Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 211 (2008) 147–157
www.elsevier.com/locate/entcs

Towards a Graphical Tool for Refining User to System Requirements

Marco Autili[1] Patrizio Pelliccione[2]

Dipartimento di Informatica, Università dell'Aquila
I-67010 L'Aquila, Italy

Abstract

Informal and abstract user requirement specifications are usually complemented by formal and detailed system requirement specifications. While user requirements provide a high level description of what services the system is expected to provide, system requirements provide a more technical specification of how that services should be provided by the system. One of the relevant problems that arise during the Requirement Engineering process is the result of failing to make a clear transition between different levels of requirements description.
Goal of this paper is to introduce a graphical tool for requirements refinement which guides software architects while moving from user requirements to (architec-tural-level) system requirements. The tool makes use of a previous work that gives a simple but expressive graphical formalism, based on UML2.0 Sequence Diagrams, for specifying temporal properties.

Keywords: Graphical Formalisms; Requirements Definition; Software Architectures.

1 Introduction

Formal methods can have an important role in developing reliable, effective computer systems. Verification techniques have been introduced to understand if a system satisfies certain expected properties. These properties are often informally specified as part of user requirements and tools have been proposed to make specification and analysis rigorous and to help software engineers in their work [2]. Even if much work has been done on this direction, the application of such techniques and tools into industrial world can be still very difficult due to some extra requirements and constraints imposed by industrial needs.

For instance, model checker tools allow for automated checking of system model compliance to given temporal properties. These properties are typically specified as linear-time formulae in suitable temporal logics. However, it is a difficult task to

[1] Email: marco.autili@di.univaq.it
[2] Email: pellicci@di.univaq.it

doi:10.1016/j.entcs.2008.04.037

accurately and correctly express properties in these formalisms. Properties that are simply captured within the context of interest and that are described in intuitive way by natural language may result very hard to specify in temporal logics. In other words, there is a substantial gap between natural language and temporal logic syntax. As a matter of fact, industries are not willing to use the above mentioned techniques and tools and this slows down the transition from "*research theory*" to "*industry practice*".

What is really needed is a semi-formal and easy to learn methodology for specifying such properties. In addition, the methodology should be time reducing, tool supported (automated tool support is fundamental for strongly reducing human effort and costs) and based on graphical notations that are widespread adopted in industrial contexts. For instance, UML [12] (as standard de-facto for software systems modelling) is one of the most attractive notations. In our context, scenario based formalisms (such as UML2.0 Sequence Diagrams) have been advocated as a means of improving requirements engineering but yet few methods or software assistant tools exist to support Sequence Diagrams based *Requirements Engineering* (RE). Since UML2.0 has not yet provided a formal semantics for its diagrams and operators, it is ambiguous and it is very difficult to develop formal techniques based on its notations. Several approaches have been proposed in the last years trying to give semantics to UML2.0 Sequence Diagrams [17], to pose constraints on UML (based on *Object Constraint Language*) or to develop UML-profiles that solve the ambiguity problem in particular contexts.

In a previous work we presented a simple and (sufficiently) powerful formalism for specifying temporal properties in a user-friendly fashion. We proposed a scenario-based visual language that is an extended graphical notation of a subset of UML2.0 Sequence Diagrams. The language is called *Property Sequence Chart* (PSC). PSC can graphically express a useful set of both *liveness* and *safety* properties in terms of messages exchanged among the components forming the system. We also presented an algorithm, called Psc2Ba, to translate PSC into Büchi automata[3] [3]. Even thought PSC has an intuitive and user-friendly graphical notation, it might be still difficult to directly express properties in this language. In fact, during the early stage of the RE process the requirements are usually too abstract and vague.

One relevant problem that arises during the requirement engineering process is the result of failing to make a clear transition between different levels of requirements description. According to the terminology adopted in [16], the term "*user requirements*" is used to mean high-level abstract requirement descriptions and the term "*system requirements*" is used to mean detailed and possibly formal descriptions. Often in practice, stake-holders are able to describe user requirements in informal way without detailed technical knowledge. They are rarely willing to use structured notations or formal ones.

