Proceeding of the

First International Workshop From Software Requirements to Architectures - STRAW'01

May 14, 2001 **Toronto, Canada**

Held In Conjuction with ICSE2001 W9

23rd International Conference on Software Engineering

Table of Contents

Foreword by the Workshop Co-Chairs

Requirements Engineering and Software Architecture have become established areas of research, education and practice within the software engineering community. Requirements Engineering is concerned with identifying the purpose of a software system, and the contexts in which it will be used. Software architecture is concerned with the structure of software systems from large grained software components, including their properties, interactions, and patterns of combination.
Significant advances have been made on both fronts. We have seen the development of techniques for eliciting and analysing stakeholders' goals, modelling scenarios that characterise different contexts of use, the use of social techniques for studying organisations and work settings, and the use of formal methods for analysing safety and security requirements [1,2]. Using an architecture based approach, applications have been built which exhibit remarkable flexibility, demonstrate significant use of off-the-shelf components, leverage experience from related applications in the same problem domain, and are analysable earlier in their development than ever before [3].

Despite these advances, we still need frameworks, techniques and tools to support the systematic achievement of architectural objectives in the context of complex stakeholders' relationships. For example, although most agree that requirements that are free of architectural influence tend to be very difficult to elicit and analyse, little effort has been devoted to date to techniques for deriving architectural descriptions in concert with the requirements specifications. It also remains very difficult to show that a given software architecture satisfies a set of functional and non-functional requirements. This is somewhat surprising, as software architecture has long been recognised to have a profound impact on the achievement of non-functional goals ("ilities") such as availability, reliability, maintainability, safety, confidentiality, evolvability, and so forth. Therefore greater effort should be devoted to bridging the gap between Requirements Engineering research and Software Architecture research.

The goal of the workshop is to bring together professionals from academia and industry to exchange ideas and experiences to improve our understanding of the relationship between requirements engineering and software architecture. Topics of interest include:

- Requirements and Architecture modelling;
- Deriving architectural description in concert with requirements specifications;
- Tracing architectural decisions to requirements;
- Systematic derivation of parameter settings from requirements;
- Dealing with requirements and architectural evolution;
- Formal foundations and analyses;
- Object-Oriented Requirements to Object Oriented Architectures;
- Agent-Oriented Requirements to Architectures;
- Education and Training: skills and traits for good requirements engineers and software architects;
- Case studies and empirical studies;
- Tools/Environments for Requirements Engineers and Software Architects.

A maximum of 30 participants attend the workshop, partially selected on the basis of the submitted material. Papers were reviewed by a programme committee in terms of their relevance to the aims of the workshop and technical content. The best papers of the workshop are to be invited to submit extended versions for a Special Issue of the Requirements Engineering Journal to be published in the Fall of 2001.

The workshop features two invited speakers, Richard Taylor and John Mylopoulos, who present views on the interplay between software architecture and requirements engineering. The aim is to set the scene, indicating where we are, where we should be, where the hard problems are, and where the strong leverage points are.

The workshop is an interactive forum. Accepted papers were made available electronically to all workshop participants before the workshop, so that presentations can be kept short. The presentation of selected papers is restricted to allow time for intensive discussions. At the end of the workshop there will be a general discussion, including a brainstorming session about areas or topics of research that the participants perceived as important. #

A summary of the workshop discussions is to be published as a technical report and made electronically available in the organizers' web sites. It aims to highlight outstanding issues that should form a part of the forthcoming research agenda.

For further details, please visit the *First International Workshop From Software Requirements to Architectures* (**STRAW´01**) website: http://www.cin.ufpe.br/~straw01

We extend our sincerest thanks to all members of the Programme Committee for their efforts. Moreover, we hope you enjoy and benefit from the STRAW´01 programme.

Jaelson Castro & Jeff Kramer

REFERENCES

[1] van Lamsweerde. A, "Requirements Engineering in the Year 00: A Research Perspective", *Proceedings of the Twenty Second International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[2] Nuseibeh, B. and Easterbrook, S., "Requirements Engineering: A Roadmap", *in The Future of Software Engineering (ed. Anthony Finkelstein), Twenty Second International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 2000.*

[3] Garlan, D, "Software Architecture: A Roadmap", *in The Future of Software Engineering (ed. Anthony Finkelstein), Twenty Second International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 2000.*

Programme Committee

Daniel Berry, *University of Waterloo, Canada*

Jaelson Castro (Co-Chair), *Universidade Federal de Pernambuco, Brazil*

Martin Feather, *Jet Propulsion Laboratory, NASA, USA*

Anthony Finkelstein, *University College, UK*

David Garlan, *Carnegie Mellon University, USA*

Carlo Ghezzi, *Politecnico di Milano, Italy*

Jeff Kramer (Co-Chair), *Imperial College, UK*

Axel van Lamsweerde, *Universite Catholique de Louvain, Belgium*

Jeff Magee, *Imperial College, UK*

Bashar Nuseibeh, *Open University, UK*

Dewayne Perry, *University of Texas at Austin, USA*

Manuel Kolp, *University of Toronto, Canada*

Session 1

Keynote Address 1

Speaker

John Mylopoulus

University of Toronto, Canada

Software Architectures as Social Structures

Software Architectures as Social Structures

John Mylopoulos
University of Toronto, Canada
jm@cs.toronto.edu

Abstract

The explosive growth of the internet -- and a host of relevant technologies, such as the Web, eCommerce, peer-to-peer computing, Application Service Providers and more -- is posing new challenges for Software Engineering research and practice. These application areas call for software architectures that are totally open and evolve over time to meet the needs of their users.

We adopt the i* framework, originally intended as a modeling language for early requirements, to propose a social perspective on software architectures.According to this perspective, software is viewed as an evolving collection of actors (agents, positions or roles) who have associated goals and depend on other actors for the fulfillment of these goals. These dependencies can be set up at design time or run time. The presentation motivates our approach and sketches a process for generating a software architecture from a requirements specification (also modelled in i*).

The presentation is based on on-going research with Jaelson Castro, Paolo Giorgini and Manuel Kolp.

Session 2

Keynote Address 2

Speaker

Richard Taylor

University of California, Irvinie, USA

*"Requirements engineering is *SO* twentieth century..."*

Requirements engineering is *SO* twentieth century...

Richard Taylor
University of California, Irvinie, USA

Abstract

Ever since Watergate in the 70's the dictum "Follow the money" has led investigators to interesting places. If we "follow the money" in the software development profession, does it lead us to careful requirements engineering? Or does it rather lead us to discussions of market share, time-to-market, "owning a market", and so on? And when people set out to solve a problem or create a product, do they do that best by thinking carefully and abstractly about the goals, proceeding carefully until all the stakeholders are known, the constraints described, and the risks identified? Nuseibeh's "Twin Peaks" model starts us down an attractive road, interweaving requirements and architectures. I will argue a bit stronger position, that economic factors drive us further down the road, where domain expertise, building block components, and select architectural styles provide the leadership in system development ... and specification.

A Social Organization Perspective on Software Architectures

Manuel Kolp
Dept. of Computer Science
University of Toronto
10 King's College Road
Toronto M5S3G4, Canada
mkolp@cs.toronto.edu

Jaelson Castro
Centro de Informática
Universidade Federal de Pernambuco
Av. Prof. Luiz Freire s/n
Recife PE, Brazil 50732-970
jbc@cin.ufpe.br

John Mylopoulos
Dept. of Computer Science
University of Toronto
10 King's College Road
Toronto M5S3G4, Canada
jm@cs.toronto.edu

Abstract

This paper proposes a set of concepts for describing a software architecture as a social organization. This social structure consists of actors who have goals to fulfil and social dependencies describing their obligations. The framework is an adaptation of i [17] proposed as a modeling language for early requirements. Based on this framework, the paper advocates architectural styles for software which adopt concepts from organization theory and strategic alliances literature. The styles are modeled in i* and formalized in terms of Telos metaconcepts. Each proposed style is evaluated with respect to a set of software quality attributes, such as predictability, adaptability and openness. The use of these styles is illustrated and contrasted with two examples of software architectures reported in the literature.*

1. Introduction

We are interested in narrowing the semantic gap between a software architecture and the requirements model from which it was derived. One way to achieve this is to adopt the same concepts for describing requirements and software architectures. This paper reports on an experiment to use concepts from *i**, a modeling framework for early requirements, to model software architectures.

i* offers concepts such as actor, goal, and social dependency intended to model social structures involving social actors, their goals and social inter-dependencies. To adopt this framework for software architectures, we first propose a set of architectural styles inspired by organizational theory and strategic alliance literature, and formalize these as Telos [9] metaconcepts. To guide the selection process among the styles, we evaluate them with respect to a number of software qualities. Finally, we illustrate their use by applying them to two examples of software architectures reported in the literature.

This research is being conducted in the context of the Tropos project [1], which is developing a requirements-driven methodology for software systems.

Section 2 presents our organization-inspired architectural styles described in terms of the strategic dependency model from *i** and specified in Telos. Section 3 introduces a set of desirable software quality attributes for comparing them. Section 4 overviews a mobile robot and an e-business examples while Section 5 sketches the Tropos project within which this research has been conducted. Finally, Section 6 summarizes the contributions of the paper and points to further research.

2. Organizational Styles

Organizational theory (such as [7, 10]) and strategic alliances (e.g., [5, 16]) study alternatives for (business) organizations. These alternatives are used to model the coordination of business stakeholders -- individuals, physical or social systems -- to achieve common goals. Using them, we view a software system as a social organization of coordinated autonomous components (or agents) that interact in order to achieve specific, possibly common goals. We adopt (some of) the styles defined in organizational theory and strategic alliances to design the architecture of the system, model them with *i**, and specify them in Telos [9].

In *i**, a strategic dependency model is a graph, in which each node represents an actor, and each link between two actors indicates that one actor depends on another for something in order that the former may attain some goal. We call the depending actor the depender and the actor who is depended upon the dependee. The object around which the dependency centers is called the dependum. By depending on another actor for a dependum, an actor is able to achieve goals that it is otherwise unable to achieve, or not as easily or as well. At

the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals.

The model distinguishes among four types of dependencies -- goal-, task-, resource-, and softgoal-dependency -- based on the type of freedom that is allowed in the relationship between depender and dependee. Softgoals are distinguished from goals because they do not have a formal definition, and are amenable to a different (more qualitative) kind of analysis [2].

For instance, in the structure-in-5 style (Figure 1), the coordination, middle agency and support actors depend on the apex for strategic management purposes. Since the goal *Strategic Management* is not well-defined, it is represented as a softgoal (cloudy shape). The middle agency actor depends on both the coordination and support actors respectively through goal dependencies *Control* and *Logistics* represented as oval-shaped icons. The operational core actor is related to the coordination and support actors respectively through the *Standardize* task dependency and the *Non-operational service* resource dependency.

In the sequel we briefly discuss ten common organizational styles.

The **structure-in-5** (Figure 1) style consists of the typical strategic and logistic components generally found in many organizations. At the base level one finds the operational core where the basic tasks and operations -- the input, processing, output and direct support procedures associated with running the system -- are carried out. At the top of the organization lies the apex composed of strategic executive actors.

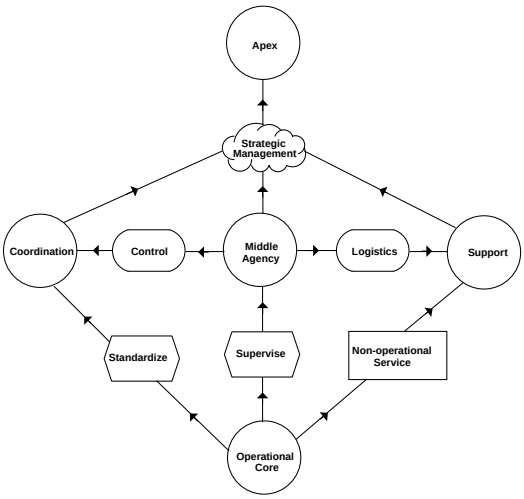


Figure 1. Structure-in-5.

Below it sit the control/standardization, management components and logistics, respectively coordination, middle agency and support. The coordination component carries out the tasks of standardizing the behavior of other components, in addition to applying analytical procedures to help the system adapt to its environment. Actors joining the apex to the operational core make up the middle agency. The support component assists the operational core for non-operational services that are outside the basic flow of operational tasks and procedures.

Figure 2 specifies the structure-in-5 style in Telos [9]. Telos is a language intended for modeling requirements, design, implementation and design decisions for software systems. It provides features to describe metaconcepts that can be used to represent the knowledge relevant to a variety of worlds – subject, usage, system, development worlds - related to a software system. Our styles are formulated as Telos metaconcepts, primarily based on the aggregation semantics for Telos presented in [8].

The structure-in-5 style is then a metaclass - *StructureIn5MetaClass* - aggregation of five (*part*) metaclasses: *ApexMetaClass*, *CoordinationMetaClass*, *MiddleAgencyMetaClass*, *SupportMetaClass* and *OperationalCoreMetaClass*, one for each actor composing the structure-in 5 style depicted in Figure 1. Each of these five components exclusively belongs (*exclusivePart*) to the composite (*StructureIn5MetaClass*) and their existence depend (*dependentPart*) on the existence of the composite. A structure-in-5 specific to an application domain will be defined as a Telos class, instance of *StructureIn5MetaClass* (See Section 4). Similarly each structure-in-5 component specific to a particular application domain will be defined as a class, instance of one of the five *StructureIn5Metaclass* components.

```
TELL CLASS StructureIn5MetaClass
  IN Class WITH /*Class is here used as a MetaMetaClass*/
        attribute
          name: String
        part, exclusivePart, dependentPart
          ApexMetaClass: Class
          CoordinationMetaClass: Class
          MiddleAgencyMetaClass: Class
          SupportMetaClass: Class
          OperationalCoreMetaClass: Class
  END StructureIn5MetaClass
```

Figure 2. Structure-in-5 in Telos.

Figure 3 formulates in Telos one of these five structure-in-5 components: the coordination actor. Dependencies are described following Telos specifications for *i** models [17]. The coordination actor is a metaclass, *CoordinationMetaclass*. According to Figure 1, the coordination actor is the dependee of a task dependency *StandardizeTask* and a goal dependency *ControlGoal*, and the depender of a softgoal dependency *StrategicManagementSoftGoal*.

```
TELL CLASS CoordinationMetaclass
  IN Class WITH /*Class is here used as a MetaMetaClass*/
      attribute
          name: String
      taskDepended
          s:StandardizeTask
                WITH depender
                        OperationalCoreMetaClass: Class
                END
      goalDepended
          c:ControlGoal
                WITH depender
                        MiddleAgencyMetaClass: Class
                END
      softgoalDepender
          s:StrategicManagementSoftGoal
                WITH dependee
                        ApexMetaClass: Class
                END
END CoordinationMetaclass
```

Figure 3. Structure-in-5 coordination actor in Telos.

The **flat structure** has no fixed structure and no control of one actor over another is assumed. The main advantage of this architecture is that it supports autonomy, distribution and continuous evolution of an actor architecture. However, the key drawback is that it requires an increased amount of reasoning and communication by each participating actor.

The **pyramid** style is the well-known hierarchical authority structure exercised within organizational boundaries. Actors at the lower levels depend on actors of the higher levels. The crucial mechanism is direct supervision from the apex. Managers and supervisors are then only intermediate actors routing strategic decisions and authority from the apex to the operating level. They can coordinate behaviors or take decisions by their own but only at a local level. This style can be applied when deploying simple distributed systems.

Moreover, this style encourages dynamicity since coordination and decision mechanisms are direct, not complex and immediately identifiable. Evolvability and modifiability can thus be implemented in terms of this style at low costs. However, it is not suitable for huge distributed systems like multi-agent systems requiring many kinds of agents. Even tough, it can be used by these systems to manage and resolve crisis situations. For instance, a complex multi-agent system faced with a non-authorized intrusion from external and non trustable agents could dynamically, for a short or long time, decide to migrate itself into a pyramid organization to be able to resolve the security problem in a more efficient way.

The **joint venture** style (Figure 4) involves agreement between two or more principal partners to obtain the benefits of larger scale, partial investment and lower maintenance costs. Through the delegation of authority to a specific joint management actor that coordinates tasks and operations and manages sharing of knowledge and resources they pursue joint objectives and common purpose. Each principal partner can manage and control itself on a local dimension and interact directly with other principal partners to exchange, provide and receive services, data and knowledge. However, the strategic operation and coordination of such a system and its partner actors on a global dimension are only ensured by the joint management actor. Outside the joint venture, secondary partners supply services or support tasks for the organization core.

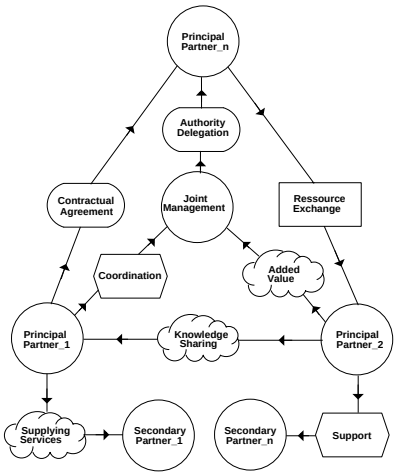


Figure 4. Joint Venture.

The **takeover** style involves the total delegation of authority and management from two or more partners to a single collective *takeover* actor. It is similar in many ways to the joint venture style. The major and crucial difference is that while in a joint venture identities and autonomies of the separate units are preserved, the takeover absorbs these critical units in the sense that no direct relationships, dependencies or communications are tolerated except those involving the takeover.

The **arm's-length** style implies agreements between independent and competitive but partner actors. Partners keep their autonomy and independence but act and put their resources and knowledge together to accomplish precise common goals. No authority is delegated or lost from a collaborator to another.

The **bidding** style (Figure 5) involves competitivity mechanisms and actors behave as if they were taking part in an auction. The auctioneer actor runs the show, advertises the auction issued by the auction issuer, receives bids from bidder actors and ensure communication and feedback with the auction issuer.

The auctioneer might be a system actor that merely organizes and operates the auction and its mechanisms. It can also be one of the bidders (for example selling an item which all other bidders are interested in buying). The auction issuer is responsible for issuing the bidding.

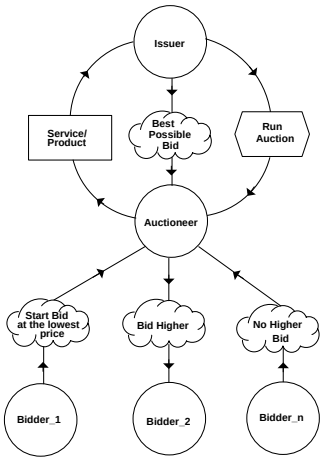


Figure 5. Bidding.

The **hierarchical contracting** style (Figure 6) identifies coordinating mechanisms that combine arm's-length agreement features with aspects associated with pyramidal authority. Coordination mechanisms developed to manage arm's-length (independent) characteristics involve a variety of negotiators, mediators and observers at different levels handling conditional clauses to monitor and manage possible contingencies, negotiate and resolve conflicts and finally deliberate and take decisions. Hierarchical relationships, from the executive apex to the arm's-length contractors (top to bottom) restrict autonomy and underlie a cooperative venture between the contracting parties. Such dual and admittedly complex contracting arrangements can be used to manage conditions of complexity and uncertainty deployed in high-cost-high-gain (high-risk) applications.

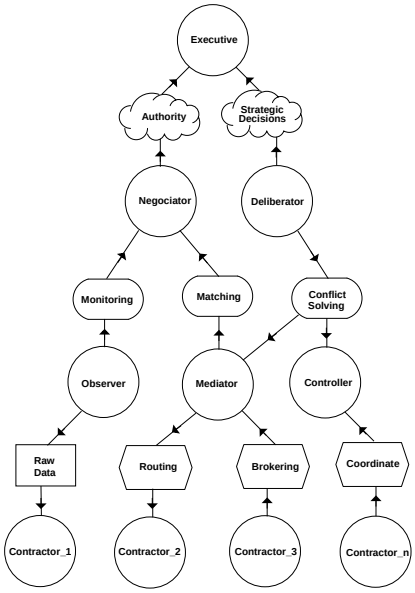


Figure 6. Hierarchical Contracting.

The **vertical integration** style merges, backward or forward, one or more system actors engaged in related tasks but at different stages of a production process. A merger synchronizes and controls interactions between each of the participants that can be considered intermediate workshops. Vertical integrations take place between exchange partners, actors symbiotically related.

The **co-optation** style (Figure 7) involves the incorporation of representatives of external systems into the decision-making or advisory structure and behavior of an initiating organization. By co-opting representatives of external systems, organizations are, in effect, trading confidentiality and authority for resource, knowledge assets and support. The initiating system, and its local contractors, has to come to terms with what is doing on its behalf; and each co-optated actor has to reconcile and adjust his own views with the policy of the system he has to communicate.

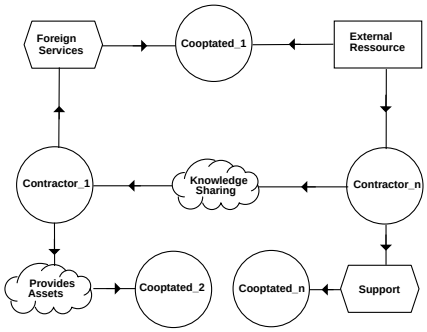


Figure 7. Cooptation.

3. Evaluating Architecture

The organizational styles defined in Section 2 can be evaluated and compared using the following software quality attributes identified for architectures involving coordinated autonomous components (e.g., Web, internet, agent or peer-to-peer software systems) :

1 - Predictability [15]**.** Autonomous components like agents have a high degree of autonomy in the way that they undertake action and communication in their domains. It can be then difficult to predict individual characteristics as part of determining the behavior of a distributed and open system at large.

2 - Security. Autonomous components are often able to identify their own data sources and they may undertake additional actions based on these sources [15]. Protocols and strategies for verifying authenticity for these data sources by individual components are an important concern in the evaluation of overall system quality since, in addition to possibly misleading information acquired by components, there is the danger of hostile external entities

spoofing the system to acquire information accorded to trusted domain components.

3 - Adaptability. Components may be required to adapt to modifications in their environment. They may include changes to the component's communication protocol or possibly the dynamic introduction of a new kind of component previously unknown or the manipulations of existing components.

- Coordinability. Autonomous components are not particularly useful unless they are able to coordinate with other components. This can be realized in two ways:

4 - Cooperativity. They must be able to coordinate with other entities to achieve a common purpose.

5 - Competitivity. The success of one component implies the failure of others.

6 - Availability. Components that offer services to other components must implicitly or explicitly guard against the interruption of offered services. Availability must actually be considered a sub-attribute of security [2]. Nevertheless, we deal with it as a top-level software quality attribute due to its increasing importance in multi-agent system design.

