

Coordination models and Software Architectures in a Unified Software Development Process

Paola Inverardi¹ and Henry Muccini¹

Dipartimento di Matematica
Università dell'Aquila
Via Vetoio, 1 - 67100 L'Aquila, Italy
e-mail{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 integrated in a UML-based software development process . . .

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 [7]. 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 explicitly 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.

In this work we are going to describe an extension to an our previous work [11] proposing a way to profitably and coherently relate these two system development phases. We cast our approach in a UML software development process [13] assuming that UML Diagrams can describe each step in the software development life-cycle [13,15,5,10]. Our approach allows us to use the SA model

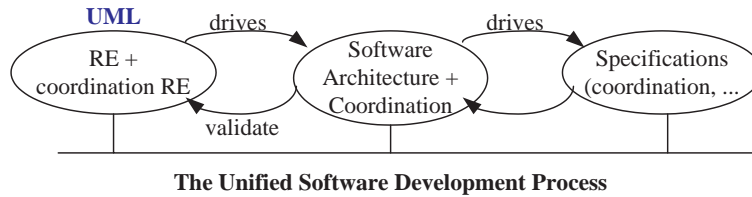


Figure 1. SA description drives Coordination models

to validate coordination requirements and to drive the generation of a correct (with respect to requirements) coordination model.

The paper is organized as follows: section 2 summarizes the generation of an analysis model starting from Use Cases; section 3 introduces software architectures. The approach is analyzed in section 4. Section 5 contains conclusions and future works.

The approach has been applied to the Teleservice and Remote Medical Care System (TRMCS) [4,11] case study, presented in the full version of this paper [12].

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 [13] is Use-Case driven, Architecture-Centric and, Iterative and Incremental. In this section we summarize only the process steps relevant to 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 [9] 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** (see Section 4). To create this model a set of use cases is selected and the analysis classes that participate in his realization are identified. 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, Interaction Diagrams can be used. In particular, Collaboration Diagrams show how the focus control passes from object to object.

3 Software Architecture

Software architectures (SAs) represent the overall system structure by modeling individual components and their interactions. 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.

Thus, they naturally provide the right level of abstraction to express the logical coordination structure of complex distributed software.

Our approach assumes the existence of a formal definition of the Software Architecture encompassing both the static and the dynamic descriptions [8,1,14,6]. We assume that from an architectural description a Labelled Transition System (LTS) 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. See [12] for the definitions.

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*: $p = S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} S_2 \xrightarrow{l_3} \dots \xrightarrow{l_n} S_n$. p is complete if S_0 is the initial state and S_n is a final one.

4 The Approach

Following the Unified Software Development Process [13] UML can be used to graphically depict the functional requirements and the high-level design of the system (as shown in Section 2).

In the following subsections we are going to identify how UML can be used to capture coordination policies at the Requirement level (Subsection 4.1), to understand how this information can be used to *drive* the SA description (Subsection 4.2) and how the SA can be used to *validate* Coordination requirements (Subsection 4.3). In Subsection 4.4 we briefly describe how the software architectural model can be subsequently used to *drive* a coordination model specification of the system and finally we summarize the approach (Figure 2).

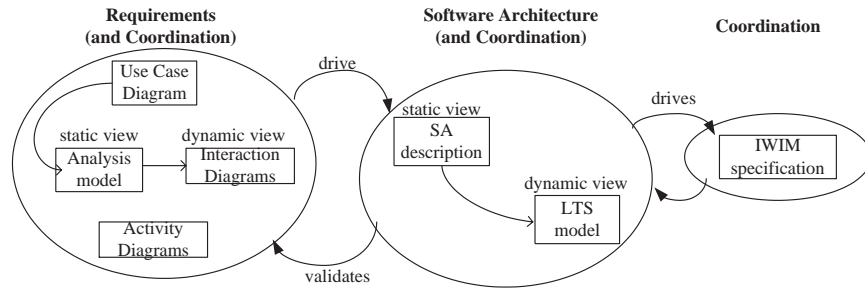


Figure 2. From Requirements to Coordination models

4.1 UML capturing Coordination

UML Interaction Diagram can be extracted on the Analysis model to represent possible execution scenarios [11]; each scenario represents one expected behavior (with respect to the coordination policies) and every scenario implicitly embodies the coordination policies. Interaction Diagrams can give information on the action sequencing but on how many processes are involved in the system, how they cooperate, how they 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. In the next Subsection we will show how the information captured in this first step can be propagated into the software architecture step.

4.2 From UML Diagrams To SA Model

We first need to identify the Components and the Connectors.

The Analysis model generated in section 2 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*; they are used to encapsulate control related to a specific use case.

This model can be used to model the *topological* SA description but needs other information to describe the *dynamic interactions* architectural model: Activity Diagrams can be used for this goal. The underlying idea to generate an SA description from the Analysis model and the Activity Diagrams is the following:

- actors become components;

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

4.3 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). 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 Sequence Diagrams (SDs) 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 are components.

Since we know how analysis objects are mapped to SA components (as briefly discussed in section 4.2) we can validate the SDs looking at objects that are in the two representations intersection.

4.4 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 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 (as defined in Section 4.2) 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.

The formalization of this mapping will be of interest for future works.

4.5 Summarizing the approach

Here the steps followed by our approach are listed:

1. – Use Case Diagrams are captured from functional Requirements;
– Use Cases are realized by the analysis classes;
– interaction diagrams are extracted to describe the analysis model scenarios;
– activity diagrams highlight system activities and synchronization;
2. using Analysis level objects and activity diagrams information, SA components and connectors can be selected and formally described;
3. – an LTS can be automatically generated (from the SA specification) to describe the system dynamics in terms of components interactions;
– interaction diagrams and LTS can be compared for consistency checking;
– an IWIM model construction can be driven by SA description.

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 Assoc. for Theor. Comp. Sc. (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*, Japan (April 1998).
5. Booch, G.: Software Architecture and the UML. Slides available at <<http://www.rational.com/uml/index.jt.html>>.
6. Compare, D., Inverardi, P., Wolf, A. L.: Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming* (33)2 (1999) pp. 101-131.
7. Coordination '99: *Proc. 3rd Int'l Conf. on Coordination Languages and Models*, LNCS 1594, Springer Verlag, April 1999.
8. Darwin, an Architectural Description Language. Web site: <<http://www-dse.doc.ic.ac.uk/research/darwin/darwin.html>>.
9. Eriksson, H.E., Penker, M.: UML Toolkit. John Wiley & Sons (1998).
10. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley, (1999).
11. Inverardi, P., Muccini, H.: A Coordination Process Based on UML and a Software Architectural Description On *Proc. of the 2000 I.C. PDPTA*, session: Coordination Models, Languages, Systems and Applications, June 2000, Las Vegas, USA.
12. Inverardi, P., Muccini, H.: Coordination models and Software Architectures in a Unified Software Development Process On-line at: <<http://univaq.it/~inverard>>.
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. 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.