Transiting from user requirements to system requirements is an expensive task,

[3] In our context, a Büchi automata is an operational description of a temporal property formula. It represents all the system behaviors that respects the logic of the specified temporal property.

even if required. In fact, we are speaking about decisions made during this early phase of the software development process, when the system under development is vague also in the mind of the customer. What we need is a speculative and tool supported process that facilitates understanding and structuring requirements. A well recognized instrument by human society for problems understanding is conversation and discussion. Inspired to the human nature we think that a "*conversational*" tool is what we need at this phase. The tool we are proposing is called W_PSC. By means of a set of sentences (based on expertise in requirements formalization and on a set of well-known patterns [6] for specifying temporal properties used in practice) and classified according to temporal properties main keywords, W_PSC forces to make decisions that break the uncertainty and the ambiguity of user requirements.

2 Background

2.1 Our Context

Software Architecture (SA) acts as a *bridge* between the requirements and the implementation code (which has to reflect architectural properties) [7]. An SA specification represents a high-level design model and captures the system structure by identifying architectural components and connectors in order to assess at an early stage what is the best way to ensure that all key requirements are satisfied.

Usually, software architects go through informal user requirements, talk with customers, analyze existent architectural patterns [15] in order to understand which components they need to use, how such components behave and how they have to be connected. The relationship between requirements and architectures has recently received increased attention [18].

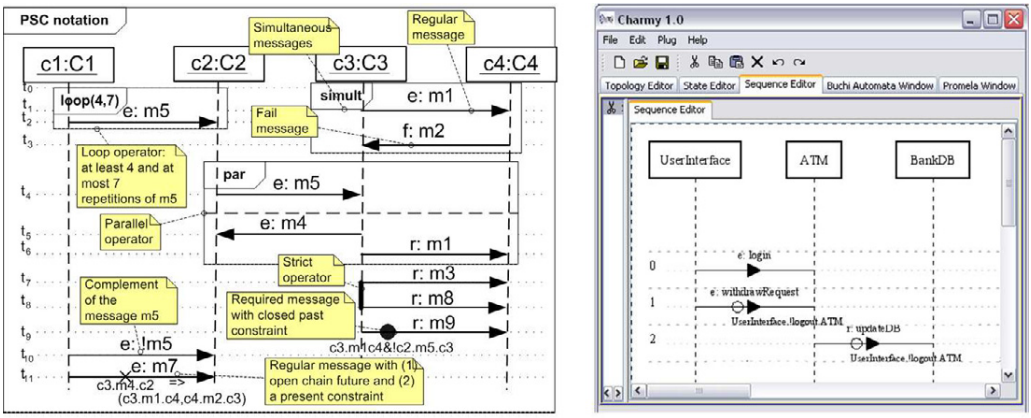
In this context, while user requirements embody some knowledge of the problem domain, system requirements describe properties we expect our system (structured as a given software architecture) satisfies. While user requirements might be informal and ambiguous, system requirements must be well formalized and unambiguous, since they will be used to drive the design and implementation stages and may be used for validating the system model conformance to user requirements.

That is, the transition from informal user requirements to formal (architecture-level) system requirements is unavoidable and usually relies on stake-holders experience. Moreover, this transition spans from the problem space to the solution space where requirements are described in terms of components, connectors and their interactions.

*2.2 Properties Sequence Charts (*PSC*)*

PSC [1,13] is a simple but expressive graphical formalism for specifying temporal properties. Two are the main requirements of PSC, *simplicity* and *expressiveness*. Remaining close to the graphical notation of UML2.0 Sequence Diagrams, the requirement of simplicity is satisfied. The PSC expressiveness is measured with the *property specification patterns* proposed in [6].

PSC describes interactions between a collection of components that can be simultaneously executed and that communicate by message passing. PSC distinguishes among three different types of messages called *arrowMSGs* (see Figure 1.a). *Regular messages*: the labels of such messages are prefixed by "e:". They denote messages that constitute the precondition for a desired (or an undesired) behavior. It is not mandatory for the system to exchange a Regular message; however, if it happens the precondition for the continuations has been verified. *Required messages*: are identified by "r:" prefixed to the labels. It is mandatory for the system to exchange this type of messages. *Fail messages*: the labels are prefixed by "f:".



a) b)

Fig. 1. PSC graphical notation (a) and the PSC tool (b)

They identify messages that should never be exchanged. Fail messages are used to express undesired interactions. We also define *Constraint* operators that impose "restrictions" on the set of messages (called *intraMSGs*) possibly exchanged between the considered message and its predecessor or successor (the predecessor of the first message of a PSC is the startup of the system). Restrictions specify either a chain of *intraMSGs* or a boolean formula (over a set of *intraMSG* labels). *Parallel*, *Loop*, and *Simultaneous* operators are introduced with a UML 2.0 like graphical notation. For the sake of brevity, we omit the full description of PSC features and we entirely refer to [1,13] for it.