7 - Integrity. A failure of one component does not necessarily imply a failure of the whole system. The system then needs to check the completeness and the accuracy of data, information and knowledge transactions and flows. To prevent system failure, different components can have similar or replicated capabilities and refer to more than one component for a specific behavior.

8 - Modularity [14] increases efficiency of task execution, reduces communication overhead and usually enables high flexibility. On the other hand, it implies constraints on inter-module communication.

9 - Aggregability. Some components are parts of other components. They surrender to the control of the composite entity. This control results in efficient tasks execution and low communication overhead, however prevents the system to benefit from flexibility.

	1	2	3	4	5	6	7	8	9
Flat	--	--	-			+	+	++	-
Struct-5	+	+		+	-	+	++	++	++
Pyramid	++	++	+	++	-	+	--	-	
Joint-Vent	+	+	++	+	-	++		+	++
Bid	--	--	++	-	++	-	--	++	
Takeover	++	++	-	++	--	+		+	+
Arm's-Lgth	-	--	+	-	++	--	++	+	
Hierch Ctr			+	+	+	+		+	+
Vert Integr	+	+	-	+	-	+	--	--	--
Coopt	-	-	++	++	+	--	-	--	

Table 1. Correlation catalogue.

Table 1 summarizes the correlation catalogue for the organizational patterns and top-level quality attributes we have considered. Following notations used by the NFR (non functional requirements) framework [2], +, ++, -, --, respectively model partial/positive, sufficient/positive, partial/negative and sufficient/negative contributions.

4. Examples

To motivate our organizational styles, we consider two application domains where distributed and open architectures (e.g., Web, internet, agent or peer-to-peer software systems) are becoming increasingly important: mobile robots and e-business systems.

We first consider the mobile robot example presented in [13]. That case study describes notably the layered architecture (Figure 8) implemented in the Terregator and Neptune robots and used more recently to design the architecture of the Xavier office delivery robot [11].

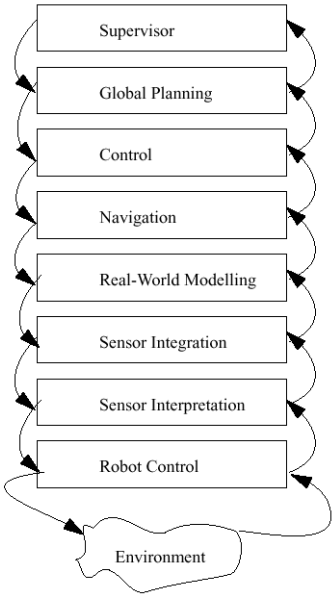


Figure 8. Classical mobile robot layered architecture.

According to [13] at the lowest level, reside the robot control routines (motors, joints,...). Levels 2 and 3 deal with the input from the real world. They perform sensor interpretation (the analysis of the data from one sensor) and sensor integration (the combined analysis of different sensor inputs). Level 4 is concerned with maintaining the robot's model of the world. Level 5 manages the navigation of the robot. The next two levels, 6 and 7, schedule and plan the robot's actions. Dealing with problems and replanning is also part of the level-7 responsibilities. The top level provides the user interface and overall supervisory functions.

The following software quality attributes are relevant for the robot's architecture [13]: *Cooperativity*, *Predictability*, *Adaptability*, *Integrity*. Take for instance, consider *Cooperativity* and *Predictability*.

Cooperativity: the robot has to coordinate the actions it undertakes to achieve its designated objective with the reactions forced on it by the environment (e.g., avoid an obstacle). The idealized layered architecture (Figure 8) implemented on some mobile robots does not really fit the actual data and control-flow patterns [13]. The layered architecture style suggests that services and requests are passed between adjacent layers. However, data and information exchange is actually not always straightforward. Commands and transactions may often need to skip intermediate layers to establish direct communication. A structure-in-5 proposes a more distributed architecture allowing more direct interactions between component.

Another recognized problem is that the layers do not separate the data hierarchy (sensor control, interpreted results, world model) from the control hierarchy (motor control, navigation, scheduling, planning and user-level control). Again the structure-in-5 could better differentiate the data hierarchy - implemented by the operational core, and support components - from the control structure – implemented by the operational core, middle agency and strategic apex as will be described in Figure 9.

Adaptability: application development for mobile robots frequently requires customization, experimentation and dynamic reconfiguration. Moreover, changes in tasks may require regular modification. In the layered architecture, the interdependencies between layers prevent the addition of new components or deletion of existing ones. The structure-in-5 style separates independently each typical component of an organizational structure but a joint venture isolating components and allowing autonomous and dynamic manipulation should be a better candidate. Partner components, except the joint manager, can be added or deleted in a more flexible way.

Figure 9 depicts a mobile robot architecture following the structure-in-5 style from Figure 1. The *control routines* component is the *operational core* managing the robot motors, joints, etc. *Planning/Scheduling* is the *coordination* component scheduling and planning the robot's actions. The *real world interpreter* is the *support* component composed of two sub-components: *Real world sensor* accepts the raw input from multiple sensors and integrates it into a coherent interpretation while *World Model* is concerned with maintaining the robot's model of the world and monitoring the environment for landmarks. *Navigation* is the *middle agency* component, the central intermediate module managing the navigation of the robot. Finally, the *user-level control* is the human-oriented *strategic apex* providing the user interface and overall supervisory functions.

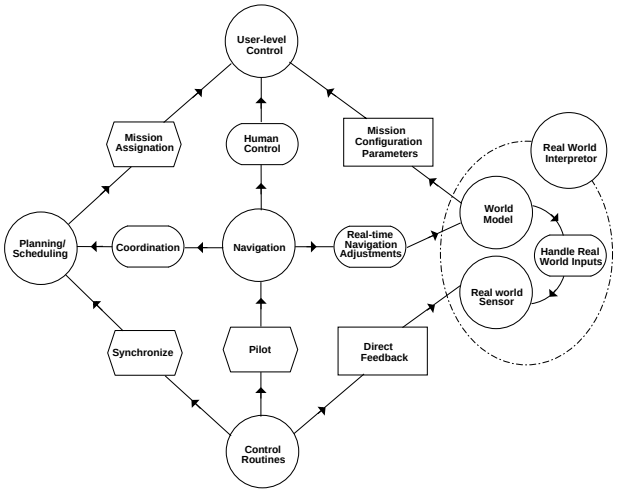


Figure 9. A structure-in-5 mobile robot architecture.

Figure 10 formulates the media robot structure-in-5 in Telos. *MobileRobotClass* is a Telos class, instance of the *StructureIn5Metaclass* specified in Figure 2. This aggregation is composed of five exclusive and dependent parts *ControlRoutinesClass*, *RealWorldInterpreterClass*, *NavigationClass*, *PlanningClass* and *UserLevelControl-Class*, each of them is instance of one metaclass, component of *StructureIn5MetaClass*.

```
TELL CLASS MobileRobotClass
        IN StructureIn5MetaClass WITH
        attribute
            name: String
        part, exclusivePart, dependentPart
            ControlRoutinesClass: OperationalCoreMetaClass
            RealWorldInterpreter: SupportMetaClass
            NavigationClass: MiddleAgencyMetaClass
            PlanningClass: CoordinationMetaClass
            UserLevelControl: ApexMetaClass
END MobileRobotClass
```

Figure 10. Mobile robot structure-in-5 architecture in Telos.

Our second example is a user-to-online-buying application. E-business systems are designed to implement "virtual enterprises". By now, software architects have developed catalogues of web architectural styles (e.g., [3]). Some most common styles are the *Thin Web Client*, *Thick Web Client* and *Web Delivery*. These architectural styles focus on web concepts, protocols and underlying technologies but not on business processes nor non functional requirements of the application. As a result, the organization of the architecture is not described nor the conceptual high-level perspective of the e-business application. The following requirements for a business-to-consumer architecture could be stated according [1]: *Security*, *Availability* and *Adaptability*.

Adaptability (decomposed into *Updatability* and *Maintainbility*) deals with the way the system can be designed using generic mechanisms to allow web pages and user interfaces to be dynamically and easily changed. Indeed, information content and layout need to be frequently refreshed and updated to give correct information to customers or simply be fashionable for marketing reasons.

Availability (decomposed into *Usability*, *Integrity* and *Response Time*): Network communication may not be very reliable causing sporadic loss of the server. There should be concerns with the capability of the e-business system to do what needs to be done, as quickly and efficiently as possible: in particular with the ability of the system to respond in time to client requests for its services. It is also important to provide the customer with a usable application to be usable, i.e., comprehensible at first glimpse, intuitive and ergonomic. Equally strategic to usability concerns is the portability of the application across browser implementations and the quality of the interface.

Security (decomposed into *Authorization* and *Confidentiality*): Clients, especially those on the internet are, like servers, at risk in web applications. It is possible for web browsers and application servers to download or upload content and programs that could open up the client system to crackers and automated agents all over the net. JavaScript, Java applets, ActiveX controls, and plug-ins all represent a certain degree of risk to the system and the information it manages.

Figure 11 suggests a possible assignment of system responsibilities, based on the joint venture architectural style for such a e-business application. The system is decomposed into three principal partners (*Store Front*, *Billing Processor* and *Back Store*) controlling themselves on a local dimension and exchanging, providing and receiving services, data and resources with each other.

Each of them delegates authority to and is controlled and coordinated by the joint management actor (*Joint Manager*) managing the system on a global dimension. *Store Front* interacts primarily with *Customer* and provides her with a usable front-end web application. *Back Store* keeps track of all web information about customers, products, sales, bills and other data of strategic importance to *Media Shop*. *Billing Processor* is in charge of the secure management of orders and bills, and other financial data; also of interactions to *Bank Cpy*. *Joint Manager* manages all of them controlling *security* gaps, *availability* bottlenecks and *adaptability* issues.

To accommodate the responsibilities of *Store Front*, we introduce *Item Browser* to manage catalogue navigation, *Shopping Cart* to select and custom items, *Customer Profiler* to track customer data and produce client profiles, and *On-line Catalogue* to deal with digital library obligations. To cope with the identified software quality attributes (*Security*, *Availability* and *Adaptability*), *Joint Manager* is further refined into four new system sub-actors *Availability Manager*, *Security Checker* and *Adaptability Manager* each of them assuming one of the main softgoals (and their more specific subgoals) and observed by a *Monitor*. Further refinements are shown on Figure 11.

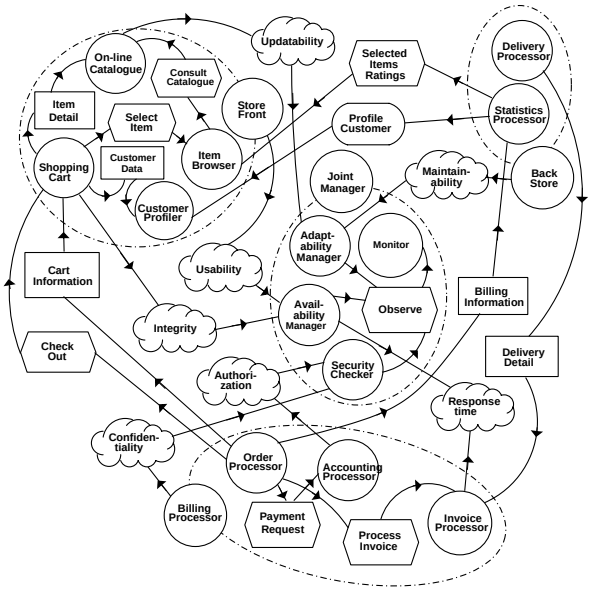


Figure 11. An e-commerce system joint venture architecture.

5. A Requirements-Driven Methodology

This research is conducted in the context of Tropos [1], a software system development methodology which is founded on the concepts of actor and goal. Tropos is intended as a seamless methodology which describes in terms of the same concepts the organizational environment within which a system will eventually operate, as well as the system itself. The proposed methodology supersedes traditional development techniques, such as structured and object-oriented ones in the sense that it is tailored to systems that will operate within an organizational context and is founded on concepts used during early requirements analysis. To this end, we adopt the concepts offered by *i** [17], a modeling framework offering concepts like actor, agent, position and role, as well as social dependencies among actors, including goal, softgoal, task and resource ones.

Tropos spans four phases of software development:

- Early requirements, concerned with the understanding of a problem by studying an organizational setting; the output is an organizational model which includes relevant actors, their goals and dependencies.

- Late requirements, in which the system-to-be is described within its operational environment, along with relevant functions and qualities.

- Architectural design, in which the system's global architecture is defined in terms of subsystems, interconnected through data, control and dependencies.

- Detailed design, in which behaviour of each architectural component is defined in further detail.

6. Conclusion

The paper proposes a set of concepts for specifying software architectures which is inspired by requirements modeling research. As such, we believe that our proposal narrows the gap between a requirements specification and the software architecture to be produced from it. The software architectures produced within our framework are intentional in the sense that components have associated goals that are supposed to fulfil. The architectures are also social in the sense that each component has obligations/expectations towards/from other components. Obviously, such architectures are best suited to open, dynamic and distributed applications, such as those that are becoming prevalent with Web, internet, agent, and peer-to-peer software technologies.

The research reported here is still in progress. We are working on formalizing precisely the styles that have been identified, as well as formalizing the sense in which a particular architecture is an instance of such a pattern.

The organizational styles we have described will eventually define a software architectural macrolevel. At a micro level we will be focusing on the notion of patterns. Many existing patterns can be incorporated into system architecture, such as those identified in [4]. For distributed and open systems characteristics, patterns like the broker, matchmaker, embassy, mediator, wrapper are more appropriate [6, 15]. Another direction for further work is to relate the architectural styles proposed in this work to extentional, classical architectural components such as (software) components, ports, connectors, interfaces, libraries and configurations [12].

References

[1] J. Castro, M. Kolp and J. Mylopoulos. "A Requirements-Driven Development Methodology", To appear in *Proc. of the 13th Int. Conf. on Advanced Information Systems Engineering* (CAiSE'01), Interlaken, Switzerland, June 2001.

[2] L. K. Chung, B. A. Nixon, E. Yu and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*, Kluwer Publishing, 2000.

[3] J. Conallen. *Building Web Applications with UML*, Addison-Wesley, 2000.

[4] E. Gamma., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995

[5] B. Gomes-Casseres. *The alliance revolution : the new shape of business rivalry*, Cambridge, Mass., Harvard University Press, 1996.

[6] S. Hayden, C. Carrick and Q. Yang. "Architectural Design Patterns for Multiagent Coordination" In *Proceedings of the International Conference on Agent Systems '99* (Agents'99), Seattle, WA, May 1999.

[7] H. Mintzberg. *Structure in fives : designing effective organizations*, Englewood Cliffs, N.J., Prentice-Hall, 1992.

[8] R. Motschnig-Pitrik. "The Semantics of PartsVersus Aggregates in Data/Knowledge Modeling", In *Proc. of the 5th Int. Conference on Advanced Information Systems Engineering* (CAiSE'93), Paris, June 1993, pp 352-372.

[9] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis. "Telos: Representing Knowledge About Information Systems" in *ACM Trans. Info. Sys.*, 8 (4), Oct. 1990, pp. 325 – 362.

[10] W. Richard Scott. *Organizations : rational, natural, and open systems*, Upper Saddle River, N.J., Prentice Hall, 1998.

[11] R. Simmons, R. Goodwin, K. Haigh, S. Koenig, and J. O'Sullivan. "A modular architecture for office delivery robots". In *Proc. Of the 1st Int. Conf. on Autonomous Agents* (Agents '97), Marina del Rey. CA, Feb 1997, pp.245 - 252.

[12] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. "Abstractions for software architecture and tools to support them." In *IEEE Transactions on Software Engineering*, 21(4), pp. 314 - 335, 1995.

[13] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Upper Saddle River, N.J., Prentice Hall, 1996.

[14] O. Shehory. *Architectural Properties of Multi-Agent Systems*, Technical report CMU-RI-TR-98-28, Carnegie Mellon University, 1998.

[15] S. G. Woods and M. Barbacci. *Architectural Evaluation of Collaborative Agent-Based Systems.* Technical Report, CMU/SEI-99-TR-025, SEI, Carnegie Mellon University, PA, USA, 1999.

[16] M.Y. Yoshino and U. Srinivasa Rangan. *Strategic alliances: an entrepreneurial approach to globalization*, Boston, Mass., Harvard Business School Press, 1995.

[17] E. Yu. *Modelling Strategic Relationships for Process Reengineering*, Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.

Evolving System Architecture to Meet Changing Business Goals: an Agent and Goal-Oriented Approach

Daniel Gross & Eric Yu
Faculty of Information Studies
University of Toronto
{gross, yu}@fis.utoronto.ca

Abstract

Today's requirements engineering approaches focus on notation and techniques for modeling the intended functionality and qualities of a software system. Little attention has been given to systematically understanding and modeling the relationships between business goals and system qualities, and how these goals are met during architectural design. In particular, modeling must encompass changes to business goals over time and their effects upon a system's architecture. This paper reports on a case study, performed at a telecommunication company, that illustrates the decision-making process regarding architectural changes introduced into an existing switching system product. A notation including goals, strategic agents and intentional dependency relationships is used to support the architectural modeling and reasoning.

Keywords:
Goal, architecture, non-functional requirement, architectural evolution, knowledge-based design

1. Introduction

During architectural design, many of the quality aspects of a system are determined. System qualities are often expressed as non-functional requirements, also called quality attributes [1,2]. These are requirements such as reliability, usability, maintainability, cost, competitiveness, time to market and the like. Many of these originate at the business level, and are better viewed as business goals. Achieving business goals is crucial for system success. As business goals change, the system architecture needs to evolve to ensure continued satisfaction of business goals. Therefore, a systematic modeling framework needs to support linking business goals to architectural design.

Goal-oriented approaches, such as the NFR framework [3,4,5] that treats non-functional requirements as goals to be achieved during the design process, took a significant step in making explicit the relationships between quality requirements and design decisions. The NFR framework uses such goals to drive design [6], to support architectural design [7,8], and to deal with change [9]. While providing a systematic way to deal with the relationships between quality requirements and design, this approach has only limited support for dealing with the functional and structural aspects of the system under development. More recent approaches [8, 10] make a step to further incorporate functional and structural aspects into the design process

This paper proposes a strategic agent-oriented and goal-oriented approach that systematically relates business goals to architectural design decisions and architectural structures during software development and evolution.

This approach emphasizes goal modeling based on the observations that business goals that represent or give rise to non-functional requirements predominate during the architectural design deliberation process, and that changes in business goals may create a need to reevaluate and evolve the architectures of software systems. Goals serve as a guide in the search for design alternatives, and serve as criteria for choosing among them.

This approach uses the agent concept to model human organizations as well as technical components. The rationale for using agents for modeling social concepts is based on the observation that different stakeholders within the development and deployment organizations may have different business goals that they may wish to pursue. These differences may give rise to conflicting interests and rationales. By linking stakeholder goals to the design decision-making process, it becomes possible to express the positive and negative impacts of design decisions upon those goals during software development and evolution [8]. Agents enable the various interests within an organization to be expressed.

The rationale for using agents for modeling technical concepts is based on the observation that the computational elements within coarse-grained software structures, not unlike those within organizational structures, represent focal points for intentional properties, such as design goals and capabilities. Agent concepts lend themselves well to modeling and reasoning about the distribution of capabilities and allocation of responsibilities within a software system, and to show how computational elements are intended to contribute to the overall goals and objectives of the system and the business organization.

The approach uses the notion of strategic agents [15,16] based on the observation that designers of subsystems, concerned with achieving intended design goals, are at the

same time concerned with avoiding or at least mitigating vulnerabilities that might be imposed on them by design decisions taken within other subsystems. This approach models such vulnerabilities, which designers negotiate among themselves during the design process, and highlights how others are expected to contribute in achieving their respective subsystem design goals.

The approach is process-oriented, as it focuses on supporting an iterative decision-making process during design. Design goals are iteratively "reduced" to runtime structures. This is based on the observation that designers establish and refine architectural structures in an iterative manner, where structures first introduced establish coarse-grained partitioning of responsibilities, and iteratively refine to structures that are sufficiently fine-grained to guide implementation of the system.

Finally, based on the observation that designers often reapply previously known design solutions to achieve business- and system-related goals, this approach emphasizes the need to support capturing, generalizing and reapplying design knowledge. Previous design solutions can be sought, based on goals they met, tradeoffs they made, or system structures they created. This supports a knowledge-based approach to design.

The next section describes the modeling approach. Section three introduces the case study. Section four illustrates the modeling approach using the case study. Section five discusses the case study results, while section six concludes and points to future work.

2. An agent & goal-oriented approach

In order to relate business goals to the architectural decision-making process, and to the architectural structures during design, the modeling approach proposes the following main categories of features. Each category is represented as a separate view. All views are iteratively constructed during analysis and design.

- The *design process view* expresses how business goals relate to architectural choices and how changes in business goals invalidate architectural choices, and provides the basis for removing them to choose among alternative design options. This includes support for expressing alternative design paths, and relates alternative choices to the business and system goals that are traded-off against each other.
- The *structural view* provides an architectural description during design that expresses the principal roles played by architectural design elements within a system, and how roles are composed during the design process to arrive at the system design. Architectural elements are characterized by their capabilities, their expectations of other elements, and how they contribute in achieving system- and business-related goals. The notation of this view is taken from the strategic dependency model of the i* framework [15].

  This view provides architectural descriptions of the system at several levels of abstraction, and how these are related to each other during the design process. This includes expressing architectural structures at different stages of completion, together with a description of where architectural structures need further refinement through design decision-making.
- The *organizational view* identifies stakeholders and their goals, and expresses how they depend on each other and on the emerging system design to achieve their goals. This includes support for deducing during the design process how, and upon whom, design choices have an effect. Due to space limitations, this view is not diagrammed in this paper.

This approach also provides knowledge-based support by enabling capturing, storage, retrieval and guidance in reapplying relationships between goals and design elements, when similar goals need to be met during future design efforts.

The organizational view is used to capture the pertinent stakeholders and their business and system related goals. Goals related to functional abilities provide the basis for system requirements, while goals related to business and system qualities provide the basis for non-functional requirements. Goals from the organizational view can be used as a starting point when constructing the design process view.

The design process view is used to construct a goal graph during the development process. The goal graph is used to search for and generate alternative design solutions. Goals denoting functional abilities are refined to alternative design options. Goals denoting non-functional requirements (called softgoals) are used to systematically drive the search for alternative solutions and to determine how each alternative solution relates to pertinent business- and system-related qualities, and to their respective stakeholders described in the organizational view.

The structural view is constructed in accordance with refinements of the goal graph. Existing or new design elements introduced within the structural view are related to architectural decisions described in the goal graph. Alternative refinements provide the basis for searching and identifying refinements within the goal graph.

3. The case study introduced

The case study was performed during the fall of 1999 at a multi-national telecommunication company. We studied a project that intended to utilize WAP/WML[1] technology to

[1] WAP - Wireless Application Protocol, WML - Wireless Markup Language

provide Internet browsing and service provision capabilities to telephone sets [2], which would require architectural changes within their "flagship" switching system.

