IEEE Software*, 12(6):42-50, November 1995.
- [14] J. Lilius and I. P. Paltor. vUML: a Tool for Verifying UML Models. *TUCS Technical Report, Number 272 May 1999.*, On-line at: <<http://www.abo.fi/~iporres/vUML/vUML.html>>.
- [15] Rational-Corporation. UML documentation, version 1.3. On-line at: <<http://www.rational.com/uml/index.jhtml>>.
- [16] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, New Jersey, 1996.
- [17] Spin. Home page. On-line at: <<http://cm.bell-labs.com/cm/cs/what/spin/index.html>>.