2.3 Specification patterns for finite-state verification

Specification patterns [6] are a repository with the intent of collecting patterns that commonly occur in the property specification of concurrent and reactive systems. A specification pattern has a scope that defines the range in which the pattern must hold. By recasting the notion of scope in our context, five basic kinds of scopes are distinguished: **Global** (the entire program execution), **Before** (the execution up to a given message), **After** (the execution After a given message), **Between** (any part of the execution from one given message to another one) and **After-Until** (like between but the designated part of the execution continues even if the second message does not occur).

One way to classify the patterns is based on the kinds of system behaviors they describe. A first classification splits the patterns into two main categories: *Occurrence Patterns* and *Order Patterns*. Occurrence Patterns are further partitioned in *Absence*, *Universality*, *Existence*, and *Bounded Existence*. Order Patterns are further partitioned in *Precedence*, *Response*, *Chain Precedence*, and *Chain Response*. For the sake of readability we do not go through a detailed description of the classification but we refer to [6] for it.

3 The Solution Space

Much effort has been spent in the last years in formalizing requirements and expressing them in some formalism. Scenarios (such as UML2.0 Sequence Diagrams) have been advocated as a means of improving requirements engineering and they have been confirmed as an important design artifact that can be used for a variety of purposes. Scenarios are particularly useful for adding details to an outline requirements description and represent paths of the system behavior representing possible interactions and relationships between participating components.

Several form of scenarios have been developed, each of which provides different types of information at different levels of details. While it could be not difficult to write "high-level" scenarios (e.g., use case scenarios) to document user requirements, more expressive and formal scenario-based notations are needed to document architectural system requirements. While it is useful to keep clear in mind this separation, it is also important *to bound the gap* between these different levels and to create a link among them. An informal and guided decision process should guide developers to move from user to system requirements.

In this section, a methodological approach for generating a formal specification to the user requirements is introduced. We require that the generation of the formal specification, corresponding to user requirements, has a methodological guidance. Consequently a *"conversational"* graphical tool, which permits to automate the entire process, should be implemented. In the following we describe the *decision helper tool* we have in mind. The tool wants to be an attempt to bridge the gap between possibly informal requirement specifications (as found in practice) and formal ones (as needed in formal methods).

The decision helper we are proposing is called W_PSC and it drives software architects in making decisions while writing *Property Sequence Charts* (PSC) introduced in Section 2.2. Within the PSC language, a property is seen as a relation on a set of exchanged system messages, with zero or more constraints. By means of W_PSC all the patterns, briefly described in Section 2.3, can be easily expressed. As already said, each pattern has a *scope* which is the extent of the program execution over which the pattern must hold.

W_PSC should offer a user-friendly wizard helpful while translating a user requirements description into PSC scenarios. In Figure 2 we show the W_PSC framework. The first phase comprises three primary steps: (i) ***Derivation***, (ii) ***Selection*** and (iii) ***Restriction***. By taking into account the requirements description, the first

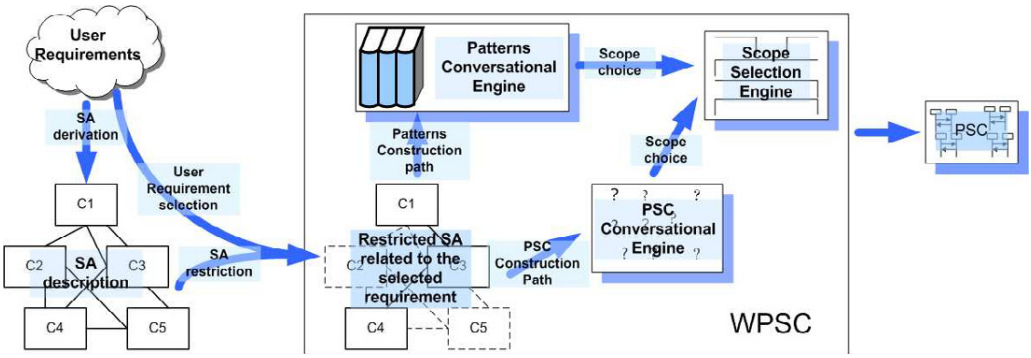


Fig. 2. W_PSC overview

step concerns the (i) derivation of an SA for the system in terms of components, their exchanged messages and connectors. Subsequential, (ii) by selecting one or more user requirements and by identifying an informal property definition within them, the involved components are distinguished. At this point, the SA can be possibly (iii) "restricted" to the subset of distinguished components and related connectors.