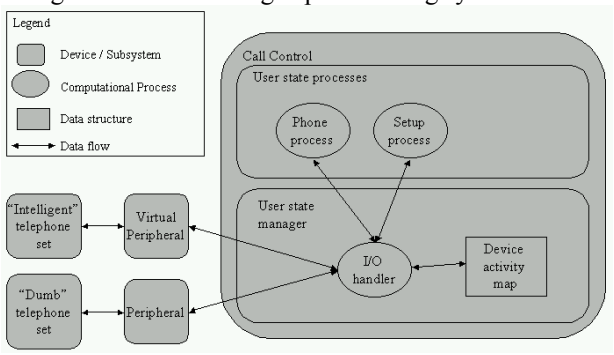


Figure 1: Telephone system architecture

Figure 1 shows the principal architectural elements of the telephone system analyzed during this study. The call control subsystem is responsible for all aspects of a telephone session: establishing calls; enabling features such as call forwarding, call waiting and the like; and terminating calls. All these are implemented by the "phone" process within the call control subsystem. Call control is also responsible for providing to users the set-up functionality for all desired services and features of the telephone set. The "setup" process within the call control subsystem implements this function. Call control is considered the main user application running within the switching system. Figure 1 also shows the peripheral component, which is a proprietary hardware device that connects proprietary telephone sets to the switching system; and the virtual peripheral components, which is software on standard PC-based hardware that emulates a peripheral device for "intelligent" telephone sets. These intelligent telephone sets are connected through a standard IP-based environment (such as an in-house LAN) to the virtual peripheral. The principal architectural question was to find where to place the WML browser component within the components or subsystems of the current telephone system architecture.

1. Within call control
2. Within the virtual peripheral[3]
3. Within the "intelligent" telephone set

It was assumed that the WML browser would be one of many future applications that would be made available on the telephone sets. The question discussed, therefore, was to find where future applications would reside within the telephone system, and what component or subsystem would control what application would interact at what time with the telephone set.

Figure 2 shows how moving from old to new business goals relates to the systems' architecture evolution path. In particular it shows:

- How business goals impact the architecture of a software system. This is shown by the impact links (straight arrows).
- How the current architecture may evolve to the different alternative architectures, each providing different support for adding and controlling new applications. This is shown through architectural evolution links (curved arrows).
- How alternative architectures resemble specializations of a common architectural pattern. This is shown through inheritance links (dotted arrows).

The "curved" links between the architectural alternatives in figure 2 show how "far" the proposed alternative architectures evolve away from the current set of business goals, toward the ideal appliance-based architecture that best achieves the new set of business goals.

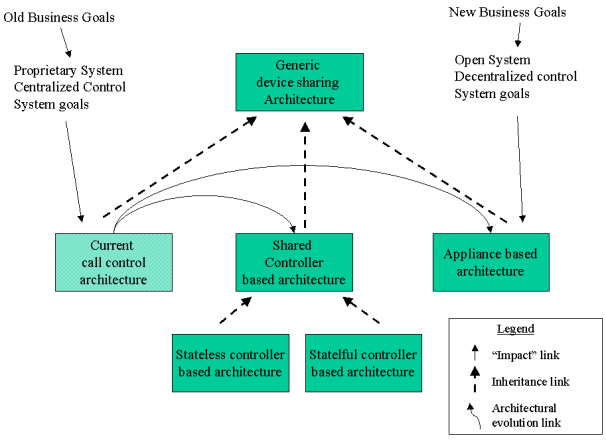


Figure 2: Architectural evolution paths

4. Illustrating the modeling approach

Figure 3 shows part of a goal graph produced during the case study. In the top half of the diagram are pertinent business goals that were voiced by stakeholders. The bottom half of the diagram shows design goals, and the architectural solution elements proposed.

The design goal service_creation_infrastructure_be-_WML_based, shown by the oval modeling element, denotes the overall functional goal to provide the current telephone system with a service creation infrastructure based on WAP/WML technology. This design goal is decomposed, through means-ends links, into the three alternative architectural design solutions. Means-ends links relate alternative design solutions (means) to design goals (ends). The design solutions proposed were master-

[2] Although WAP is used for mobile devices, the project considered its use for their non-mobile telephone sets.

[3] The "regular" peripheral, and the "dumb" phone devices did not support the addition of browser software.

_controlled_WML_based_infrastructure, shared_controller_based_WML_infrastructure, and appliance-_based_WML_infrastructure, each denoted by the hexagonal "design task" symbol.

The first architectural solution, master_controlled-_WML_based_infrastructure, is further decomposed, through task decomposition links, into design solution elements that describe how the WAP/WML architectural elements are added to the current switching system architecture. Since the switching system itself runs on Windows NT, this solution suggests adding the WML browser within the Windows NT environment outside of the switching system. It adds a Browser proxy component within call control as another user state process, and pertinent Browser state information within the user state manager subsystem of call control.

Figure 3 shows how all of these design elements relate through contribution or correlation links to business- or system-related quality goals. A contribution link shows that the design solution was chosen to achieve a business or system goal, while a correlation link denotes a side effect a design solution has on a goal. Both links can be either sufficiently or insufficiently positive, or to some extent, or sufficiently negative, to reject a design option. These degrees of contribution are denoted by the plus and minus signs, and dots within figure 3. They are used to evaluate design solutions through qualitative reasoning, and to direct the exploration of further design alternatives. Placing the browser within Windows NT, for example, has a sufficiently positive effect on reusing commercial software code, which reduces time to market. Placing browser proxy code within the user state process subsystem of call control allows maintaining architectural integrity, which in turn reduces time to market. Maintaining architectural integrity also aids in reducing the complexity of software code, which in turn reduces the cost of software development. However, placing the browser proxy within call control has a sufficiently negative impact on the architectural evolution goals for the switching system, by further entrenching the current architectural principles — rather than moving away from them or at least creating "evolvable" components that are reusable within next generation telephone systems.

In the middle of figure 3 we can see that for the shared_controller_based_WML_infrastructure design task two alternative design options were identified. This is shown by refining the design task into a corresponding design goal, WML_infrastructure_be_shared_controller_based, to denote that this design task, when further explored, raises further design alternatives. This design goal is then refined into the two alternatives: stateless_shared_controller_WML_infrastructure and stateful_shared_controller_WML_infrastructure.
Figure 3 shows how stateful_shared_controller_WML-_infrastructure is further refined, through task decomposition links, into design elements that are proposed as additions to the current switching system architecture. Each one of these design elements contributes to business and system goals. Figure 3 does not show all contribution or correlation links identified during the case study, but only the most pertinent ones for our discussion. For example, it shows that placing the Browser within the virtual peripheral contributes positively to the architectural evolution goal (namely the ability to provide "evolvable" state manager components to future switching systems). Adding the stream interpreter component, which is another design element, both affects adversely the performance of telephone sets attached to the system, and increases the likelihood of processing errors due to the difficulty of interpreting data streams without all the knowledge of its meaning, which resides within call control.

Let us now describe the structural view, and how it relates to the modeling elements in the goal graph. Figure 4 shows the structural view of the master_controlled_WML-_based_infrastructure design alternative, and how it relates to the generic device sharing architecture. The top part of figure 4 shows the structures defined for the device sharing architecture. These are the shared_device, the device_controller and the application agent. An agent represents a computational component during design. It encapsulates the design goals it achieves, the capabilities it provides, the capabilities it offers to other parts of the system, and the quality constraints it depends on. Figure 4 shows how the design of each agent depends on other agents through goals, tasks and resource dependencies. For example, the resource dependency data_stream between the application and the device_controller agent denotes the expectation of each agent to receive such a data stream from the other during runtime. The goal dependency exclusive_ownership_granted between the application and the device_controller agent denotes the expectation of the application agent that the device_controller agent will provide it with exclusive access to the data stream received from, and sent to the shared_device. This expectation expressed by the goal dependency is a design goal that is directed from the application agent toward the device_controller agent. The dependency does not prescribe how the device_controller agent will achieve this design goal, but only expects that it will be achieved during further design. Furthermore, the goal dependency denotes that it is up to the designer of the device_controller to decide how to achieve that design goal, and thus how to implement such exclusive ownership over data streams within the device controller component. The two softgoal dependencies, performance and minimize_processing_errors are quality attributes that the application agent depends on and wishes to have satisfied. These quality attributes serve as design constraints imposed by the application agent on the

device_controller agent in its exploration of design alternatives. Only those design alternatives that provide good performance and minimize processing errors are deemed acceptable to the application agent.

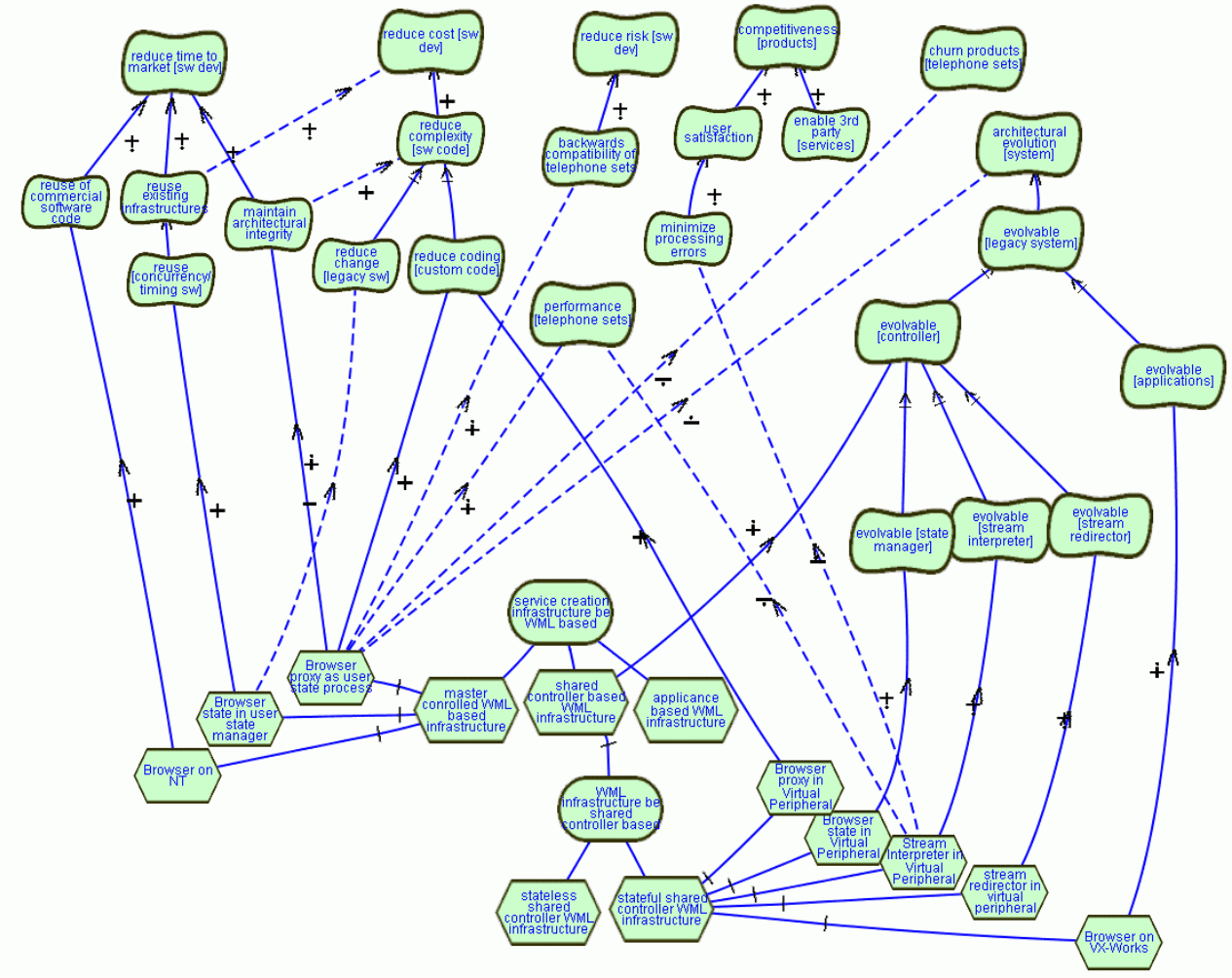


Figure 3: Goal graph denoting a design process with alternative architectural choices

For completeness, let us mention the send_state-_changed_commands task dependency between the device_controller and the shared_device. A task dependency denotes a design goal having constraints to a particular implementation. In our example, the device_controller agent expects the shared_device agent to send commands reflecting state change information, and expects that such commands will appear within the data stream.

This example highlights the difference between a structural view expressed in an agent-oriented manner and the common blocks-and-arrows diagrams. It shows how agents in conjunction with strategic dependencies are used to represent computational elements where design goals still exist and a design process still needs to unfold. Goal dependencies direct further design deliberations, while softgoals provide a means to constrain the selection of future proposed design alternatives in terms of quality requirements that need to be achieved within the system or the organization. Task dependencies provide a means to constrain design to exhibit particular functional features. Blocks-and-arrows diagrams represent final design choices and do not guide where and how further design choices need to be made.

The top part of figure 4 further shows that the device_controller agent is made out of three sub-agents, the command_interpreter, state_manager and data_stream_redirector agents, each performing part of the controller tasks. The command_interpreter scans the incoming data stream from the shared_device for commands to switch applications. The state_manager maintains a record of what application currently "owns" the shared device, and what application needs to be activated based on incoming commands. Finally, the data_stream_redirector agent directs the data stream between the shared device and the application that

currently has exclusive ownership. Any architecture that makes use of this generic device-sharing architecture needs to incorporate these three agents within its design. The bottom part of figure 4 shows how this device sharing architecture, and in particular how these three components within the device_controller agent, are allocated within the master_controlled_WML_based_infrastructure architectural alternative described in the goal graph in figure 3. It shows the call_control agent and its two sub-agents, the I/O_handler and the user_services agent. User_services is part of the user state processes subsystems and denotes all services available within call control. The user_services agent depends on I/O_handler to provide it with exclusive_telephone-_set_ownership and to receive a user_input_data. The I/O_handler in turn depends on the user_services to receive signal&response_data, which it directs to the telephone_set agent. The telephone_set depends on the I/O_handler to be shared, and to receive signal (i.e. commands in telephone set terminology) and response data streams.

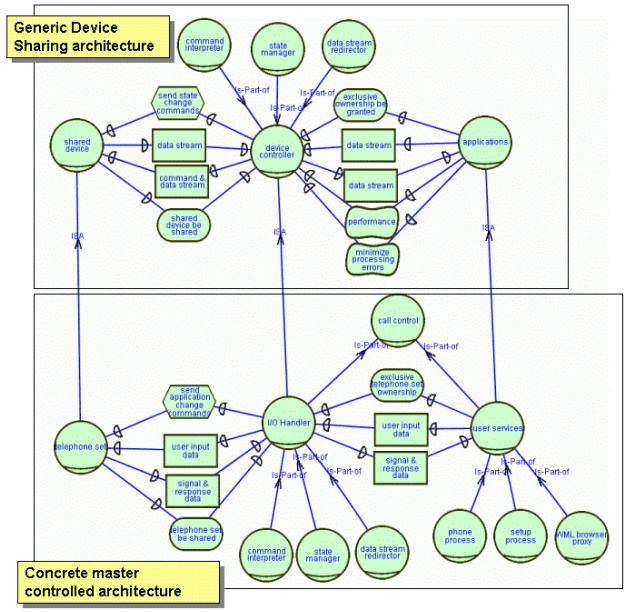


Figure 4: Abstract device sharing architecture and concrete master controlled WML infrastructure

Figure 4 further shows that the master_controlled-_WML_based_infrastructure architecture is a specialization of the generic device sharing architecture. The telephone_set agent is a shared_device, the I/O_handler is a device_controller, and user_services is an application. These relationships or "mappings" are denoted by "ISA" links. When mapping agents from the generic device sharing architecture to the more concrete master-controller architecture, the corresponding dependency links among agents may also be mapped. For example, the data_stream dependencies among the shared_device and the device_controller agents are created between their "counterparts", the telephone_set and the I/O_handler agents, albeit often renamed to fit the domain meaning of those dependencies. Mapping dependencies, through ISA links, from abstract to more concrete architectures is a design activity that needs judgment of designers. Unlike "conventional" inheritance, ISA links denote possible mappings available. Designers, in conjunction with the design process view, decide whether and what dependencies to map onto what agents, and what domain meaning and possible further constraining specializations to provide.

Sub-agents are also "inherited" from the abstract architectural view to the more concrete one. The state_manager, user_input_data_redirector and change_command_interpreter that are part of the I/O_handler are all inherited from the device-_controller agent. All these agents are allocated as described by design elements within the goal graph in figure 3, to achieve good performance and to minimize processing errors. Both good performance and minimizing processing errors are achieved by maintaining the centralized way that incoming signals from the telephone sets are interpreted, and by not having external computational elements performing similar tasks elsewhere. The other alternatives described in the goal graph allocate the state_manager, data_stream-_redirector and command_interpreter differently within the system to make different tradeoffs among these quality requirements, in particular to create an architecture that is more favorable to the architectural evolution goals. Finally, figure 4 shows that the WML_Browser_proxy agent is considered as a part of user services, since it is considered as an application, and in this architecture alternative, applications run within user services.

Let us now illustrate how the stateless and the stateful shared controller-based architectures are derived, through design steps described in the goal graph, from the generic device sharing architecture. We will see how goals and softgoal dependencies provide guidance in exploring alternatives during the design process. Each design task within the goal graph (denoted by the hexagonal symbol) refers to the structural view. Refining such tasks either through means-ends links or task-decomposition links into sub-tasks prompts the creation of additional components within the structural view. Goals and softgoals, both within the goal graph and within the structural views, guide the search for alternative design refinements.

The "legacy system with new extensions" structural view in figure 5 denotes an abstract architecture for extending legacy systems with new functionality. It defines two principal agents, the legacy_system and the new_system_extension agent. The goal and softgoal

dependencies between these two agents describe the design expectations each agent has of the other, which should be fulfilled during the subsequent design efforts. In particular, the view shows that the legacy_system agent is concerned with performance and maintain_architectural_integrity. On the other hand, the new_system_extension agent is concerned with creating "evolvable components" within the legacy system. These are components that are designed both to be implemented within the legacy system and to be reused within new systems ("next generation systems") that will comply with evolved system architectures.

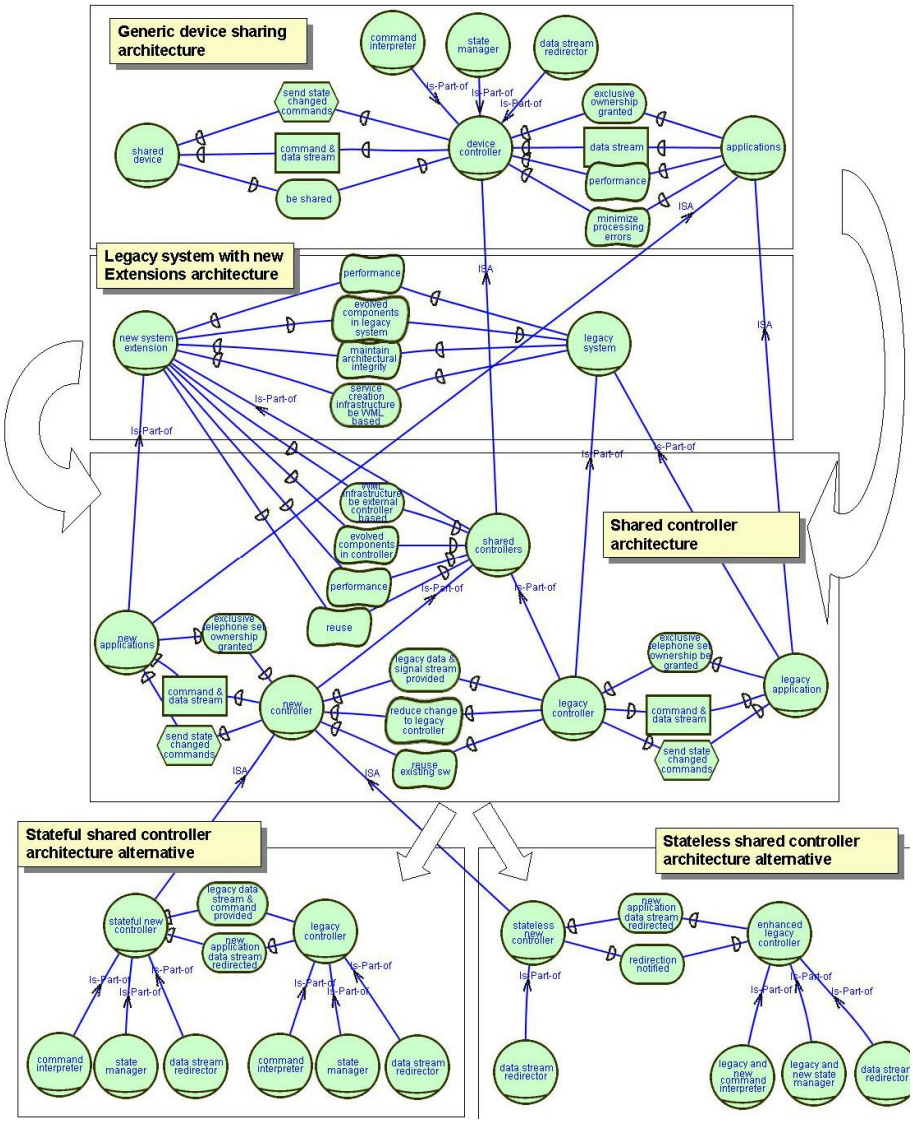


Figure 5: Shared-controller architecture alternatives

As discussed earlier, the goal graph in figure 3 shows that the WML_based_service_creation_infrastructure design solution can be achieved through three different architectures, each one based on a different specialization of the generic device-sharing architecture. Choosing this design task corresponds to consolidating the generic device-sharing architecture and the "legacy system with extension architecture" into the shared controller architecture structure described in figure 5. Note that choosing the shared controller architecture already achieves quality goals, such as creating evolvable controller components. This is shown in figure 3 through a contribution link from shared_controller_based-_WML_infrastructure to the evolvable [controller] softgoal. Having achieved this softgoal, further goals and softgoals are now identified that need to be achieved within the shared controller design, namely evolvable [state_manager], evolvable [stream_interpreter] and evolvable [stream_redirector] components. These are identified through the structure of the controller agent as shown in the structural view. The need to now achieve

these softgoals is shown by these softgoals and their contribution links in figure 3.

This shared-controller architecture introduces the shared_controller agent, which is composed of the new_controller agent and a legacy_controller agent. It further introduces two application agents, the new_application and legacy_application agents. The dependencies among new_application and new_controller, and legacy_application and legacy-_controller agents, correspond to the dependencies defined among the application and device_controller agents within the generic device-sharing architecture. These are inherited according to the inheritance links defined between the agents of both structural views. This shared-controller architecture provides architectural structure for any system that wishes to provide two focal points of control, for which legacy applications control is provided within the legacy system and for new applications control is provided within an additional component or subsystem.

A key question during the following design task is how exactly control is shared between the new_controller and legacy_controller agents such that the right tradeoffs are found among 1) maintaining the architectural integrity of the legacy system 2) optimizing performance of the system 3) providing further evolvable components 4) reducing change to the legacy controller and, finally, 5) reuse of existing software within the system. All these quality requirements are described in figure 5. The first ones (1-2) are inherited from the dependencies between the new_system_extension and legacy_system agents. The others (3-5) are represented by the dependencies between the new_controller and legacy_controller agents.

Figure 5 shows the structural view of the major components of the stateful shared controller and the stateless shared controller architectural alternatives. The goal graph in figure 3 shows how each alternative trades-off differently the above-mentioned quality requirements. Figure 5 shows in what way each alternative differs, in terms of allocating the device_controller sub-agents inherited from the generic device-sharing architecture between the new_controller and the legacy_controller. The stateful architectural alternative inherits all sub-agents to both the legacy and new controllers. The stateless architectural alternative inherits only the data_stream_redirector to the new controller (denoted by the stateless_new_controller), and makes it dependent on an enhanced version of the legacy_controller agent. This enhanced agent processes commands, manages the state of new applications and notifies the stateless controller of when to redirect and stop redirecting data streams. Figure 5, thus, demonstrates how dependencies among agents, in conjunction with the goal graph in figure 3, serve as criteria for searching and evaluating further alternative architectural designs.

5. Discussion

The requirements engineering research community has recognized the importance of goal modeling [11, 12, 13, 15,16,17]. However, goals are typically used to guide the establishing of requirements or designing of business processes, and serve as criteria for requirements completeness. The approach expounded in this paper recognizes the need to utilize goals during analysis and during the design process. This aids in representing the "unfolding" of the design decision process over time. Goals during design provide a focal point for unmet design requirements without (over) committing to particular design solutions.

This approach allows representing the many stages of completion through which design solutions move, and the stakeholder or system goals still to be addressed during further design. Goals denoting quality requirements provide an effective means for denoting constraints over further design efforts, and criteria for choosing among alternatives. Research in architectural design has given rise to notations that emphasize the compositional and behavioral aspect of coarse-grained system structures [14]. Quality attributes, or non-functional requirements, were identified as key driving forces, and rationales for different compositional system configurations. However, their treatment is often informal and not included in the architectural design notation. Both research communities recognize the importance of such links. However, little research has been done so far in bridging the requirements and architectural design gap.

The concepts of business goals and their relationships to functional and non-functional system requirements are not clear-cut. In this paper we took the stance that business goals are purposes that the business organization desires to achieve, both in the short and in the long term. Such goals are not necessarily tied to one product, but may relate to all product portfolios developed, maintained and evolved in the organization. Such goals originate from a variety of organizational and marketplace stakeholders. They are used to negotiate and determine functional and non-functional requirements, and, as we have seen, also architectural design decisions. For the purpose of modeling the architectural evolution process we did not feel the need to make a clear distinction between goals that originated from the business level and goals that represented system requirements. Both are seamlessly linked together through contribution (and correlation) links, and reside within the context of business and system development stakeholders. Precise boundaries might be needed for areas such as contracting and other legal purposes.

During the case study it was observed that the generic device-sharing architecture pattern, although being technical in nature, lent itself well to describing alternative business models pursued by the organization. System architectures that assigned the application and control components to one computational element in the target architecture pursued a centralized business model. Architectures that distribute these components, in particular among computational elements belonging to applications or devices under the jurisdiction of other organizations, pursue a decentralized and distributed business model. During the case study, the design decision to allow the organization's telephone sets to be operated by providers of competing switching systems would pursue both an open and decentralized business model.

An important feature of the "mapping" mechanism proposed is its ability to determine conformance among architectures. When changing the design of the concrete architecture, it can be determined whether it still conforms or violates one or more of the abstract architectures from it took over components and dependencies from. For example, figure 5 does not show how the WML browser proxy appeared within the switching system architecture. Two architectural patterns were, in fact, applied. One is the abstract architecture describing the WAP/WML reference architecture, which defines the WML browser agent, and the other describes how proxy components are utilized when wishing to split components among two spatial locations, while maintaining both parts as a logical computational unit. The structural view, in conjunction with the goal graph, allows representing such relationships among various "reference architectures" and how and why each contributes to the establishing of solution architectures.

6. Conclusion and future work

The case study highlighted the need for a modeling approach that supports modeling and analyzing how business goals relate to the architectural decision-making process, and how changing business goals give rise to alternative architectural choices and solution structures. It illustrated the need to describe the organizational stakeholders, their goals, and how these are affected by alternative choices during the design process. The case study highlighted the utility of goal modeling for expressing alternative design choices, and to serve as criteria during design deliberation. It showed the utility of using agents and goal concepts for modeling architectural solution structures. Agents were used to describe architectural distribution of capabilities, while goals were used as a focal point for expressing where within architectural structures further design choices needed to be made. Future work needs to focus on refining the integrated modeling framework, further formalizing the relationships among its diagrams, and investigating how its abstraction and mapping facilities can support knowledge-based tools that provide systematic design guidance and analysis support.

6. Acknowledgements

We like to thank the anonymous reviewers for their valuable comments; Tauba Staroswiecki, Douglas Anderson for their proof reading; and CITO, NSERC and our industrial research partner for their financial support.

7. References

[1] Boehm BW. Characteristics of software quality. North-Holland Pub. Co., Amsterdam New York 1978.
[2] Bowen TP. Wigle GB. Tsai JT. Specification of software quality attributes (Report RADC-TR-85-37).
[3] Chung L. Representing and using non-functional requirements: a process-oriented approach. Department of Computer Science University of Toronto. Toronto 1993.
[4] Chung L. Nixon B. Yu E. et al. Non-functional requirements in software engineering. Kluwer Academic, Boston 2000.
[5] Mylopoulos J. Chung L. Nixon B. Representing and using nonfunctional requirements: a process-oriented approach. IEEE Transactions on Software Engineering 1992; 18(6).
[6] Chung L. Nixon B. Yu E. Using quality requirements to systematically develop quality software. Proceedings of the 4th Int. Conf. on Software Quality. McLean, VA, USA. 1994.
[7] Chung L. Nixon B. Yu E. Using non-functional requirements to systematically select among alternatives in architectural design. Proceedings of the First International Workshop on Architecture for Software Systems. Seattle, Washington. 1995.
[8] Chung L. Gross D. Yu E. Architectural design to meet stakeholder requirements. In: Donohue, P(ed.). Software architecture. Kluwer Academic Publishers. San Antonio, Texas, USA 1999. pp 545-564.
[9] Chung L. Nixon B. Yu E. Dealing with change: An approach using non-functional requirements.
[10] D. Gross, E. Yu, From Non-Functional Requirements to Design through Patterns, Requirements Engineering. (to appear).
[11] A. I. Anton, "Goal-based Requirements Analysis." Proc.2nd IEEE Int. Conf. Requirements Eng. April 1996.
[12] A. Dardenne, A. van Lamsweerde and S. Fickas, Goal-Directed Requirements Acquisition, Science of Computer Programming, 20, pp. 3-50, 1993.
[13] S. Jacobs and R. Holten, "Goal-Driven Business Modelling – Supporting Decision Making within Information Systems Development," Proc. Conf. On Organizational Computing Systems, Milpitas, Calif. 1995, pp. 96-105.
[14] Shaw, M. and Garlan, D. (1996) “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall.
[15] Yu E. Modelling strategic relationships for process reengineering. Ph.D. thesis, Dept. of Computer Science, University of Toronto. 1995.
[16] E. Yu Agent Orientation as a Modelling Paradigm, Wirtschaftsinformatik. 43(2) April 2001. pp. 123-132.
[17] E. Yu and J. Mylopoulos `Why Goal-Oriented Requirements Engineering', Proceedings of the 4th Int. Workshop on Requ. Engineering: Foundations of Software Quality (8-9 June 1998, Pisa, Italy). E. Dubois, A.L. Opdahl, K. Pohl, eds. Presses Universitaires de Namur, 1998. pp. 15-22

From Requirements to Architectural Design –Using Goals and Scenarios

Lin Liu Eric Yu
Faculty of Information Studies, University of Toronto
{liu, yu}@fis.utoronto.ca

Abstract

To strengthen the connection between requirements and design during the early stages of architectural design, a designer would like to have notations to help visualize the incremental refinement of an architecture from initially abstract descriptions to increasingly concrete components and interactions, all the while maintaining a clear focus on the relevant requirements at each step. We propose the combined use of a goal-oriented language GRL and a scenarios-oriented architectural notation UCM. Goals are used in the refinement of functional and non-functional requirements, the exploration of alternatives, and their operationalization into architectural constructs. The scenario notation is used to depict the incremental elaboration and realization of requirements into architectural design. The approach is illustrated with an example from the telecom domain.

1. Introduction

In the context of Requirement Engineering and system architectural design, goal-driven and scenario-based approaches have proven useful. In order to overcome some of the deficiencies and limitations of these approaches when used in isolation, proposals have been made to couple goals, scenarios and agents together to guide the RE to architectural design process. As there are both overlap and gaps between these approaches, their interactions are complicate and highly dynamic.

In General, goals describe the objectives that the system should achieve through the cooperation of agents in the software-to-be and in the environment. It captures "why" the data and functions were there, and whether they are sufficient or not for achieving the high-level objectives that arise naturally in the requirement engineering process. The integration of explicit goal representations in requirement models provides a criterion for requirement completeness, i.e. the requirements can be judged as complete if they are sufficient to establish the goal they are refining.

Scenarios present possible ways to use a system to accomplish some desired functions or implicit purpose(s). Typically, it is a temporal sequence of interaction events between the intended software and its environment (composed of other systems or humans). A scenario could be expressed in forms such as narrative text, structured text, images, animation or simulations, charts, maps, etc. The content of a scenario could describe either system-environment interactions or events inside a system. Purpose and usage of scenarios also varies greatly. It could be used as means to elicit or validate system requirements, as concretization of use-oriented system descriptions, or as basis for test cases. Scenarios have also become popular in other fields, notably human-computer interaction and strategic planning.

In this paper, we explore the combined use of goal-oriented and scenario-based models during architectural design. The GRL language is used to support goal and agent oriented modelling and reasoning, and to guide the architectural design process. The UCM notation is used to express the architectural design at each stage of development. The scenario orientation of UCM allows the behavioral aspects of the architecture to be visualized at varying degrees of abstraction and levels of detail.

In next section, basic concepts of GRL and UCM are introduced. In Section 3, we summarized our approach of using GRL and UCM together to incrementally modelling requirements and architectural design. In section 4, a case study in wireless telecommunication domain is used to illustrate the proposed approach. In section 5, related works are discussed. Conclusions and future works are in section 6.

2. GRL and UCM

2.1 GRL

Goal-oriented Requirement Language (GRL) is a language for supporting goal and agent oriented modeling and reasoning of requirements, especially for dealing with Non-Functional Requirements (NFRs)[4][11]. It provides constructs for expressing various types of concepts that appear during the requirement and high-level architectural

design process. There are three main categories of concepts: intentional elements, links, and actors. The intentional elements in GRL are goal, task, softgoal and resource. They are intentional because they are used for models that allow answering questions such as why particular behaviors, informational and structural aspects were chosen to be included in the system requirement, what alternatives were considered, what criteria were used to deliberate among alternative options, and what the reasons were for choosing one alternative over the other.

A GRL model can either be composed of a global goal model, or a series of goal models distributed in several actors. If a goal model includes more than one actor, then the intentional dependency relationships between actors could also be represented and reasoned about. In this paper, the distributed goal models will not be discussed, we may have another paper studying the roles of agent-orientation in requirement and architectural design.

A goal is a condition or state of affairs in the world that the stakeholders would like to achieve. In General, how the goal is to be achieved is not specified, allowing alternatives to be considered. A goal can be either a business goal or a system goal. A business goal express goals regarding the business or state of the business affairs the individual or organization wishes to achieve. System goal expresses goals the target system should achieve, which, generally, describe the functional requirements of the target information system. In GRL graphical representation, goals are represented as a rounded rectangle with goal name inside.

A task specifies a particular way of doing something. When a task is specified as a sub-component of a (higher-level) task, this restricts the higher-level task to that particular course of action. Tasks can also be seen as the solutions in the target system, which will satisfice the softgoals (called operationalizations in NFR) or achieve goals. These solutions provide operations, processes, data representations, structuring, constraints and agents in the target system to meet the needs stated in the goals and softgoals. In GRL graphical representation, tasks are represented as a hexagon with task name inside.

A softgoal is a condition or state of affairs in the world that the actor would like to achieve, but unlike in the concept of (hard) goal, there are no clear-cut criteria for whether the condition is achieved, and it is up to subjective judgement and interpretation of the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated softgoal. Softgoal is used to represent NFRs in the future system. Non-functional requirements, such as performance, security, accuracy, reusability, interoperability, time-to market and cost are often crucial for the success of a software systems. They should be addressed as early as possible in a software lifecycle, and be properly reflected in software architecture before a commitment is made to a specific implementation. In GRL graphical representation, A softgoal, which is "soft" in nature, is shown as an irregular curvilinear shape with softgoal name inside.

A resource is an (physical or informational) entity, with which the main concern is whether it is available. Resources are shown as rectangles in GRL graphical representation.

Intentional links in GRL includes means-ends, decomposition, contribution, correlation and dependency. Means-ends is used to describe how goals are in fact achieved. Each task connected to a goal by means-ends link is an alternative means for achieving the goal. Decomposition defines what other sub-elements needs to be achieved or available in order for a task to be performed. Contribution describes how softgoals, tasks, links contribute to others. A contribution is an effect that is a primary desire during modelling. Contributions can be either negative, or positive, can be either sufficient or partial. Following are the graphical representations for links.

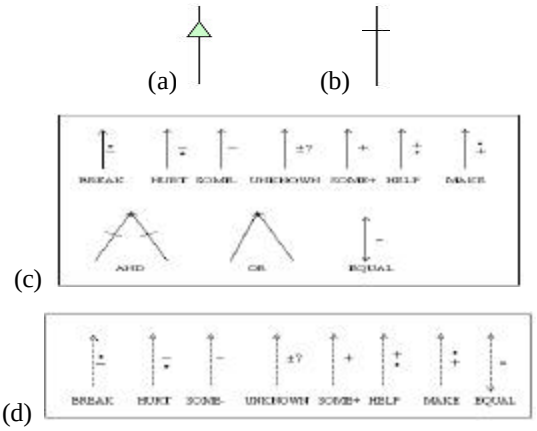


Figure 1 (a) Means-Ends; (b)Decomposition; (c) Contribution; (d) Correlation

2.2 UCM

Use Case Maps (UCM)[2][3] provides a visual notation for scenarios, which is proposed for describing and reasoning about large-grained behavior patterns in systems, as well as the coupling of these patterns. A new thing UCM offers in relation to architecture is that it provides a behavioral framework for making architectural decisions at a high-level of design, and also characterizing behavior at the architectural level once the architecture is decided.

Use Case Maps notation (UCMs) employ scenario paths to illustrate causal relationships among responsibilities. Furthermore, UCM provides an integrated view of behavior and structure by allowing the superimposition of scenario paths on a structure of abstract components. The

combination of behavior and structure in UCMs enables architectural reasoning. Scenarios in UCM can be structured and integrated incrementally. This enables reasoning about and detection of potential undesirable interactions of scenarios and components. Furthermore, the dynamic (run-time) refinement capabilities of the UCM language allow for the specification of (run-time) policies and for the specification of loosely coupled systems where functionality is decided at runtime through negotiation between components.

The UCM notation is mainly composed of path elements, and also of components. The basic path notation address simple operators for causally linking responsibilities in sequences, as alternatives, and in parallel. More advanced operators can be used for structuring UCMs hierarchically and for representing exceptional scenarios and dynamic behavior. Components can be of different natures, allowing for a better and more appropriate description of some entities in a system.

Basic elements of UCMs are start points, responsibilities, end points and components. Starting points are filled circles representing pre-conditions or triggering causes. End points are bars representing post-conditions or resulting effects. Responsibilities are crosses representing actions, tasks or functions to be performed. Components are boxes representing entities or objects composing the system. Paths are the wiggle lines that connect start points, responsibilities and end points. A responsibility is said to be bound to a component when the cross is inside the component. In this case, the component is responsible to perform the action, task, or function represented by the responsibility.

Alternatives and shared segments of routes are represented as overlapping paths. An OR-join merges two (or more) overlapping paths while an OR-fork splits a path into two (or more) alternatives. Alternatives may be guarded by conditions represented as labels between square brackets. Concurrent and synchronized segments of routes are represented through the use of a vertical bar. An AND-join synchronizes two paths together while an AND-fork splits a path into two (or more) concurrent segments.

When maps become too complex to be represented as one single UCM, a mechanism for defining and structuring sub-maps become necessary. A top level UCM, referred to as a root map, can include containers (called stubs) for sub-maps (called plug-ins). Stubs are represented as diamonds. Stubs and plug-ins are used to solve the problems of layering and scaling or the dynamic selection and switching of implementation details.

Other notational elements include: timer, abort, failure point, and shared responsibilities. Detailed introduction and example of these concepts can be found in [2] [3].

Although UCM could represent the alternatives of system architectural design precisely in a high-level way, the tradeoffs between these alternatives, and the intentional features of making a design decision could not be explicitly shown in UCM models. And inevitably, as other scenario-based approaches, UCM models are partial.

GRL provides support for reasoning about scenarios by establishing correspondences between intentional GRL elements and functional components and responsibilities in scenario models of UCM. Modelling both goals and scenarios is complementary and may aid in identifying further goals and additional scenarios (and scenario fragments) important to architectural design, thus contributing to the completeness and accuracy of requirement, as well as quality of architectural design.

3. Modelling Methodology with GRL+UCM

A complete requirement specification should clarify the objectives of a system to be achieved, the concrete behaviors and constraints of the system-to-be, and the entities being responsible for certain functions in that system.

Goal-based approaches focuses on answering the "why" questions of requirements (such as "why the system needs to be redesigned?" "Why a new architecture for TSMA is necessary?"), the strength of these approaches is that they could cover not only functional requirements but also non-functional requirements (in other words, the quality requirements). Although goal-orientation is highly appropriate for requirement engineering, goals are sometimes too abstract to capture at once. Operational scenarios of using the hypothetical system are sometimes easier to get in the first place than some goals that can be made explicit only after deeper understanding of the system has been gained.

In our approach, GRL models are created, the original business goals and non-functional requirements are refined and operationalized, until some concrete design decisions are launched. These design decisions are then further elaborated into UCM scenarios. In the scenario authoring of this step, "how" questions are asked instead of "what".

At the same time, UCM scenarios are used to describe the behavioral features and architectures of the intended system in the restricted context of achieving some implicit purpose(s), which basically answers the "what" questions, such as "what the system should do as providing a in-coming call service?" "What is the process of wireless call transmitting?" Then, by issuing "why" questions referring to these scenarios (e.g. "why to reside a function entity in this network entity instead of the other?") some implicit system goals are further disclosed.

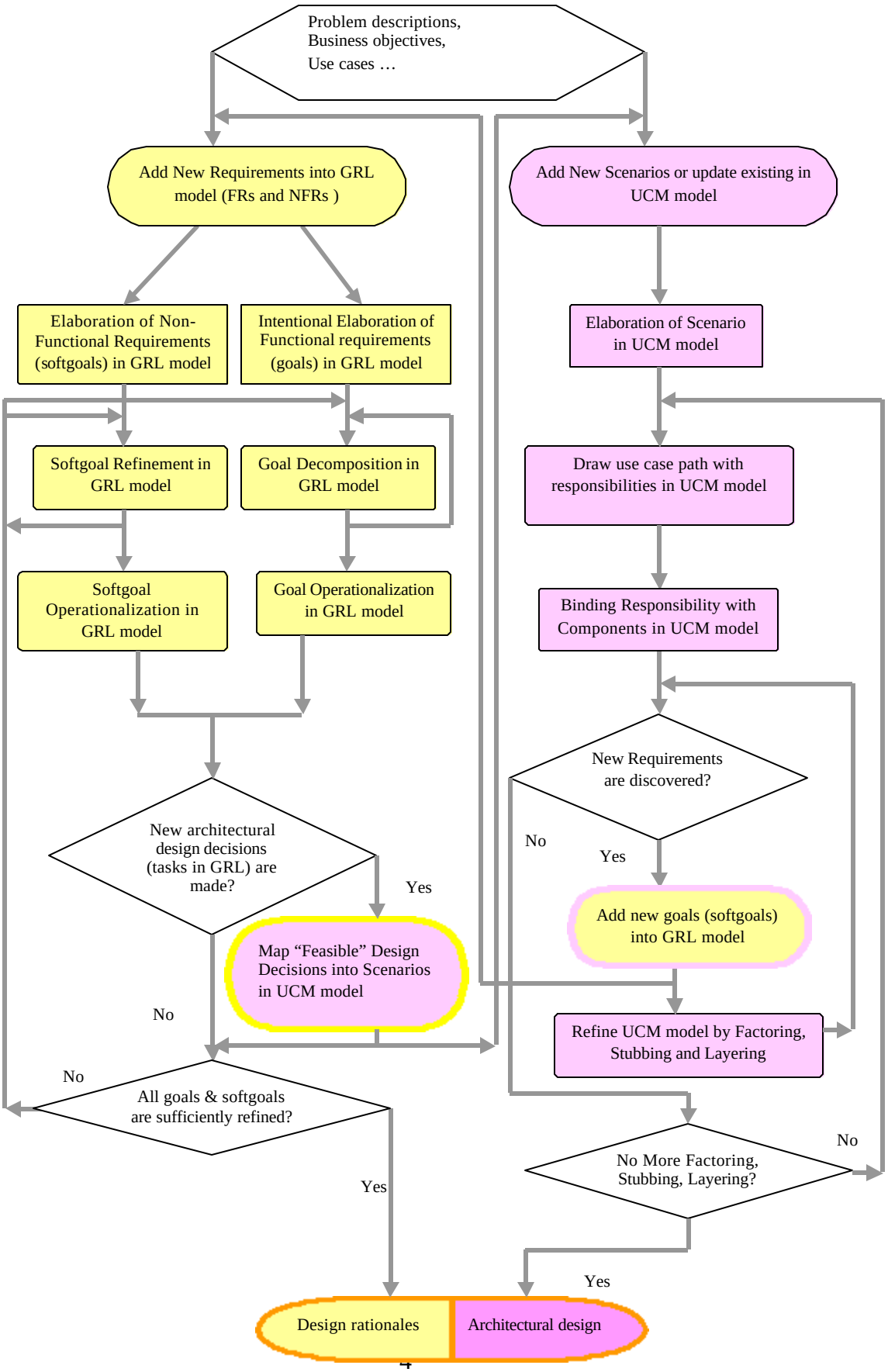


Figure 2. Integration of Goal-Oriented and Scenario-based Modelling

The GRL-UCM combination aims to elicit, refine and operationalize requirements incrementally until a satisfying architectural design is launched. The general steps of the process are illustrated in Figure 2.