Note that, an SA can be derived by extracting real world entities from the requirements description and by mapping these entities into architectural components [16,18]. Moreover, we are assuming that some simple guidelines have been followed to minimize misunderstandings. For instance, we are taking for granted that user requirements, described in the *Requirements Document* (RD), have been written by using language consistently (to distinguish between *mandatory* and *non-mandatory requirements*), by picking out key parts, by avoiding (as far as possible) the use of computer jargon, and possibly by using dedicated structured cards [14]. In this manner the user chooses between W_PSC sentences in the user requirements specification.

When the first phase has been accomplished, the W_PSC user has in mind the SA (and a set of possible exchanged messages) to be given as input to W_PSC by means of a user friendly visual interface. Since such an interface has been already developed for the CHARMY tool [4] and the PSC tool has been already implemented as its plugin, it is possible to input the architecture by the same CHARMY interface.

Now the wizard is ready to propose a set of sentences that drive the W_PSC user through the ***Construction Path*** while deriving the PSC scenario. The Construction Path is composed of several steps and at each step it poses sentences helpful for requirements understanding and for accurately defining them. The Construction Path is twofold: on one hand, the set of sentences are dedicated to PSC specific features (***PSC Construction Path***); on the other hand it is devoted to the library of property specification patterns (***Pattern Construction Path***).

We split the Construction Path in two different paths because, even though the patterns capture a big variety of common property specifications, we let the user to whether going through a particular and specific solution (by exploiting PSC

features) or trying to find an already existent elegant solution (i.e., a pattern). By a series of interactive images, specific text, field and structured dialog boxes, a set of specific sentences are arranged in a *dialog window tree* and a set of *window paths* can be identified from the root to each leaf. Dynamically, according to user decisions, a path is generated and the unique desired PSC scenario is produced.

The wizard engine for supporting the user through the PSC Construction Path is the ***PSC Conversational Engine***. This engine guides the user by means of specific sentences that are brought into focus for PSC features. It might be not easy for a PSC user to choose which type of messages and possible constraints are needed to properly express the informal property he has previously identified from user requirements. Thus, through the window path the user is helped on selecting those messages that are *arrowMSGs* and those ones that are *intraMSGs* (subjected to possible Constraint operators). For each arrow message a type (i.e., Regular, Required, and Fail) must be chosen. For *intraMSGs* there will be a window for constructing allowed boolean formulae (through a graphical syntax-directed editor).

The ***Pattern Conversational Engine*** supports the user through the Pattern Construction Path. This engine asks the W_PSC user by a set of ad-hoc sentences focussed on guiding him through the choice of the appropriate patterns category. By taking into account the "original" pattern descriptions [6] (close to temporal logic jargons), we derive a set of non-technical sentences that can be easily understood. In other word, the sentences are formulated as natural language sentences in such a way that the user is able to quickly identify and select the needed patterns category without any particular knowledge of the patterns themselves. We also propose the creation of a special online help text to answer technical questions for both PSC and Pattern Construction Paths.

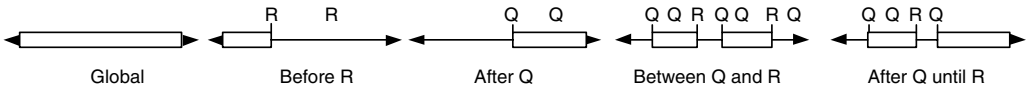


Fig. 3. Pattern scopes

The last phase concerns the ***Scope Selection Engine***. By following the same principles of the above described engines, the Scope Selection Engine exploits interactive images that graphically represent extents of program execution. In Figure 3 we show the graphical representation of the scopes by following the one given in [6]. For W_PSC we propose images based on this representation. Acting with these images the user is driven while selecting the right scope without difficulties.

4 Case Study

In this section we describe how to put into practice the W_PSC approach in a very simple ATM withdrawal case study. Let us suppose that scenarios for withdrawing cash are described as part of the user requirements into the Requirement Document. A high-level SA description (depicted in Figure 4) of an ATM system can be derived. It allows users to: buy a refill card for its mobile phone, check its bank account, and

make a withdraw operation. The system has been designed as the composition of a set of distributed components: a *User Interface*, the *Phone Company (PC)*, the *Bank DB*, and an *ATM* that manages all the interactions between the user and the other entities.