4. Illustration with examples

To illustrate the interleaved application of GRL and UCM, we use an example from the mobile telecommunication systems domain [9]. A mobile switching center (MSC) is required to support narrowband and wideband voice, data

Figure 3: Original Goal Model with one functional goal and two non-functional goals

and imaging services and so on. We use GRL and UCM together to trace the process from capturing the original business objective, to refining and operationalizing this objective, and to trading off each architectural design options.

Step 1: GRL Model- Original functional and non-functional requirements are represented as three floating nodes in Figure 3. The goal node in the middle represents the functional requirement on the TDMA that it must support Narrowband and wideband voice, data and image services. There are two quality requirements identified at the very beginning, one is to maximize the call capacity in the new TDMA architecture, the other is to minimize the cost of the infrastructure.

Step 2: UCM Model- The essential scenario that implements the functional goal in above GRL model is given in Figure 4. The scenario path (denoted by the

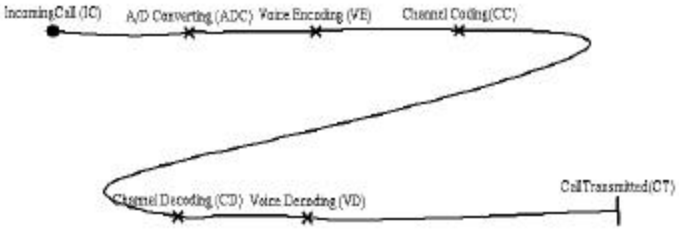


Figure 4: Unbound use case path with responsibilities

wiggle line) represents a causal sequence of responsibilities (denoted by a cross) that is triggered by an initial event (denoted by a filled circle), resulting in a terminating event (denoted by a bar). The responsibilities are not bound to any components.

Step 3: UCM Model – Binding Responsibilities to components of the future system.

The following UCM diagram (Figure 5) shows the existing TDMA architecture. In this architecture, the Decoder of

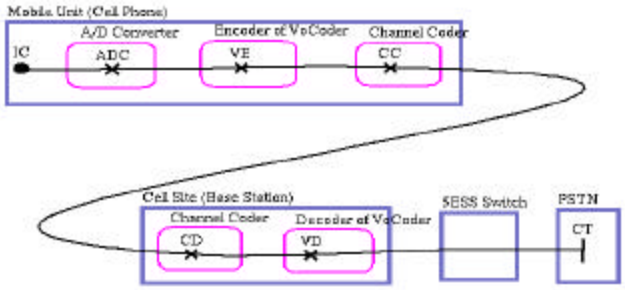


Figure 5: Bound use case path with functional objects and physical entities

the Voice Coder is located in the base station. This implies that the 64-kb/s PCM of decoded voice will be transmitted out of the cell site to the switch for each call, requiring an entire Digital Signal level 0 channel (DS0) to support the 64-kb/s signal.

Step 4: GRL Model – Goal Refinement and Operationalization. In the goal model in Figure 6, the original functional goal is connected to the task node representing current solution for TDMA. It can be seen that current solution can cause some delay per call, which may negatively influence the voice quality of the call, and call capacity of the system. This solution does not use packet switching protocol enough, so cost could not be saved. Traffic performance between base station and switch is also low.

With current infrastructure, the efficiency of TDMA is barely equivalent to that of analog system, which means the requirements on improving the capacity, quality, cost and performance are all weakly denied.

Step 5: UCM Model – Change the Binding of Responsibilities.

As the above design could not satisfy the non-functional requirements of the infrastructure, other options should be explored. The UCM model (in Figure 7) describes a new

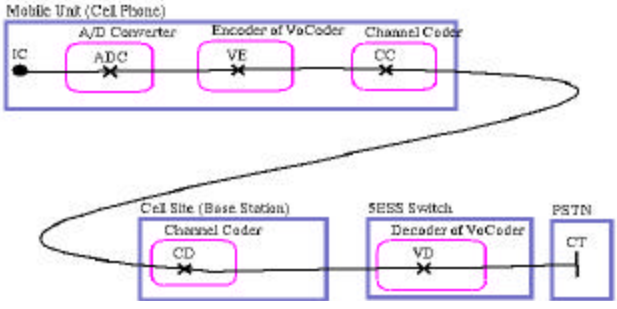


Figure 7: UCM model of another way of binding

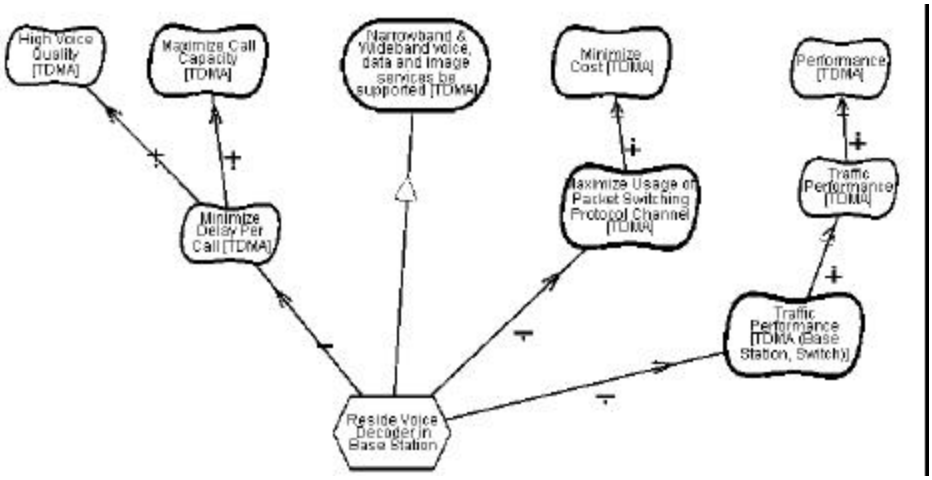


Figure 6: Refined GRL model with one design solution and more non-functional

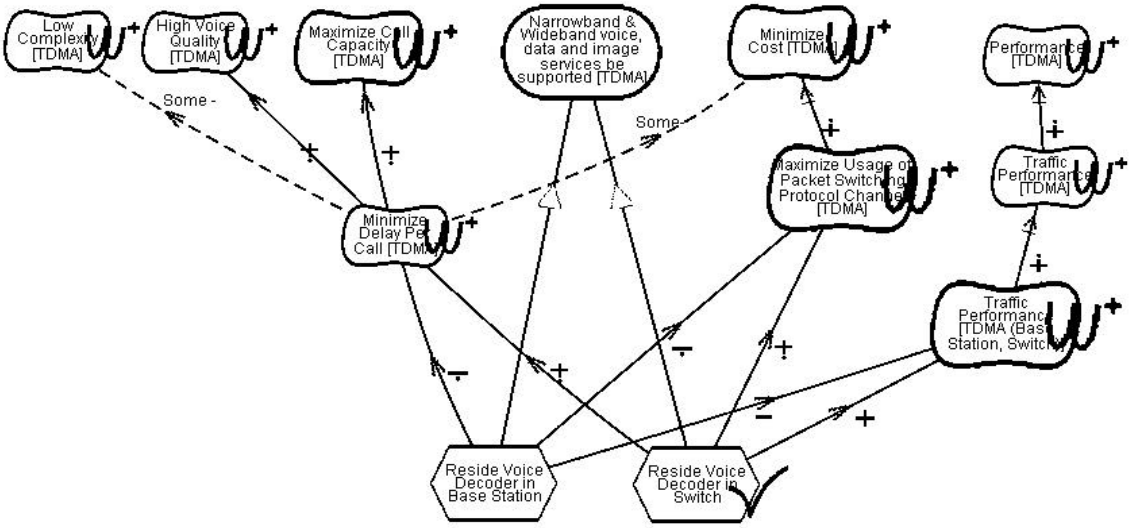


Figure 8: GRL model evaluating the contribution of the new architecture to NFRs

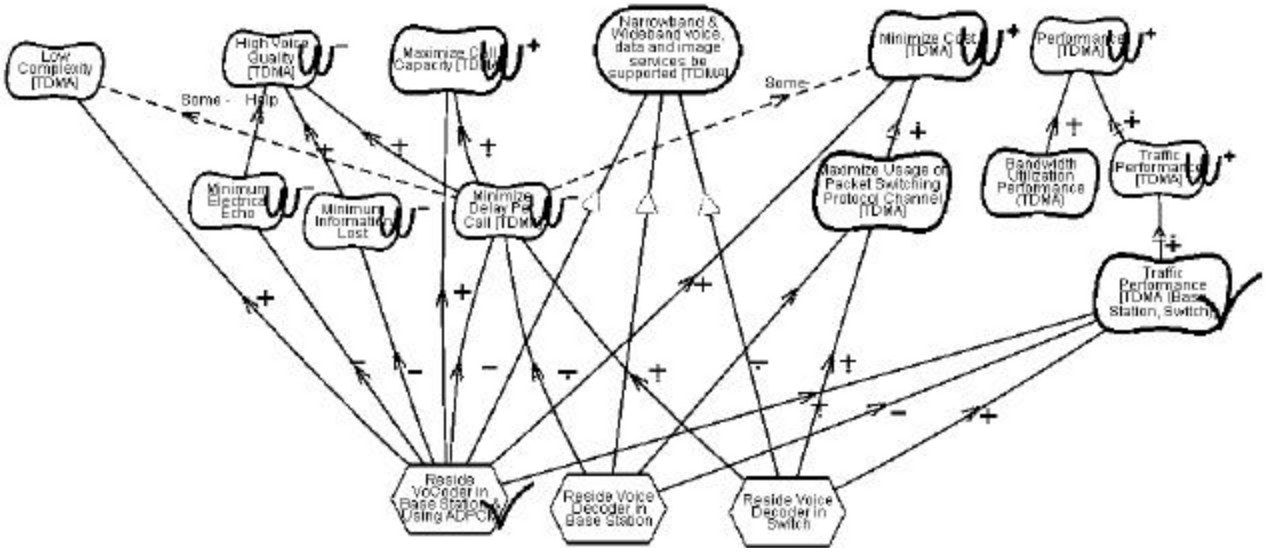


Figure 10: Goal model evaluating the viability of solution 3

architecture to improve the capacity of the TDMA cellular telecommunications system. In this design, the Decoder of voice Coder is relocated into the switch instead of the base station, then for each call the base station would transmit an 8-kb/s signal – rather than a 64-kb/s signal to the switch. In such a system, a theoretical maximum of 8× capacity improvement is possible.

Step 6: GRL Model – Contributions of the new architecture to the non-functional requirements. The GRL model (in Figure 8) shows that the new TDMA architecture with voice coder relocated in the switch weakly satisficed the requirements on improving the capacity, quality, performance, though at the same time the cost and complexity are negatively influenced. To minimize call delay somehow increased the complexity and cost of the architecture (represented in Figure 7 with correlation links). Compare the two architectures, if a cell site supported x calls, the previous architecture would need x DS0s to support those calls. But the Voice Coder relocation architecture would requirement only x/3 DS0s. Given the evaluation result, we judged that the new architecture to be an acceptable design.

GRL supports the evaluation of the satisficing of softgoal with a qualitative labeling procedure. The label of high-level model is propagated from the label of low level nodes, and the contribution from these nodes.

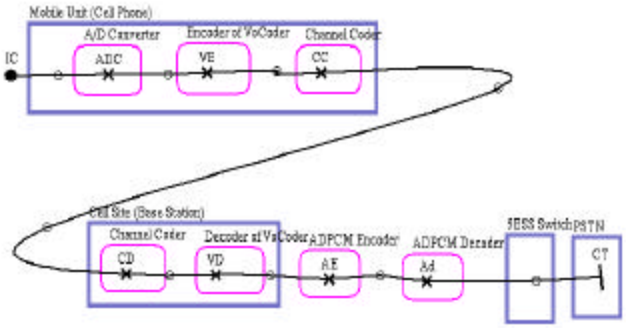


Figure 9: UCM model of solution 3: new responsibilities and functional units added

However, before putting this relocating solution into practice, other possible solutions should also be. The following is one possible solution without relocating the Decoder of Voice Coder.

Step 7: UCM Model – In Figure 9, by adding new functional units without changing the location of Decoder of Voice Coder, a simplest solution is described. For increasing call capacity, 32-kb/s adaptive differential pulse code modulation (ADPCM) equipment is used with voice decoder still in the base station.

Step 8: GRL Model – Evaluation of new architecture according to the non-functional requirements, and compare to other options. The GRL model in Figure 10 shows that this simplest solution weakly satisficed the requirements on improving the capacity, performance, low cost and low complexity. However, voice quality is seriously eroded by the electrical echo, the delay for the extra cycle of speech coding, and the information lost produced in this kind of architecture. While user puts voice quality in a lower priority, this architecture could also be an acceptable choice.

Having analyzed the benefits and tradeoffs of these architectures, we could see that UCM is a natural counterpart to GRL in the process from requirement to high-level design, because it provides the concrete model of each design alternative. Based on the architectural features in this model, new non-functional requirements of concern could be detected and added into the GRL model. At the same time, in the GRL model, new means to achieve the functional requirement could always be explored and be embodied in UCM model.

In the case study above, the UCM model are rather simplistic because we have only tackled the highest level of architectural design in the wireless telecommunication protocol. As we go down to the enough detailed design, a UCM model could be fairly complex, and more modelling constructs could be used. Figure 11 (From [1]) is a root map of a mobile system, it illustrates the "big picture" of a simplified mobile wireless communication system. As shown in this graph, stubs are used to hide details of certain sections of a scenario, e.g., the mobility management functions (MM stub), the communication management functions (CM stub), the handoff procedures (HP stub) and handoff failure actions (HFA stub).

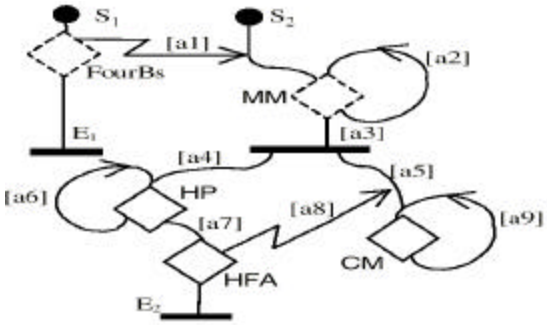


Figure 11: The Mobile system Root Map[1];

A plug-in gives more detail for the stubs. For the limitation of space we won't present all of the plug-ins as well as explain the details of each responsibility. However, one thing need to be notified is, for each stub (especially a static stub), there could be more than one ways to refine the plug-ins. This is a powerful construct to form new design alternatives by integrating possible designs of various parts of the system.

Figure 12 depicts an integrated scenario of establishing a call between the originating and the terminating parties. There could be other possible designs, but we won't investigate for the limitation of space. Components in

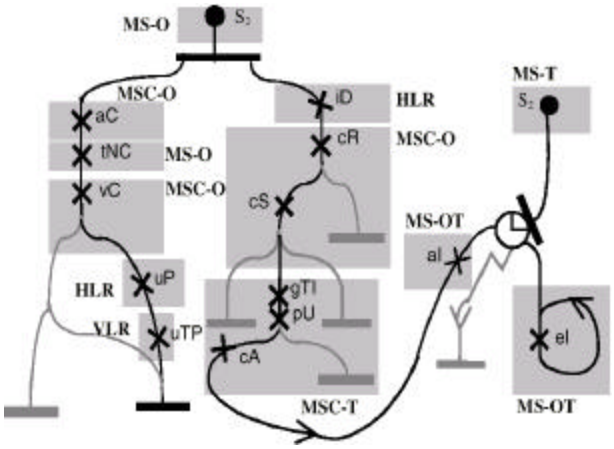


Figure 12: Integration of Scenario Fragments [1]

Figure 12 include: Originating Mobile Station (MS-O), Originating Mobile switching Center (MSC-O), Home Location Register (HLR), Visitor Location Register (VLR), Terminating Mobile Station (MS-T), Terminating Mobile switching Center (MSC-T), Originating and Terminating Mobile Stations (MS-OT).

Although we used a telecommunication system architecture example, the approach is applicable to allocation of responsibility in software systems in general, where there are usually conflicting goals and tradeoffs.

5. Discussions and related works

As existing scenario-based approaches are serving different purposes, using different representational features, and having different analysis capabilities, the concept of scenario needs to be differentiated according to these contexts.

In Krutchen's 4+1 model of software architecture [7], scenarios are used to show connections across other views such as logical view, process view, physical view and development view. The use of a multiple view model of architecture allows to address separately the concerns of the various stakeholders of the architecture. However, with an architecture model composed of several separate views it is not easy to keep a coherent track of the incremental design process. As UCM shows the behavioral and structural aspects together as one view, it is good for showing incremental elaboration of the design.

The Software Architecture Analysis Method (SAAM) [5, 6] is a scenario-based method for evaluating architectures. It provides a means to characterize how well a particular architectural design responds to the demands placed on it by a particular set of scenarios. Based on the notion of context-based evaluation of quality attributes, their method adopts scenarios as the descriptive means of specifying and evaluating quality attributes. For example, to evaluate the modifiability of a user interface architecture Serpent, two scenarios are considered, one is "changing the windows system/toolkit", and the other is "adding a single option to a menu". The similarities between this paper and SAAM include: both works concerns the quality of architecture, and both use scenarios to describe architectures. However, there are obvious differences: SAAM scenarios are use-oriented scenarios, which are designed specifically to evaluate certain quaility attributes of architecture. In GRL vs. UCM, scenarios are more design-oriented, which is the refinements of system requirements. The quality of the architectures corresponding to these scenario are judged based on expert knowledge rather than simulations or tests as in SAAM.

The combined use of goals and scenarios have been explored within RE, primarily for eliciting, validating and documenting software requirements. Van Lamsweerde and Willement studied the use of scenarios for requirement elicitation and explored the process of inferring formal specifications of goals and requirements from scenario descriptions in [8]. Though they thought goal elaboration and scenario elaboration are intertwined processes, their work regarding scenarios in [8] mainly focuses on the goal elicitation. Our emphasis happens to be on the other way around, i.e., how to use goal model (especially NFRs) to direct scenario –based architectural design. The fundamental point is that both the goal-oriented modeling in GRL and the scenario-based modeling in UCM run through requirement process to architectural design, so as their interactions.

In the CREWS project, Collete Rolland et al. have looked into the coupling of goal and scenario in RE with CREWS-L'Ecritoire approach [10]. In CREWS-L'Ecritoire, Scenarios are used as a means to elicit requirements/goals of the system-to-be. Their method is semi-formal. Both goals and scenarios are represented with structured textual prose. The coupling of goal and scenario could be considered as a "tight" coupling, as goals and scenarios are structured into <Goal, Scenario> pairs, which are called "requirement chunks". Their work focuses mainly on the elicitation of functional requirements/goals.

In UCM-GRL, both graphical representations and textual descriptions (in natural language and XML format) for goal model and scenario model are provided. The semi-formal graphical notations are intended to be used during the early stages of architectural design, to help explore and prune the space of design more alternatives. They are to be supplied by for formal notations and analyses in subsequent stages. The current coupling of goal and scenario is loose, as goal models and scenario are all

maintaining their local completeness, and one scenario may refer to more than one goal, and vice versa. There are no rigid constraints on the requirement process. That is, the goal model and scenario model could be developed in parallel simultaneously, they interact whenever there are design decisions need to be traded off, or new design alternatives need to be sought, or new business goals, non-functional requirements are discovered…. Both functional and non-functional requirements are considered, and perhaps even more attentions are devoted to non-functional requirements. The modelling process involves both requirements engineering activities and high-level architecture design.

6. Conclusions and future works

In summary, goal-orientation and scenario-orientation compensate to each other not only in requirement engineering but also during the incremental architectural design process. The combined use of GRL and UCM enables the description of both functional and non-functional requirements, both abstract requirements and concrete system architectural models, both intentional strategic design rationales, and non-intentional details of concurrent, temporal features of the future system.

In the future, we hope to look into create visualized the connections between GRL and UCM to support a more formal combination of the two notations. Thus, the mapping and interacting between the two kinds of models would not rely so much on the human behaviors how they are used.

Another direction would be the accumulation of domain knowledge as well as software design knowledge represented in GRL and UCM. We would say that GRL and UCM are actually the container of knowledge, and it is the knowledge that can be reused, and to guide the future design process.

7. Acknowledgements

The work of this paper is motivated by an original submission to ITU-T study group 10 on the topic of User Requirements Notation (URN). The kind cooperation of people from Mitel Networks, Nortel Networks and other institutions is gratefully acknowledged.

8. References

[1] Andrade, R. and Logripo, L. Reusability at the Early Development Stages of Mobile Wireless Communication Systems. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2000)*, 12, Computer Science and Engineering: Part I, July 2000. Orlando, Florida, 11-16.

[2] Amyot, D. Use Case Maps Quick Tutorial Version 1.0. On-line at: http://www.usecasemaps.org/pub/UCMtutorial/UCMtutorial.pdf.

[3] Buhr, R.J.A. and Casselman, R.S. Use Case Maps for Object Oriented Systems, Prentice-Hall, USA, 1995.

[4] Chung, L., Nixon, B.A., Yu, E.and Mylopoulos, J. Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, 2000.

[5] Kazman, R. Using Scenarios in Architecture Evaluations. *SEI Interactive*, June 1999. On-line at http://interactive.sei.cmu.edu/Columns/The_Architect/1999/June/Architect.jun99.htm

[6] Kazman, R., Bass, L., Abowd, G. and Webb, M. SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proceedings of the 16th International Conference on Software Engineering*. May 1994. Sorrento, Italy. 81-90.

[7] Kruchten, P. The 4+1 view Model of Software Architecture. *IEEE Software*, 12, 6 (November 1995). 42-50.

[8] Lamsweerde, A.V., Willemet, L. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering*, Special Issue on Scenario Management, December 1998.

[9] Lee, A.Y. and Bodnar, B.L. Architecture and Performance Analysis of Packet-Based Mobile Switching Center-to-Base Station Traffic Communications for TDMA. *Bell Labs Journal*. Summer 1997. 46-56.

[10] Rolland, C. , Grosz, G. and Kla, R. Experience With Goal-Scenario Coupling In Requirements Engineering. In *Proceedings of the IEEE International Symposium on Requirements Engineering* 1998. June 1999. Limerick, Ireland.

[11] Yu, E. and Mylopoulos, J. Why Goal-Oriented Requirements Engineering. In *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality*. June 1998, Pisa, Italy. E. Dubois, A.L. Opdahl, K. Pohl, eds. Presses Universitaires de Namur, 1998. pp. 15-22.

Weaving the Software Development Process Between Requirements and Architectures

Bashar Nuseibeh
Computing Department
The Open University
Walton Hall
Milton Keynes MK7 6AA, U.K.
Email:B.A.Nuseibeh@open.ac.uk

ABSTRACT
This position paper argues for the concurrent, iterative development of requirements and architectures during the development of software systems. It presents the "Twin Peaks" model – a partial and simplified version of the spiral model – that illustrates the distinct, yet intertwined activities of requirements engineering and architectural design. The paper suggests that the use of various kinds of patterns – of requirements, architectures, and designs – may provide a way to increase software development productivity and stakeholder satisfaction in this setting.

1 INTRODUCTION

There are compelling economic arguments why an early understanding of stakeholders' requirements leads to systems that more closely meet these stakeholders' expectations. There are equally compelling arguments why an early understanding and construction of a software system architecture provides a basis for discovering further system requirements and constraints, for evaluating a system's technical feasibility, and for evaluating alternative design solutions.

Many software development organisations often make a choice between alternatives starting points – requirements or architectures – and this invariably results in a subsequent waterfall-like development process. Such a process inevitably leads to artificially frozen requirements documents – frozen in order to proceed with the next step in the development life cycle; or leads to systems artificially constrained by their architecture that, in turn, constrain their users and handicap their developers by resisting inevitable and desirable changes in requirements.

There is some consensus that a "spiral" (Boehm 1988) life cycle model addresses many of the drawbacks of a waterfall model, and consequently addresses the need for an incremental development process in which project requirements and funding may be unstable, and in which project risks change and thus need to be evaluated repeatedly. This article suggests that a finer-grain spiral development life cycle is needed, to reflect both the realities and necessities of modern software development – a life cycle that acknowledges the need to develop software architectures that are stable, yet adaptable, in the presence of changing requirements. The cornerstone of such a process is that a system's requirements and its architecture are developed concurrently, that they are "inevitably intertwined" (Swartout & Balzer 1982), and that their development is interleaved.

2 TWIN PEAKS: A CONCURRENT, SPIRAL DEVELOPMENT PROCESS

Figure-1 suggests a partial development model that highlights the concurrent, iterative process of producing progressively more detailed requirements and design specifications. We informally call this model *Twin Peaks* to emphasize the equal status we give the specification of requirements and architectures[1]. We could have also used the more general term *design* rather than *architecture*, as we consider our model to be as applicable to the development of detailed design specifications as it is to architectural design specifications. However, from a project management perspective, the abstraction provided by architectures is sufficient for our purposes.

A shorter, heavily edited version of this paper entitled "Weaving Together Requirements and Architectures" appeared in IEEE Computer, *34(3):115-117, March 2001.*

[1] The model is an adaptation of one first published in P. Ward and S. Mellor's *Structured development for real-time systems* (Volume 1: Introduction and tools, Prentice Hall, 1985), and subsequently adapted by Andrew Vickers in his student lecture notes at the University of York (UK).

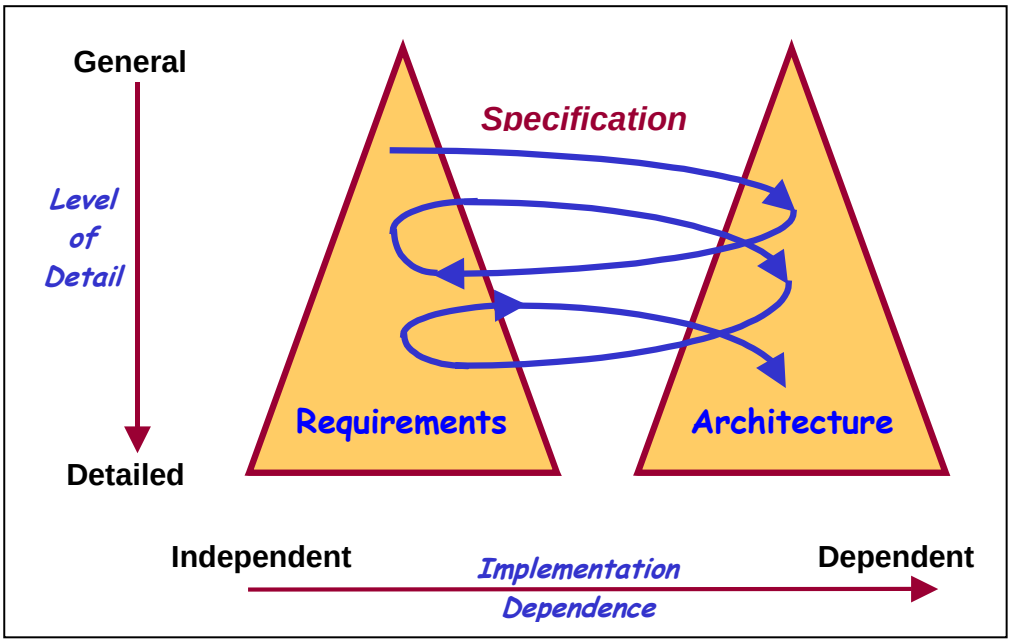


FIGURE-1: TWIN PEAKS – A MODEL OF THE CONCURRENT DEVELOPMENT OF PROGRESSIVELY MORE DETAILED REQUIREMENTS AND ARCHITECTURAL (DESIGN) SPECIFICATIONS.

Our experience on a number of industrial projects support this development life cycle model. Except for very well defined problem domains and strict contractual procedures, most software development projects address issues of requirements specification and design more or less concurrently, and rightly so. At worst, initial requirements gathering is quickly followed by the development or selection of one or more candidate architectures of a system. In many cases, candidate architectures are even provided as part of the requirements. Indeed, candidate architectures may constrain what requirements can be met, and of course the choice of requirements impacts the architecture selected or developed.

As the development process proceeds (and we focus here on specification development in particular), the requirements are elaborated as is the software system architecture. However, although both requirements and architecture are developed concurrently, their distinct content is preserved; that is, the activities of problem structuring and specification (requirements) are separated from (but related to) solution structuring and specification (architecture). This separation, while recognized as important, is often difficult to achieve in alternative life-cycles, since the artificial ordering of requirements and design steps compels developers to focus on either of the two at any one time.

The Twin Peaks model addresses the three management concerns identified by Barry Boehm in an earlier article (Boehm 2000), namely IKIWISI (I'll Know It When I See It), COTS (Commercial off-the-shelf) software, and rapid change.

IKIWISI. Requirements often "emerge" only after significant analysis of models or prototypes of a system has taken place, and users have been given an opportunity to view and provide feedback on the models or prototypes. The Twin Peaks model explicitly allows early exploration of the solution space, thereby allowing incremental development and the consequent management of risk.

COTS. Increasingly, software development is actually a process of identifying and selecting existing software packages (Finkelstein *et al.* 1996, Maiden & Ncube 1998). Package selection requires the identification of desirable requirements (often expressed as features or services), and a matching of these to what is commercially available. Adopting the Twin Peaks model allows rapid and incremental requirements identification and architectural matching. This can be invaluable in quickly narrowing down the choices available, or perhaps making key architectural choices to accommodate existing COTS solutions.

Rapid Change. Managing change continues to be a fundamental problem in software development and project management. *Rapid* change exacebates this problem. We believe that the Twin Peaks model focuses on finer-grain development and is therefore more receptive to changes as they occur. However, analysing the impact of change is still

a difficult task. Key to addressing this task is identifying the *core* requirements of a software system. Such requirements express those stakeholders' goals that are likely to persist for the longest period of time, and that are likely to lead to a software architecture that is stable in the face of changes in other requirements. Core requirements may also be those whose impact on a software architecture is so substantial that they have to be frozen in order to avoid the excessive (unacceptable) cost of changing them.

3 BUILDING MODULAR SOFTWARE INCREMENTALLY

Developing software systems in the context of IKIWISI, COTS, and rapid change suggests that we need to consider different processes of development. IKIWISI means that we need to start design and implementation much earlier than usual; COTS means that we need to consider reuse at the much earlier stage of requirements specification; and rapid change means that we have to be able to do all this much more quickly in order to be competitive.

There is increasing recognition that building systems in terms of components with well defined interfaces offers opportunities for more effective reuse and maintenance. It is not always clear, however, how component-based development approaches fit into the development process. One way is to consider, concurrently, "patterns" of requirements, architectures and designs. The software design community has already identified a number of "design patterns" (Gamma *et al.* 1996) that can be used to express a range of implementations. The software architectures community has identified "architectural styles" (Shaw & Garlan 1996) that are suited to meeting various global requirements. And, the requirements engineering community has promoted the use of "problem frames" (Jackson 2001) or "analysis patterns" (Fowler 1996) to identify classes of problems for which there are existing, known solutions. This begs the question: what are the relationships between these different kinds of patterns?

Figure-2 suggests that such patterns of requirements, designs, and architectures can be treated as a resource for component-based development. Indeed, the figure suggests that – in line with the Twin Peaks model – the "starting point" of development may be requirements, design, or architecture. If a given architecture is fixed, this impacts on the kinds of problems that can be addressed by that architecture, and the kinds of designs that may be possible. If the requirements are more rigid, then the candidate architectures and design choices are also limited. If a certain design is chosen, then both the architectures in which this design fits and the problems that are addressed by this design are also limited.

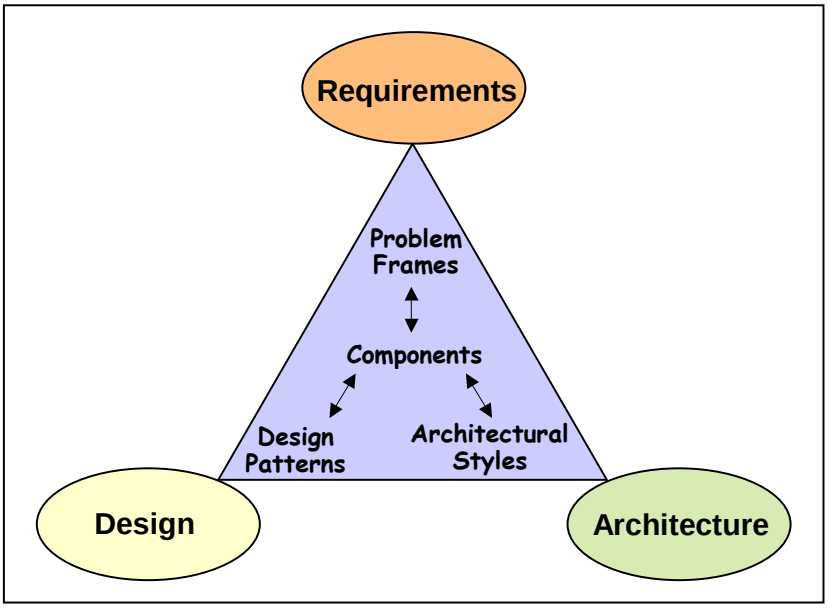


FIGURE-2: PART OF THE SOFTWARE DEVELOPMENT TERRAIN, WITH REQUIREMENTS, ARCHITECTURE, AND DESIGN RECEIVING SIMILAR ATTENTION. PATTERNS OF EACH HAVE AN IMPACT ON THE KIND OF SYSTEM (COMPONENTS) DEVELOPED, AND THE RELATIONSHIP BETWEEN THEM IS A KEY DETERMINANT OF THE KIND OF THE DEVELOPMENT PROCESS ADOPTED.

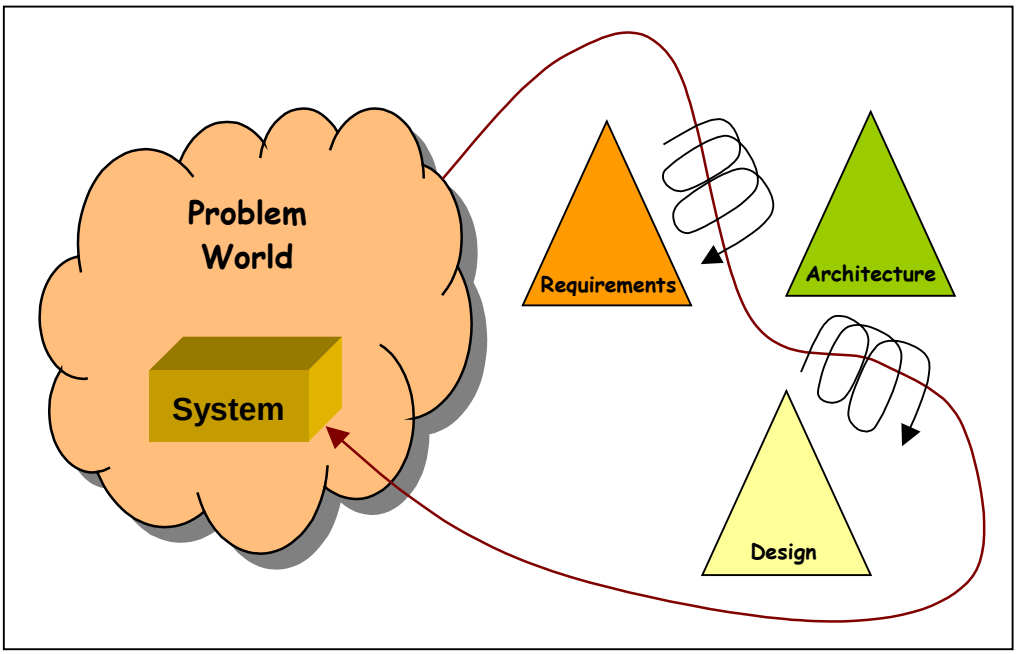


FIGURE-3: WEAVING THE SOFTWARE DEVELOPMENT PROCESS FROM PROBLEM TO SOLUTION. THE TRIANGULAR ICONS INDICATE ARTEFACTS THAT GROW (FROM TOP TO BOTTOM) AS THE PROCESS PROCEEDS. THERE IS AN IMPLICIT GLOBAL FEEDBACK LOOP AS WELL, IN WHICH THE SYSTEM INSTALLED IN THE WORLD CHANGES THE WORLD, AND POTENTIALLY NECESSITATES ANOTHER WEAVING JOURNEY.

From a requirements engineering perspective, it is essential that a satisfactory problem structuring is achieved as early as possible. A vehicle for achieving such a structuring is Jackson's problem frames[2]. However, given that existing architectures may influence how problems are structured, it may be necessary – as Jackson suggests – to "reverse engineer" some problem frames from existing architectural designs.

4 PROJECT MANAGEMENT: WEAVING THE DEVELOPMENT PROCESS

The Twin Peaks model of software development shares much in common with Extreme Programming (XP) (Beck 1999), such as the goal of exploring implementation possibilities early and iteratively. The focus of Twin Peaks, however, is very different from, but perhaps complementary to, the XP model in that it focuses on the front-end activities of the software development life cycle; that is, on requirements and architectures. This potentially addresses some of the issues of scale that are often claimed as weaknesses of XP. Early understanding of requirements and the choice of architecture are key to managing large scale systems and projects. XP focuses on producing code – sometimes at the expense of the "wider picture" of requirements and architecture. Of course, the focus on requirements and architectures in itself is not sufficient to achieve scalability. Modularity and iteration are also crucial. Twin Peaks is inherently iterative, and combined with the use of patterns can facilitate incremental development of large scale systems more quickly, using tried and tested components derived from well-understood patterns.

The resultant overall software development process inevitably takes a much more complex path from problem to solution. Figure-3 is a hugely simplified illustration that tries to convey the winding route in which development proceeds as requirements, architectures, and designs are elaborated iteratively and often concurrently.

5 CONCLUSIONS

With the many advances in software development in recent years, it is perhaps appropriate to re-visit the relationships between requirements and design. Although the conceptual differences between requirements and design are now much better understood and articulated (Jackson 1995), the *process* of moving between the problem world and the solution world is much less so (Goedicke & Nuseibeh 1996). Researchers and practitioners are struggling to develop processes that allow rapid development in a competitive market, combined with the improved analysis and planning that is necessary to produce high quality systems within tight time and budget constraints.

2 Recall that a problem frame defines the shape of a problem for which there is a known solution.

We have suggested that a more robust and realistic development process is one that allows both requirements engineers and system architects to work concurrently and iteratively to describe the artefacts they wish to produce. In this way, problems are better understood through consideration of architectural constraints, and architectures can be developed and adapted based on requirements.

We have exposed only the tip of the iceberg. Many difficult questions remain unanswered:

- What software architectures (or architectural styles) are stable in the presence of changing requirements, and how do we select them?
- What classes of requirements are more stable than others, and how do we identify them?
- What kinds of changes are systems likely to experience in their lifetime, and how do we manage requirements and architectures (and their development processes) in order to manage the impact of these changes?

The answers to these questions will have significant impact on the way software is developed and projects are managed. Particular impact will be felt in some key emerging development contexts:

- *Product lines and product families* – where there is a need for stable architectures that tolerate changing requirements.
- *COTS systems* – where there is a need to identify and match existing units of architectures to requirements (as opposed to developing system requirements from scratch).
- *Legacy systems* – where there is a need to incorporate existing system constraints into requirements specifications.

For software systems that need to be developed quickly, with progressively shorter times-to-market as a key requirement, development processes that facilitate fast, incremental delivery are essential. The Twin Peaks model that we have presented in this article represents much of the existing current state-of-the-practice in software development. It is also based on accepted research into evolutionary development as embodied in Boehm's spiral model (Boehm 1988). What is missing, however, is a more explicit recognition by the software development community that such a model represents acceptable practice.

Processes that embody some of the characteristics of the Twin Peaks, are the first steps in tackling the need for architectural stability in the face of inevitable requirements volatility.

ACKNOWLEDGEMENTS

Many of the ideas in this article are the result of numerous discussions with Jeff Kramer, Jeff Magee, and Alessandra Russo at Imperial College, David Bush at the UK National Air Traffic Services, and Anthony Finkelstein at University College London. I am also grateful to Leonor Barroca, Pat Hall and Michael Jackson at The Open University for comments on an earlier draft of the article, and to the UK EPSRC for financial support (GR/L55964 & GR/M38582).

6 REFERENCES

[1] B. Boehm (1988), "A Spiral Model of Software Development and Enhancement", *Computer*, 21(5):61-72, IEEE CS Press.

[2] B. Boehm (2000), "Requirements that Handle IKIWISI, COTS, and Rapid Change", *Computer*, 33(7):99–102, July 2000, IEEE CS Press.

[3] K. Beck (1999), *Extreme Programming Explained: Embracing Change*, Addison-Wesley.

[4] A. Finkelstein, M. Ryan and G. Spanoudakis (1996), "Software Package Requirements and Procurement", In Proceedings of 8th International Workshop on Software Specification & Design (IWSSD-8), 141-146, Schloss Velen, Germany, 22-23 March 1996, IEEE CS Press.

[5] M. Fowler (1996), *Analysis Patterns*, Addison Wesley.

[6] E. Gamma, R. Helms, R. Johnson, J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[7] M. Goedicke & B. Nuseibeh (1996), "The Process Road Between Requirements and Design", *In Proceedings of 2nd World Conference on Integrated Design and Process Technology*, 176-177, Austin, Texas, USA, 1-4 December 1996, SDPS.

[8] M. Jackson (1995), *Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices*, Addison-Wesley.

[9] M. Jackson (2001), *Problem Frames: Analyzing and Structuring Software Development Problems*, Addison-Wesley.

[10] N. Maiden and C. Ncube (1998), "Acquiring Requirements for Commercial Off-the-Shelf Package Selection", *Software*, 15(2):46-56, IEEE CS Press.

[11] M. Shaw and D. Garlan (1996), *Software Architectures*, Prentice-Hall.

[12] W. Swartout and R. Balzer (1982), "On the Inevitable Intertwining of Specification and Implementation", *Communications of the ACM*, 25(7): 438-440.

A Framework for Requirements Engineering for Context-Aware Services

Anthony Finkelstein Andrea Savigni

Department of Computer Science
University College London
Gower Street
London WC1E 6BT
United Kingdom
E-mail: {A.Finkelstein|A.Savigni}@cs.ucl.ac.uk

Abstract

Context-aware services, especially when made available to mobile devices, constitute an interesting but very challenging domain. It poses fundamental problems for both requirements engineering, software architecture, and their relationship. We propose a novel, reflection-based framework for requirements engineering for this class of applications. The framework addresses the key difficulties in this field, such as changing context and changing requirements. We report preliminary work on this framework and suggest future directions.

1. The Rationale

The purpose of this section is to highlight the key problems associated with requirements engineering in the area of context-aware services. In order to properly classify concepts, we will adopt Michael Jackson's terminology, as introduced in [10] and briefly reviewed in Sect. 2.1. Particularly the critical distinction he maintains between the "world" and the "machine". That terminology will be used throughout the paper.

In this paper, by "context-awareness" we mean the ability of a particular service to adapt itself to a changing context. One classical example is mobile commerce (m-commerce) applications, which should run equally well on full-fledged Web browsers running on desktop computers, on graphic Personal Digital Assistants (PDAs), on Wireless Application Protocol (WAP)-enabled mobile phones, and possibly even on low-end mobile phones, maybe using Short Message System (SMS).

Requirements engineering in the area of context-aware services, especially when these are targeted towards mobile devices, poses new and very challenging problems, that can be summarised as *changing context* and *changing requirements*.

A changing context means essentially that one cannot, while analysing requirements, rely on reassuring assumptions about the world. A changing world complicates the machine by orders of magnitude. In the case of context-aware mobile services, changing context may entail:

- changing location. This means not only that the absolute location of a device can change, but also that the relative locations of two devices must be taken into consideration;
- changing bandwidth for networked devices, most often in unpredictable ways;
- changing display characteristics e.g., graphics PDAs, text-only mobile phones, colour vs. monochrome displays, etc.;
- changing usage paradigm. For example, from a user perspective having a full-screen, button-centred PDA is very different from using a scroll-centred mobile phone;
- target platforms unknown in advance. Note that this problem is *not* implied by any of the preceding points. Platforms may be unknown in advance, and the service should anyway be able to dynamically adapt itself to this aspect of the new context. This means of course performing a very hard abstraction job in order to express the common set of characteristics in a general, uniform way.

This very volatile context of course influences requirements. A key distinction, adapted from Axel van Lamsweerde's work (see Sect. 2.2), is made here between *goals* and *requirements*. We define a goal as a fixed objective of the service, whereas a requirement, in our view, is a more

volatile concept that can be influenced by the context. For example, in a m-commerce service, a goal can be "maximise usability of the system", which is a very abstract objective that the system should tend to [6]. By contrast, a requirement can be: "the display must show both the current state of the shopping basket and a set of available options". This requirement of course makes sense only if the display is large enough.

One more, fundamental issue related to such services is that they usually belong to the "new economy". This means in general that these systems have an extremely short time-to-market, which in turn means that traditional, heavyweight methodologies – such as the Rational Unified Process (RUP) [5] – are not applicable.