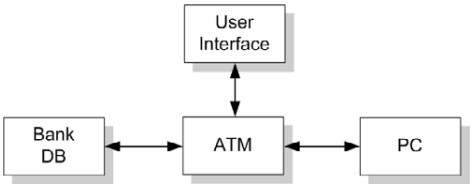


Fig. 4. ATM Application

The informal description of the selected property is: "*The ATM withdrawal shall provide the service for withdrawing cash; there will be a login and logout feature; the ATM will be connected to the bank Data-Base (DB) that will be updated after that a withdraw request has been satisfied*".

The components involved in this property are the *User Interface*, the *Bank DB*, and an *ATM*.

Within such a user requirement description, it is not difficult to capture a desirable system property that states: "*If the withdraw request has been satisfied, the bank DB must be updated; the withdraw request is allowed only after the login request and only until the logout request*".

As already said above, it is a non-trivial task to choose both the needed type of message and the right scope. For instance, the above property might be erroneously expressed as an ordered sequence of regular messages *login*, *withdrawRequest*, *updateDB*, *logout* among the interested components. While such a scenario may be correct for scratching a possible system interaction, it is incorrect for formally specifying our system property. In fact, *updateDB* is mandatory and if it is not exchanged, the system will fail.

That is, by using W_PSC, the PSC formalization for this property can be obtained by performing these subsequential steps:

(i) By tacking into account the first part of the property within the above described property (i.e., *"If the withdraw request has been satisfied,"*), the first choice in which the user is guided is recognizing that the withdrawal request is optional. Thus, between the optional sentences, the user is asked to select the right W_PSC sentence (i.e.: If the message *withdraw_request* is exchanged then ...,).

(ii) Considering the following part of the property (i.e., *"the bank DB must be updated;"*) the involved message *bank_DB* is recognized as mandatory. Thus, the user is asked to choose the right sentence between the mandatory sentences (i.e.: The message *bank_DB* must be exchanged).

(iii) The last part of the property (i.e. *"the withdraw request is allowed only after the login request and only until the logout request"*) it is easily recognized as a scope. The Scope Selection Engine proposes the different choices and the

user is then guided to choose the "**After-Until**" scope (i.e. "*After login*" and "*Until logout*" scope) that embraces the withdraw request and the DB update.

By composing the chosen sentences the property is rewritten as follows:

"After the *login* message has been exchanged and until the message *logout* is not exchanged, if the message *withdraw_request* is exchanged then the message *bank_DB* must be exchanged."

This description can be "compared" with the informal property definition identified within the Requirement Document. Obviously, it will be rarely the same, but it will be simple to understand if the generated property is the wanted one. In other words we are supposing that the user have in mind the property but he needs help in making decisions: this textual representation helps in this sense thanks to the given feedback in terms of textual language.

The automatically generated PSC is showed in Figure 1.b. The resulting PSC is different from a simple sequence of ordered messages. Even though the previous example is a toy example, it is not difficult to grasp the main advantage of having such a conversation.

5 Related Work

Several works have been proposed in the last years attempting to bridge the gap between informal requirement descriptions to formal ones. For lack of space, we discuss only those works closest to our approach.

The approach described in [8,10] is able to translate OCL specifications into natural language by a multilingual syntax-directed editor. Even though foundations and design principles might be inherited, in a contrary manner from our approach, they guide the user to transit in the opposite way. In other words, once formal OCL specifications have been produced, they can be translated into natural language descriptions that can be understood by people who do not know OCL.

In [19] authors exploit a software tool that allows system engineers to write detailed use case descriptions using structured templates. The specification is guided by use case style guidelines, temporal semantics and an extensive dictionary of naval domain nouns. Once the use case description phase has been accomplished, system engineers derive use case specifications and, after parametrization, corresponding scenarios are automatically generated. Differently from our proposal, this approach is domain specific and it is dedicated to software engineers with specific domain expertise that are able to directly describe and subsequently specify parameterized use case diagrams.

In [9] the authors present STAIRS, a formal approach to the incremental specification of UML 2.0 interaction for capturing requirements. By referring to Section 2.2, STAIRS is primarily related to our work about PSC [1,13]. Like us, they can deal with mandatory, forbidden and optional scenarios and have the notion of refinements. They use the trace semantics of UML 2.0 but, as we pointed out in [1], UML 2.0 has yet again not provided a formal description of that semantics. Differently from them we provide PSC with a precise semantics via translation,

by means of an algorithm implemented as a plugin of our CHARMY tool [4], into Büchi automata. After translation PSC diagrams can be directly used by CHARMY for model-checking software architectures. Moreover, as it is showed in [1,13], the expressiveness of PSC has been validated with respect to well known *property specification patterns* [6].