For all these reasons, we argue that a new approach is needed to tackle this kind of services. Such an approach is the subject of this paper, and will be described as follows. Section 2 will provide the reader with some background information. Section 3 outlines the reflective approach that will be used throughout the work. Section 4 explains the framework itself, while Sect. 5 sets out some of the key challenges it poses. Finally, Sect. 6 sketches some possible ways to move towards an implementation of the framework.

2. Background

The goal of this section is to give a very brief overview of the two main influences behind this paper, namely Michael Jacksons's "world and machine" work [10], and Axel van Lamsweerde's "Kaos" [6].

2.1. The World and the Machine

[10] represents a cornerstone in understanding the relationships between a software artifact and the surrounding world. Jackson identifies four facets of relationships between the world and the machine:

- the modelling facet, in which the machine simulates the world;
- the interface facet, where the world touches the machine physically;
- the engineering facet, where the machine controls the world;
- the problem facet, where the shape of the world and of the problem influences the shape of the machine and of the solution.

The discussion of the engineering facet turned out to be particularly useful to us, and particularly the distinction between requirements, specifications, and programs. Requirements are concerned solely with the world, programs are concerned solely with the machine, specifications are the bridge between the two. Section 4 will use these concepts in working out the boundaries between world and machine within our framework.

2.2. Goal-oriented Requirements Engineering

The seminal works by Yue [17] and van Lamsweerde [6] opened a new direction in requirements engineering: the *goal-oriented* approach. The key achievement of this new approach is that it makes explicit the *why* of requirements. Quoting van Lamsweerde, "[before goal-oriented requirements engineering] the requirements on data and operations were just there; one could not capture *why* they were there and whether they were sufficient" [16].

van Lamsweerde's goal-oriented requirements engineering approach provides for three levels of modelling:

- the meta level, that refers to *domain-independent abstractions*. This model contains concepts such as `goal`, `requirement`, `object`, `entity`, and so on;
- the base level, containing domain-dependent concepts, such as `service`, `telephone`, `bandwidth`, etc. The structure of the meta-level model constitutes a meta-level guide on how to conduct a requirements engineering activity. For example, since `goal` and `constraint` are linked by a `operationalisation` link, every concept in the base level that is an instance of a meta-level concept `constraint` must be linked to an instance of a meta-level `goal` by an instance of a meta-level `operationalisation` link;
- the instance level, containing specific instances of the domain-level concepts.

3. The Reflective Approach

"Computational reflection is the activity performed by a computational system when doing computation about its own computation" [11]. A reflective system maintains, *at run-time*, data structures that materialise some aspects of the system itself.

The problem of allowing a program to reason upon, and possibly change, itself is not new, and has been studied extensively especially in the programming languages community. For example, languages such as LISP and Prolog allow programs to be manipulated as data. More recently, so-called "open languages" (such as OpenC++ [4] or OpenJava [14]) allow programmers to influence the translation process, thus actually providing for the definition of new languages.

For our purposes, reflection means that an explicit, runtime representation of system behaviour is maintained, which *reifies* the actual system behaviour in the sense that changes in the latter are materialised in the meta-level description. Similarly, changes in the meta-level description *reflect* back into the underlying system's behaviour. This "closed loop" approach is called *causal connection*. A reflective system is structured into a (potentially unbound) number of logical levels: the *reflective tower* [13]. In practice, there are seldom more than two of them.

Reflective systems are based on two concepts: consistency between internal and external representations of the system, and separation between meta computation and computation. The consistency is guaranteed by causal connection: computations performed in the base level are reified by the meta level, whereas changes in the meta level reflect back into the base-level. The separation between meta computation (i.e., computation whose domain [11] is the base-level) and computation (whose domain is the world) is essential in order to achieve *transparency*: new functionality can be added to an existing system in a transparent way i.e., without the existing system noticing. This is especially true of functionality implementing non-functional requirements, such as fault-tolerance and security.

Why do we regard a reflective approach as such a fundamental issue? First of all, let us make one point clear: reflection, at least in our view, is a *mechanism*, not a goal. More precisely, it is a mechanism for manipulating meta data in a clean and consistent way. Now, reflection is key in this field because manipulating meta data is essential in this context of highly-dynamic services, as these must be able to dynamically adapt themselves to changing context and changing requirements.

4. The Framework

Figure 1 shows the key concepts of the proposed framework.

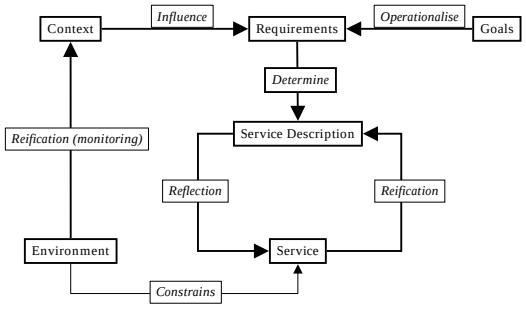


Figure 1. The overall framework.

The rest of this section is devoted to a detailed explanation of the framework constituents. This explanation will follow a precise path that moves from the outside inward i.e., from the outer world towards the boundaries with the machine, and finally inside the machine itself. Therefore, we will start from what is available in the world: goals and environment. We will operationalise goals into requirements, and represent environment information into a context; all of this still belongs in the world. Later we will move from requirements and context towards a service description, which is the bridge between the world and the machine (what Michael Jackson calls "specification"). Eventually we move inside the machine with the notion of a service. Note that throughout the paper we will stick to the notion of the machine as pure software; in other words, we will consider devices (PDAs, mobile phones, etc.) as part of the world.

4.1. Goal

A goal is an objective the system should achieve through cooperation of agents in the software-to-be and in the environment [6]. In our view goals are *immutable* i.e., they do not change with the changing context. They represent the ultimate objective the service is meant to achieve. Changing the goals would mean changing the service itself. Along the lines of [6], a goal is not immediately achievable through actions performed by one or more agents; in other words, a goal is a somewhat abstract and long-term objective.

4.2. Environment

By "environment" we mean whatever in the world provides a surrounding in which the machine is supposed to operate. Taking the environment into account is crucial because it strongly influences the behaviour of the machine. Recall the example of the m-commerce service. In this case the environment comprises such things as bandwidth, location (absolute and relative), service availability, characteristics of the device, and many more issues.

An alternative definition of environment might be: "whatever over which we have no control". If the bandwidth is low, the connection is erratic, the PDA's display is small, the person carrying the mobile phone is driving on a mountain road with many tunnels, this is something that cannot be solved by software. The job of a software engineer can be summarised as a struggle *towards* the goal *despite* the environment; all we can do with the environment is know it and describe it in the best possible way, but we cannot change it.

4.3. Context

Context is defined as the reification of the environment. Note that in this case there is no reflection whatsoever (i.e.,

no downwards arrow) because, as explained in the previous section, the environment is not modifiable. A context thus provides a manageable, easily manipulatable description of the environment. Most important, such description is continuously, dynamically updated to take into account the fact that the environment also continuously changes.

4.4. Requirement

A requirement represents one of the possible ways of achieving a goal. A requirement operationalises a goal, in that it represents a more concrete, short-term objective that is directly achievable through actions performed by one or more agents. One key assumption that we make is that *requirements can change during system execution*, which differentiates them from goals. In fact, due to a changing environment, the context may change in such a way that the operationalisation of the goals is no longer valid. This calls for monitoring of the context with respect to the goals: changes in the context may yield the necessity for changes in the requirements.

In very informal terms, one may say that requirements are a trade-off between the noble goals and the actual reality. For example, the goal of an m-commerce service might be to provide for a highly interactive user experience. Given this goal, if the context is favorable (e.g., high bandwidth, large colour display, Java Virtual Machine implementation available on the PDA) a requirement might be "use a colorful Java applet to represent the state of the shopping basket", whereas if the connection is slow or there is no JVM available, the requirement may be mitigated into "use a 16-colour animated gif".

4.5. Service Description

A service description is the meta-level representation of the actual, real-world service. As such, it is obviously influenced by the requirements, hence the `Determine` box in Fig. 1. A service description might seem redundant, as one may think of going directly from requirements to service. Why is an intermediate component needed? The answer lies in the reflective approach and in the need for continuously monitoring the service. In fact, the service can be influenced by the environment, and can therefore change in unpredictable ways. These changes can lead to inconsistencies between the service and the requirements. This calls for monitoring of the former with respect to the latter. A service description is a meta-level description of a service. If a suitable formalism is devised for this description, the latter can easily be compared against the requirements in order to establish whether a runtime violation [8] has occurred.

Now, suppose such a violation is detected. We argue that the "reflective way" is a clean and consistent manner of performing run-time changes to the underlying level (which is, at last, the actual system as perceived by the user). This approach consists in manipulating the service description in order to reconcile the service with the requirements. The causal connection, in particular the downwards link (reflection) provides for the consistency between the service description and the service itself. Architectural reflective techniques can be employed to that aim [2, 3, 15].

Since the service description describes the behaviour of the service, it can be regarded as a system specification in the sense used in [10]. Thus, it serves as the bridge between the world and the machine.

4.6. Service

Finally, the service is the heart of the machine. It provides the actual behaviour as perceived by the user. It is worth pointing out that, even though it is only this service that actually interacts with the user, it is the last link in the chain described above; in other words, the actual value delivered to the user is not the service alone, but also the whole hidden reflective infrastructure.

It is also worth pointing out that, apart from goals that are specified off-line and never changed (recall, changing goals means changing what the service provides, and this means at the very least pulling the service down), all the remaining items appearing in Fig. 1 have a run-time image, as emphasised in Fig. 2, where the run-time components are greyed. Finally, Fig. 3 emphasises (in grey) the meta-level

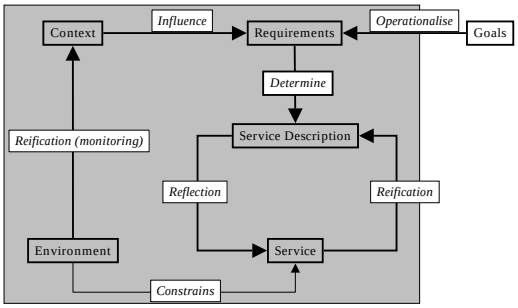


Figure 2. The run-time components.

components i.e., all those components that, even having a run-time image, are not directly visible to the end user.

5. The Challenges

The problems examined in the previous section represent a formidable challenge for any software engineer. More precisely, the following points must be addressed.

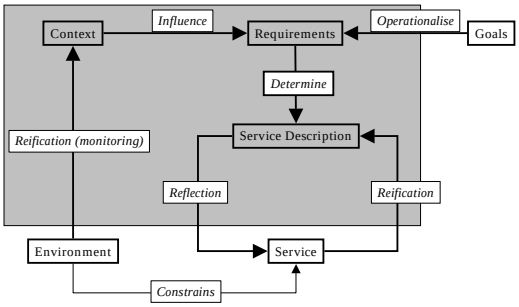


Figure 3. The meta-level components.

5.1. Representing context information at run-time

One of the key issues in these systems is that context is continuously changing. Therefore, requirements, in the first place, and system behaviour must adapt themselves to the changing context. In order for this to be feasible, the context (and its changes) must be represented at run-time. This representation must take place in a way that is both readily understandable by humans and easily manipulatable by machines.

5.2. Bringing requirements information to run-time

In order to be able to perform run-time service description monitoring against requirements, these must be readily accessible at run-time (see Sect. 4.5) [9].

In addition, as explained in Sect. 4.4, requirements typically change over time, so this representation must not simply be a read-only view, but must rather be an evolvable one.

5.3. Bringing architecture information to run-time

This is admittedly one of the most controversial points. It is widely accepted in the software engineering community that a suitable software architecture design phase should always precede the actual implementation. However, in most cases all information about system architecture is lost in the running system [15]. In other words, a running system *implements* a specification; however, this specification is scattered throughout the code, and no explicit representation of it exists at run-time.

6. Implementation Issues

6.1. Describing the Meta Levels

One key question to be answered is: "How to describe the meta levels in an easy and powerful way?" One particularly promising way is the use of XML for such description. The main reasons behind such a choice are sketched in the sequel:

- XML is a world standard. A description implies a formalism, so why not choose a standard one?
- no need to build custom parsers. A number of products implementing the standard DOM and SAX APIs are widely available, often at no cost;
- a number of standards, APIs, and products are available to easily and efficiently manipulate XML files, first of all XSLT;
- a lot of work has been (and is being) done at UCL in this field; in particular, the work on consistency checking of distributed documents (that yielded `xlinkit` [12]) could prove a very useful starting point in determining whether the runtime system behaviour is still aligned with the requirements.

6.2. Where Does All This Belong?

An interesting question to ask is: Where does all the framework belong? Or, in other words, should every single service take care of this on its own? Can all, or at least some, of the framework be collected in a separate product which can be implemented once and for all and customised at will? If so, which parts are strictly service-dependent and which can be made common?

We do not yet have a definitive answer to these questions. However, our current thought is that it should be possible to provide a service-independent set of mechanisms for representing context in a significant class of context-aware services. The mechanisms by which such a context is populated in any particular case is clearly a matter for the device vendor.

On the service description side, the situation is more complex, and service description schemes drawn from existing middleware frameworks [1, 7] may be the right direction.

Acknowledgments

This work was partially funded by UWA (Ubiquitous Web Applications), a EU-funded, Fifth Framework Programme project that the authors are carrying on in cooperation with a number of academic and industrial partners from six European countries. The work described in this paper is intended as an initial contribution to the project.

We are also grateful to Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo for the fruitful discussions that influenced some of the views in this paper.

References

[1] L. Capra, W. Emmerich, and C. Mascolo. Reflective Middleware Solutions for Context-Aware Applications. Submitted for publication.

[2] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *Proceedings of the 2^{nd} Euromicro Conference on Software Maintenance and Reengineering and 6^{th} Reengineering Forum*, Florence, Italy, March 8-11 1998.

[3] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. In *Proceedings of Automated Software Engineering – ASE'99 14^{th} IEEE International Conference*, pages 263–266, Cocoa Beach, Florida, USA, Oct 12-15 1999.

[4] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA95*, pages 285–299, October 1995.

[5] R. S. Corporation. The Rational Unified Process. `http://www.rational.com/products/rup/`.

[6] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[7] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, April 2000.

[8] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling System Requirements and Runtime Behavior. In *Proceedings of IWSSD'98 - 9^{th} International Workshop on Software Specification and Design*, Isobe, Japan, April 1998. IEEE Computer Society Press.

[9] S. Fickas and M. S. Feather. Requirements Monitoring in Dynamic Environments. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, pages 140–147. IEEE Computer Society Press, 1995.

[10] M. Jackson. The World and the Machine. In *Proceedings of the 17^{th} International Conference on Software Engineering*, pages 283 – 292, Seattle, Washington, USA, April 24 – 28 1995.

[11] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA87, Sigplan Notices*. ACM, October 1987.

[12] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. `xlinkit`: a Consistency Management and Smart Link Generation Service. Technical Report RN/00/66, University College London – Department of Computer Science, December 2000. Submitted for publication.

[13] B. C. Smith. Reflection and Semantics in Lisp. In *Conference Record of the 14^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, USA, January 1984.

[14] M. Tatsubori and S. Chiba. Programming Support of Design Patterns with Compile-time Reflection. In *OOPSLA98 Workshop on Reflective Programming in C++ and Java*, pages 56–60, Vancouver, Canada, 1998.

[15] F. Tisato, A. Savigni, W. Cazzola, and A. Sosio. Architectural Reflection. Realising Software Architectures via Reflective Activities. In *Proceedings of the 2^{nd} Engineering Distributed Objects Workshop (EDO 2000)*, Davis, California, USA, November 2–3 2000. To appear.

[16] A. van Lamsweerde. Requirements Engineering in the Year 00: A Research Perspective. In *Proceedings of ICSE'2000 - 22^{nd} International Conference on Software Engineering*, Limerick, 2000. ACM Press. Invited Paper.

[17] K. Yue. What Does It Mean to Say that a Specification is Complete? In *Proceedings of IWSSD-4 – the Fourth International Workshop on Software Specification and Design*, Monterey, CA, USA, 1987.

Refinement and Evolution Issues in Bridging Requirements and Architecture – The CBSP Approach

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 231
Marina Del Rey, CA 90232, USA
+1 310 578 5350
aegyed@acm.org

Paul Grünbacher
Johannes Kepler University
Systems Engineering and Automation
4040 Linz, Austria
+43 732 2468 8867
pg@sea.uni-linz.ac.at

Nenad Medvidovic
University of Southern California
941 W. 37th Place
Los Angeles, CA 90089-0781, USA
+1 213 740 5579
neno@usc.edu

ABSTRACT
Though acknowledged as being very closely related, requirements engineering and architecture modeling have been pursued largely independently of one another in the past years. The inter-dependencies and constraints between architectural elements and requirements elements are thus not well-understood and subsequently only little guidance is available in bridging requirements and architectures. This paper identifies a number of relevant relationships we have identified in the process of trying to relate a requirements engineering approach with an architecture-centered approach. Our approach, called CBSP (Component-Bus-System, and Properties) provides an intermediate language for representing requirements in an architectural fashion. In this paper, we will present the basics of our CBSP approach but also emphasize the challenges that still need to be resolved.

Keywords
Requirements engineering, software architecture, traceability, evolution, model integration, CBSP

1 INTRODUCTION

Requirements largely describe aspects of the problem to be solved and constraints on the solution. Requirements are derived from the problem domain (e.g., medical informatics, E-commerce, avionics, mobile robotics) and reflect the, sometimes conflicting, interests of a given set of system's stakeholders (customers, users, managers, developers). Requirements deal with concepts such as goals, conflicts (issues), alternatives (options), agreements, [3], and, above all, desired system features and properties (both functional and non-functional).

Architectures, on the other hand, model a solution to the problem described in the requirements. Software architectures provide high-level abstractions for representing the structure, behavior, and key properties of a software system. The terminology and concepts used to describe architectures differ from those used for the requirements. An architecture deals with components, which are the computational and data elements in a software system [10]. The interactions among components are captured within explicit software connectors (or buses) [11]. Components and connectors are composed into specific software system topologies. And, architectures capture and reflect the key desired properties of the system under construction (e.g., reliability, performance, cost) [11].

The relationship between a set of requirements and an effective architecture for a desired system, however, is not readily obvious. This conflicts our need of having requirements engineering and architectural modeling being intertwined and mutually-dependent development activities in order to ensure their complete and consistent treatment (i.e., refinement). In context of requirements, architectural modeling has to satisfy the roles of (1) supporting fast trade-off analyses about requirements' feasibility via the modeling of architectural options, and (2) supporting the modeling of architectural solutions in a manner that reflects functional and non-functional properties of requirements in a form that is more readily refineable to code. In context of architectural modeling, requirements engineering has to define (1) functional and non-functional constraints that affect architectural decisions, and, (2) rationale that defines purpose and goal of architectural solutions.

The existence of conceptual differences between what to do (requirements) versus how to do it (architecture, design, and code) constitutes a gap. This gap has been often observed and frequently documented [9], however, despite massive attention it remains unsolved. It is still a difficult problem on how to transition from requirements to an architecture and vice versa. Some of those many issues involve: How to interpret informal requirements in context of more formal architectures? How to elicit functional and

non-functional aspects out of requirements? How to infer architectural topologies and styles out of constraints imposed by given requirements? How to reason about mismatches among requirements or between requirements and given architectural solutions? How to maintain requirements and architectures interpedently and yet consistently while both are being evolved? And how to do all of the above if hundreds if not thousands of requirements need to be considered?

To address these challenges and others we have developed a light-weight method for identifying key architectural elements and their dependencies based on given requirements. Our method, called the CBSP approach (Component-Bus-System-Property), helps in refining a set of given requirements into potential architectures by applying a taxonomy of architectural dimensions. Input to our method can be a set of (incomplete) requirements captured in textual or formal descriptions and containing rationale. The result of CBSP is an intermediate model that captures "architectural decisions" of requirements in form of an incomplete architecture.

At the current state we applied CBSP in context of EasyWinWin [2,6], a requirements elicitation technique, and C2 [12], an architectural style for highly-distributed systems. However, we believe that it can be applied to other requirements elicitation and architecture-capture approaches. The following discusses the basics of our CBSP approach in context of an example followed by an update of the current state of the approach and needed future work to make our technique more comprehensive.

2 CARGO ROUTER CASE STUDY

We have performed a thorough requirements, architecture, and design modeling exercise to evaluate CBSP in the context of a cargo router application. The *Cargo Route*r system was built to handle the delivery of cargo from delivery ports (e.g., shipping docks or airports) to warehouses. Cargo is moved via vehicles (e.g., trucks and trains) which are selected based on terrain, weather, accessibility and other factors. The primarily responsibility of the system's user is to initiate and monitor the routing of cargo through a GUI. The user can also request reports and estimations on cargo arrival times and vehicle status (e.g., idle, in use, under repair).

We used the EasyWinWin tool to gather and negotiate requirements for the cargo router system. The WinWin negotiation model [4] and its supporting tool (EasyWinWin [2]) are based on four artifact types: Win Conditions, Issues, Options and Agreements. Win conditions capture the stakeholder goals and concerns with respect to the new system. If a Win condition is non-controversial, it is adopted by an Agreement. Otherwise, an Issue artifact is created to record the resulting conflict among Win Conditions. Options allow stakeholders to suggest alternative solutions, which address Issues. Finally Agreements may be used to adopt an Option, which resolves the Issue.

Three stakeholders participated in a 1-hour brainstorming session and gathered 81 statements (stakeholder win conditions) about its goals.

3 CBSP STEPS

To create a "CBSP view" of a given set of requirements, we identified a five-step process [7], four of which are tool supported by EasyWinWin. In this section we will discuss all five steps in context of the Cargo router example.

Identify Core Requirements

Our approach is meant to be used in an iterative manner, where requirements get continuously added or changed. Thus, initially, we found it useful and necessary to reduce the complexity of a given problem by identifying core requirements. In this step, stakeholders vote about importance and relevance of a requirement. Naturally, requirements that did not get included in this step can be included in a future iteration of our process.

Architectural Classification of Requirements

To identify architecture-relevant information out of the pool of requirements, we used a voting process to categorize requirements into six CBSP dimensions (C,B,S,CP,BP,SP). We thus asked all stakeholders to individually decide whether they believe the given requirements could contain component-relevant information (C), bus (connector)-relevant information (B), or is a more general system requirement (S) that affects a larger part of the architecture. Since we were also interested in non-functional requirements, we also gave the stakeholders the option to vote for C-B-S properties (CP, BP, and SP). For instance, the requirement "R09: Support cargo arrival and vehicle availability estimation" was voted to be strongly component-relevant by all stakeholders whereas the requirement "R25: the system must be operational within 18 months" was not voted to be architecturally relevant. Some requirements also received contradictory votes: The requirement "R10: Automatic routing of vehicles" was voted component[1] relevant (C) by all stakeholders but only system property (SP) relevant by one stakeholder.