In [11] the authors propose an approach called play-in/play-out for specifying and executing behavioral requirements based on LSC [5]. Play-in makes capturing requirements quite intuitive, based on interactions with a prototype of the application GUI. Even if the interaction with the GUI seems fashioning, the user has to choose if the operation performed is mandatory or possible. This is done by selecting in a checkbox list. Thus, as the authors themselves point out, to use complex and sophisticated features of LSC requires being familiar with the LSC language. This aspect represents the more difficult part. In fact the user can friendly interact with the GUI application but is not helped in making decisions for selecting the right checkboxes in the list. On the contrary W_PSC aims exactly in guiding the user in making choices while exploiting PSC features. Finally the play-in/play-out approach requires specifying a GUI for the application inside the play engine tool. This appears a limitation that can reduce the applicability of the approach to complicated and sophisticated GUI applications.

6 Conclusion and future work

Inspired to the human nature, in this paper we propose a "*conversational*" tool called *Wizard Property Sequence Charts*. By means of posed sentences W_PSC forces to make decisions that break the uncertainty and the ambiguity of user requirements. Sentences are derived from expertise in requirements formalization and from the commonly used *property specification patterns* [6].

W_PSC strives to guide developers while moving from user requirements to (architecture-level) system requirements. The PSC input is an SA and a set of messages possibly exchanged among the components forming the system. At least a non-fine-grained SA can be derived by extracting real world entities from the requirements description. Then, these entities are mapped into architectural components [16,18]. Later the SA can be obviously better refined and all steps we described can be reiterated. As system requirements specification language the tool makes use of PSC that is a simple but expressive graphical formalism, based on UML2.0 Sequence Diagrams, for specifying temporal properties.

On the future work side we plan to conclude the ongoing development of the tool and to actively use it in real industrial contexts in order to empirically evaluate impact and effort needed to use it.

Acknowledgement

This work is partially supported by the PLASTIC project: Providing Lightweight and Adaptable Service Technology for pervasive Information and Communication.

Sixth Framework Programme. http://www.ist-plastic.org.

References

[1] M. Autili, P. Inverardi, and P. Pelliccione. A scenario based notation for specifying temporal properties. In *5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06).*, Shanghai, China, May 27, 2006.

[2] M. Bernardo and P. Inverardi. *Formal Methods for Software Architectures, Tutorial book on Software Architectures and Formal Methods.* SFM-03:SA Lectures, LNCS 2804, 2003.

[3] J. Buchi. On a decision method in restricted second order arithmetic. In *International Congress on Logic, Method and Philosophical Sciences*, 1960.

[4] Charmy Project. Charmy web site. http://www.di.univaq.it/charmy, February 2004.

[5] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[7] D. Garlan. Software Architecture. In *Encyclopedia of Software Engineering, John Wiley & Sons*, 2001.

[8] R. Hähnle, K. Johannisson, and A. Rantac. An authoring tool for informal and formal requirements specifications. In *FASE '02*, pages 233–248, London, UK, 2002. Springer-Verlag.

[9] Ø. Haugen and K. Stølen. STAIRS - steps to analyze interactions with refinement semantics. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 388–402. Springer, 2003.

[10] K. Johannisson. *Formal and Informal Software Specifications.* PhD thesis, C. Technology and Göteborg Univ., SE-412 96 Göteborg, Sweden, 2005.

[11] R. Marelly, D. Harel, and H. Kugler. Specifying and executing requirements: the play-in/play-out approach. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 84–85, New York, NY, USA, 2002. ACM Press.

[12] Object Management Group (OMG). Unified Modeling Language (UML): Superstructure version 2.0, final adopted specification (02/08/2003).

[13] PSC home page: http://www.di.univaq.it/psc2ba, 2005.

[14] S. Robertson and J. Robertson. *Mastering the Requirements Process*, chapter 6. Harlow: Addison-Wesley, 1999.

[15] M. Shaw and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *COMPSAC97, 21st Int. Computer Software and Applications Conference*, 1997.

[16] I. Sommerville. *Software engineering (7th ed.).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[17] H. Störrle. Semantics of Interactions in UML 2.0. In *VLFM'03 Intl. Ws. Visual Languages and Formal Methods, at HCC'03, Auckland, NZ*, 2003.

[18] A. van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures, LNCS 2804*, 2003.

[19] X. Zhu, N. Maiden, and P. Pavan. Scenarios: Bringing requirements and architectures together. In *2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, 2003.