Identification and resolution of mismatches

As the last example showed, stakeholders may have distinct interpretations of requirements. Naturally, those discrepancies may lead to distinct interpretations of the architectural relevance of those requirements. Indeed, these are the kinds of conflicts we are seeking since the mapping from requirements to architecture often is a matter of

[1] Components can be either data components or processing components but that discussion is out of the scope here.

understanding the meaning of requirements in context of architectures. The conflict above indicates a case where two stakeholders have different opinions. A subsequent discussion step thus has to be initiated to identify and resolve that difference. In above example, one stakeholder thought this requirement implied that (1) the system needs to suggest paths that vehicles travel (e.g., via navigation points) but not their sources and targets whereas another stakeholder thought this requirement implied that (2) the system also needs to identify the sources and destinations for vehicles. The discussion thus clarified this conflict and an instant re-vote identified this requirement as component relevant and system property relevant (SP).

Table 1: Concordance Matrix.

Consensus	ACCEPT requirement as architecturally relevant if at least one *largely* or *fully* vote	REJECT requirement as not architecturally relevant if no vote higher than *partially*
Conflict	DISCUSS and resolve reason of conflict before proceeding (e.g., properties are implicitly captured and often ambiguous)	

Table 1 shows rules that describe conflict handling during CBSP voting. In case of consensus among the stakeholders, the requirements are either accepted or rejected based on the voted degree of architectural relevance (note that stakeholders are given the option of four votes: no, partial, strong, or full architectural relevance; our tool provides statistical reasoning on how to infer consensus). If the stakeholders cannot agree on the relevance of a requirement to the architecture, they further discuss it to reveal the reasons for the different opinions until a point of consensus is reached. This typically leads to a clarification of the particular requirements as above example has shown. Figure 1 (left) depicts a few requirements that were voted to be architecturally relevant.

Architectural refinement of requirements

Architecturally relevant requirements explicate at least one CBSP dimension that all stakeholders agreed on to be relevant. Obviously, some requirements address multiple dimensions. For instance, the requirement "R09: Support cargo arrival and vehicle availability estimation" was voted to be fully component relevant, fully system relevant, and largely bus relevant. In order to understand requirements better and to better relate them to other requirements it is necessary to refine them into more atomic entities.

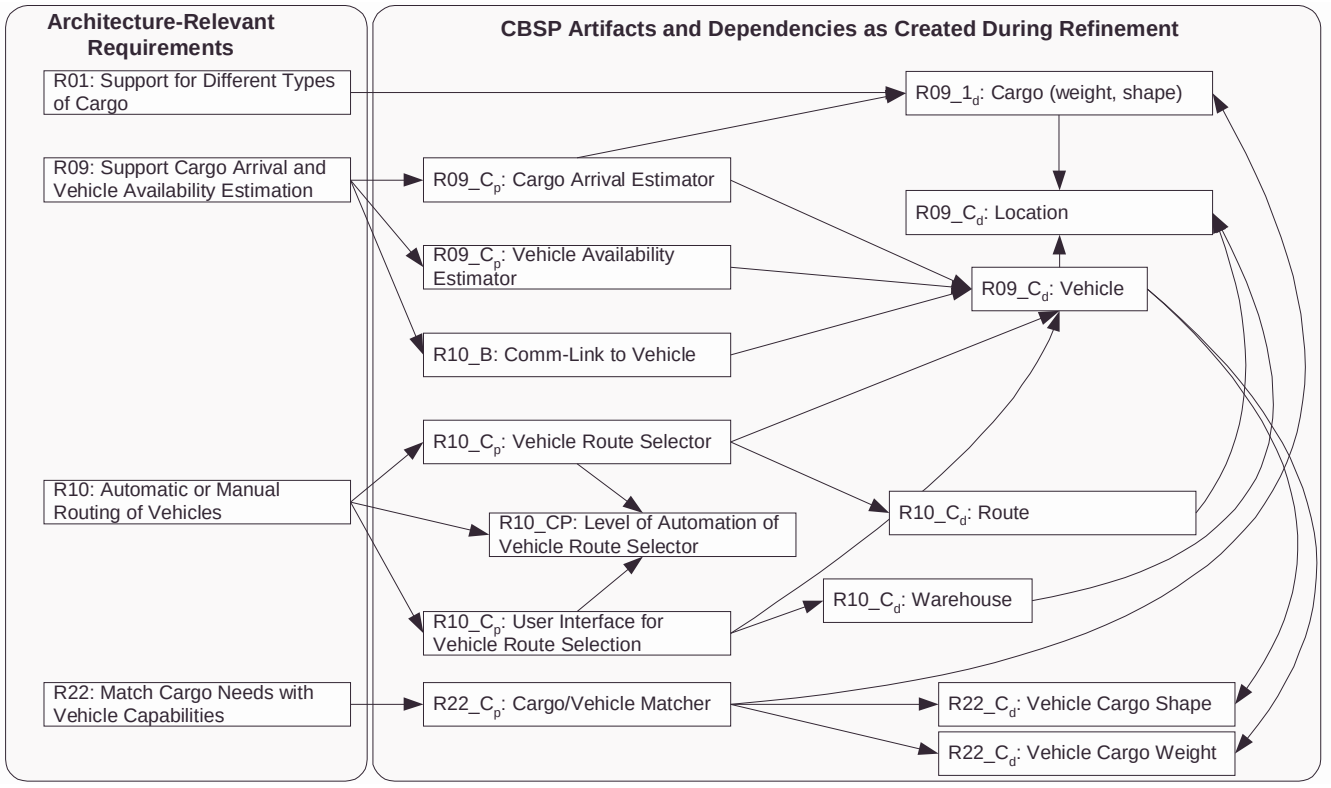


Figure 1: Sample Artifact Relationships in Cargo Router Example.

CBSP dimensions also play an important role in the refinement process of requirements. For instance, the requirement "R09: support cargo arrival and vehicle availability estimation" was determined C, B, and S relevant. This implied that the refinement should reveal component, bus, and system relevant aspects. Indeed, on a high level, this requirement supports two processing components: "R09_C_p Cargo arrival estimator" and "R09_C_p Vehicle availability estimator." Cargo arrival estimator depends on data components like cargo ("R09_C_d Cargo"), the vehicle ("R09_ C_d Vehicle") it is on and the location of the vehicle ("R09_ C_d Location"). Vehicle availability estimator only depends on the knowledge of the vehicle and its location (but not cargo). Above requirement was also rated bus-relevant. This was the case because the location of a vehicle (and its cargo) is variable as it moves. A connector (bus) is therefore needed to allow the system to track vehicles (see Figure 1 right). Note that we now do not talk about requirements any more but instead of pieces of architecturally-relevant information elicited from requirements – we refer to those pieces as CBSP artifacts.

It must be noted at this point, that generally only component, bus, and system artifacts are seen as candidates for refinement. Properties (CP, BP, SP) are harder to refine since they tend to span large parts of a system.

Derivation of Architectural Style and Architecture

CBSP artifacts and their dependencies are valuable for architectural modeling. For instance, we can see that the estimator components depend on vehicle information, thus, indicating potential architectural implications. Naturally, CBSP artifacts and their dependencies also make dependencies between requirements more explicit. For instance, we can assume some dependency between "R09: support cargo arrival and vehicle availability" and "R10: automatic or manual routing of vehicles" because they both "share" the CBSP artifact "R09_C_d: Vehicle."

In context of the Cargo Router we found that CBSP artifacts mapped straightforwardly to architectural elements defined in C2. For instance, the C2 architecture has components called *Vehicle* and *Estimator* and the architecture makes use of explicit data connectors (buses) to realize component interactions. However, we are not yet in a position where we could actually derive architectures or styles out of CBSP properties. We discuss this and other issues in future work.

4 FUTURE WORK

We found CBSP very useful in organizing requirements and systematically refining them but at the current state some of the activities are still rather labor intensive. Naturally, requirements engineering is people centric, however, this section will discuss how automation can aid stakeholders in coming up with requirements and architectures in a faster more reproducible way. We currently provide tool support for capturing requirements, voting on them, identifying conflicts, refining them, and maintaining trace dependencies. There are, however, a few areas that have not been explored in depth.

Architectural Trade-off Analyses

Thus far we treated requirements and architectures very static and defined how a CBSP approach can bridge the two. However, requirements engineering is much more iterative where stakeholders uncover not just goals but also issues (conflicts) and potential options (solutions). Issues may arise because of architectural conflicts (i.e., no suitable architectural option can be found that satisfies given requirements). We believe that CBSP artifacts are not only useful for refinement but they also provide "feedback loops" in cases where architectural decisions impact requirements. The approach thereby helps to capture findings from architectural modeling and simulation and supports analysis of an architecture for adherence to requirements.

For example, some issues can only be identified after a draft architecture has been modeled and described. These issues and corresponding architectural options can be captured by architects as CBSP elements to capture the rationale of architectural decisions and to relate this rationale with the relevant requirements. (A Bus Property issue (e.g., bottleneck) could be identified through simulation experiments, a component option could be suggested by the architect, etc.) This capturing of tradeoff decisions is similar to the ATAM technique described in [8]

Problems identified through architectural models and simulation can be captured as CBSP elements, such as "I12_S Three seconds system response time not possible." Architectural options and alternative solutions can be also described as CBSP elements. For example "O24_C Consider use of OTS staff management component." CBSP provides as an intermediate model between a requirements and an architecture definition approach that allows "bi-directional" traces; the resulting intermediate model facilitates synthesis of negotiation artifacts into architectural elements and enables feedback from architecture modeling and analysis.

CBSP can also be interpreted as a way to negotiate about architectural concerns (win conditions, issues, options, and agreements) with clearly established links to both the related requirements and the architectural elements.

CBSP Refinement of Issues, Options, and Agreements

If CBSP can be used to diversify requirements engineering via architectural trade-off analyses then naturally the subsequent negotiation results may also be in need of refinement. For instance, if a potential option were suggested to resolve a given issue then it would be desirable if CBSP could support the "explorative" refinement of that option in context of the rest of the

system to evaluate its feasibility.

To this end, Figure 2 shows one possible situations where requirements issues and options can be used to guide the refinement process. The left part of the figure shows three win condition (W1, W2, and W3) which could be requirements of an earlier negotiation process or simply new stakeholder goals. The middle part of the figure shows the corresponding CBSP artifacts (note that we did not define their interdependencies in the same detail as we did in Figure 1).

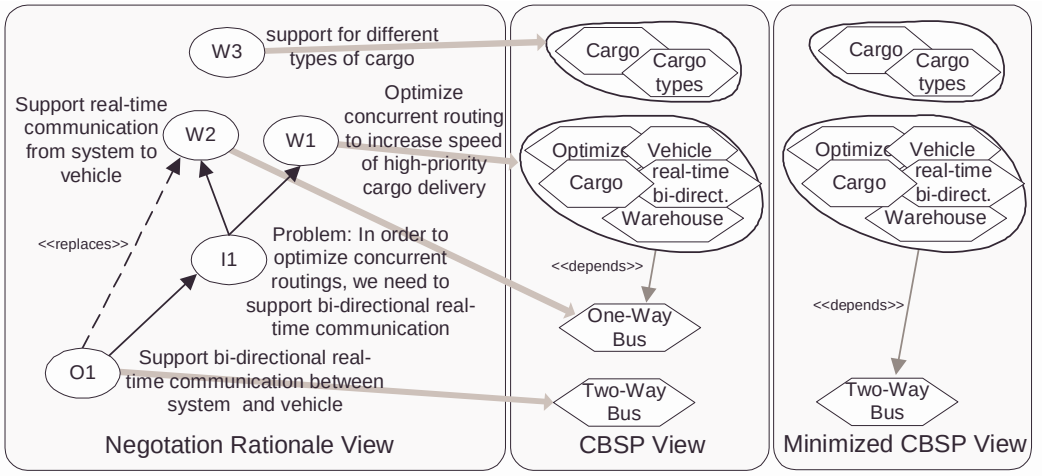


Figure 2. From Negotiation rationale to an CBSP view using Minimization

After some time (potentially with the help of architectural modeling) a problem (issue) is encountered between win conditions W1 and W2. It is found that in order to optimize concurrent routings (W1), we need to support bi-directional real-time communication. This contradicts W2 which only requires unidirectional communication from system to vehicle (e.g., to forward routing requests). This conflict could naturally be resolved via an option (O1) that states that we need a two-way bus; an option which would "replace" the win condition W2. In the CBSP view this causes a dilemma in that one requirement "artifact" requires a uni-directional bus whereas the other requires a bi-directional bus. Thus, CBSP refinement would also have to be complemented by a minimization step that would interpret requirements interdependencies (i.e., like the replace link between O1 and W2) to infer needed CBSP artifacts and their interdependencies. We believe that such a minimization step could be largely or even fully automated.

Inconsistency and Incompleteness Issues

Simplifying the refinement of requirements to architecture and reasoning about requirements feasibility in context of architectural modeling are two aspects we believe CBSP could support. However, the relationships between architecture, CBSP, and the negotiation rationale may become very complex when both architecture and requirements evolve independently. To this end, we found that CBSP could also provide powerful support for simplifying inconsistency detection between requirements and architecture. For instance, we observed the following cases:

- Inconsistencies between CBSP artifacts/dependencies and their actual realization: For instance, if a CBSP artifacts is categorized as a component but is implemented as a connector in the architecture then this could indicate a potential lack of understanding of either the requirement or architecture. Similar conclusions can be drawn when CBSP dependencies do not match architectural ones.

- Inconsistencies between architecture/CBSP and negotiation agreements: Take, for instance, the example of the *Optimizer* component in Figure 2 which depends on the *Two-Way-Bus*. If during the WinWin negotiation it is decided to implement the *Optimizer* but at the same it is decided to implement the *One-Way-Bus* instead of the *Two-Way-Bus* (i.e., O1 would not get adopted) then this indicates a potential mismatch (*Optimizer* needs the *Two-Way-Bus*).

- Completeness mismatch between architecture and requirements: For instance, CBSP can help in identifying whether all agreed-upon architecturally-relevant artifacts have actually been realized in the architecture. Similarly, CBSP can help in pointing out if there are any architectural elements for which there are no corresponding negotiation artifacts. Completeness issues such as the ones above could suggest lack of awareness by stakeholders of some architectural aspects that could have influenced the negotiation process and vice versa.

Deriving/Validating Architectural Styles out of CBSP

In our work to date, we have chosen to use architectural styles as guides in transforming the initial architectural decisions produced by CBSP into an actual architecture. Specifically, we have employed the C2 style [12] as discussed above. However, each style is particularly well suited for a certain type of problem; therefore, our intent is to extend CBSP to leverage other styles as well.

We have begun exploring the feasibility of composing CBSP artifacts into an architecture according to the Pipe-and-Filter [11], GenVoca [1], and Weaves [5] styles, in

addition to C2. Each style imposes different constraints that guide the composition of CBSP artifacts into an architecture. For example, in the cargo router example, the *Optimizer* CBSP artifact depends on the *Vehicle* and *Warehouse* artifacts. C2's substrate independence principle mandates that *Optimizer* be placed below them in the architecture. Since there are no direct dependencies between *Vehicle* and *Warehouse*, they may be adjacent. The same dependency relationship would have different topological implications in a different style. For example, GenVoca would require *Optimizer* to be above the *Vehicle* and *Warehouse* components (while still allowing *Vehicle* and *Warehouse* to be at the same level). Furthermore, unlike C2, GenVoca would allow direct interactions among its components, without the intervening connectors.

Similarly, if a component in a system, e.g., *Weather Module*, communicates by producing streams of data, while other components, e.g., *Vehicle*, assume discrete event-based communication, then the style selected to represent the architecture must supply explicit software connectors to mediate between the two types of interaction. In this case, GenVoca would not be an adequate candidate, while Weaves, Pipe-and-Filter, and C2 may be, as all three of them provide explicit connectors. However, if we further consider the types of component interaction supported by the three styles, we see that neither C2 nor Pipe-and-Filter provide adequate solutions in this case: C2 assumes purely discrete event-based communication, while Pipe-and-Filter assumes purely data stream-based communication. This would leave Weaves as the obvious choice.

Our future work will center around expanding the number of architectural style we are considering. We will also leverage existing studies on styles to codify the style elimination and selection criteria such as the ones outlined above. Finally, we intend to determine whether there are architectural styles that are inherently incompatible with CBSP, and the reasons for that incompatibility

5 CONCLUSION

This paper introduced the CBSP approach for refining requirements to architectures. The process is partially tool supported and is currently integrated with our EasyWinWin negotiation process. Besides requirements refinement, the CBSP process also has great potential for improving a variety of related issues like consistency and conformance and architectural trade-off analyses. Future work involves the exploration of those issues as well as a more tight integration of our models and tools.

ACKNOWLEDGEMENTS

This work has been supported by the Austrian Science Fund (Erwin Schrödinger Grant 1999/J 1764 "Collaborative Requirements Negotiation Aids").

REFERENCES

1. Batory D. and O'Malley S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology, 1992.
2. Boehm, B. and Gruenbacher, P.: "Supporting Collaborative Requirements Negotiation: The Easy WinWin Approach," Proceedings of SCS Virtual Worlds and Simulation Conference, January 2000.
3. Boehm B., Egyed A., Kwan J., and Madachy R.: Using the WinWin Spiral Model: A Case Study. IEEE Computer, 1998, 33-44.
4. Boehm B. W. and Ross R.: Theory W Software Project Management: Principles and Examples. IEEE Transactions on Software Engineering, 1989, 902-916.
5. Gorlick, M. M. and Razouk, R. R.: "Using Weaves for Software Construction and Analysis," Proceedings fot eh International Conference on Software Engineering (ICSE), Los Alamitos, CA, 1991, pp.23-34.
6. Gruenbacher, P. and Briggs, B.: "Surfacing Tacit Knowledge in Requirements Negotiation: Experiences using EasyWinWin," Proceedings of the Hawaii International Conference on Systems Sciences, 2001.
7. Gruenbacher, P., Egyed, A., and Medvidovic, N.: "Reconciling Software Requirements and Architectures: The CBSP Approach," Submitted to International Requirements Engineering Conference (RE), Toronto, Canada.
8. Kazman, R., Barbacci, M., Klein, M., Carričre, S. J., and Woods, S. G.: "Experience with Performing Architecture Tradeoff Analysis," Proceedings of the 21th International Conference on Software Engineering (ICSE), Los Angeles, CA, May 1999, pp.54-63.
9. Medvidovic, N., Gruenbacher, P., Egyed, A., and Boehm, B.: "Software Lifecycle Connectors: Bridging Models across the Lifecycle," submitted to International Conference on Software Engineering and Knowledge Engineering (SEKE 2001), 2001.
10. Perry D. E. and Wolf A. L.: Foundations for the Study of Software Architectures. ACM SIGSOFT Software Engineering Notes, 1992.
11. Shaw, M. and Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.
12. Taylor R. N., Medvidovic N., Anderson K. N., Whitehead E. J. Jr., Robbins J. E., Nies K. A., Oreizy P., and Dubrow D. L.: A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering 22(6), 1996, 390-406.

From Requirements Negotiation to Software Architectural Decisions

Hoh In
Dept. of Computer Science
Texas A&M University
College Station, TX 77843
hohin@cs.tamu.edu

Rick Kazman
Software Engineering Institution
Carnegie Mellon University
Pittsburgh, PA, USA 15213
Kazman@sei.cmu.edu

David Olson
Dept. of Info. Operations and Management
Texas A&M University
College Station, TX 77843
dolson@tamu.edu

Abstract

Uncertainty of system properties (e.g., performance, reliability, security, interoperability, usability, etc.) often hinders the progress of requirements negotiation. Software architecture evaluation techniques enable stakeholders to clarify the uncertainty of system properties. In another hand, software architecture alternatives cannot be evaluated in a thorough way without consideration of different stakeholders' negotiated requirements. Effective requirements negotiation is therefore needed to evaluate architecture alternatives.

This paper proposes an integrated decision-making framework from software requirements negotiation to architecture evaluation based on WinWin and CBAM (Cost Benefit Analysis Method). The integrated framework helps stakeholders elicit, explore, evaluate, negotiate, and agree upon software architecture alternatives based negotiated requirements.

Keywords: ABASs, ATAM, CBAM, conflict resolution, requirements negotiation, WinWin.

1. Motivation

Many software projects have failed because their requirements were poorly negotiated among stakeholders [4]. Several keynote speakers in the International Conference on Software Engineering (ICSE) emphasized the importance of requirements negotiation as follows:

- "How the requirements were negotiated is far more important than how the requirements were specified" (Tom De Marco, ICSE 96)
- "Negotiation is the best way to avoid "Death March" projects" (Ed Yourdon, ICSE 97)
- "Problems with reaching agreement were more critical to my projects' success than such factors as tools, process maturity, and design methods" (Mark Weiser, ICSE 97)

The WinWin negotiation model, developed by the USC Center for Software Engineering, provides a general framework for successful requirements negotiation. In WinWin, stakeholders elicit their win conditions, identify issues/conflicts, generate options to resolve the issues, negotiate the options and reach agreement [1,2,3]. However, it is not clear which architecture alternatives should be considered as the options and/or how they should be explored, evaluated, and negotiated in order to reach agreement among stakeholders.

As an architecture evaluation technique, the CMU Software Engineering Institute has developed the Cost Benefit Analysis Method (CBAM) that explores, analyzes, and makes decisions regarding software architecture alternatives (called "architecture strategies") [14]. Still, it is not clear how the explored architecture strategies satisfy the initial requirements (goals/constraints) of stakeholders who have different roles, responsibilities, and priorities. A general negotiation framework to aid in progressing from requirements to architectural decisions is needed.

In this paper, we propose an integrated decision-making framework, based on WinWin and CBAM, that aids in systematically determining architecture alternatives from negotiated requirements among stakeholders. WinWin provides a general negotiation framework to elicit requirements, explore architecture alternatives, and reach agreement. CBAM helps stakeholders negotiate architecture alternatives in a systematic way.

This paper is organized as following: Section 2 describes the context of the work. Section 3 presents and describes the proposed framework. In Section 4 and 5, future research challenges and conclusions are presented.

2. Context For the Work

2.1 WinWin Negotiation Model

The WinWin model provides a general framework for identifying and resolving requirements conflicts by eliciting and negotiating artifacts such as win conditions, issues, options, and agreements. The WinWin model uses Theory W [5], "Make everyone a winner", to generate the

stakeholder win-win situation incrementally through the Spiral Model. WinWin assists stakeholders to identify and negotiate issues (i.e., conflicts among their win conditions), since the goal of Theory W involves stakeholders identifying their win conditions, and reconciling conflicts among win conditions.

The dotted-lined box (steps 1,2,3, and 8) shown in Figure 1 presents the WinWin Negotiation Model. Stakeholders begin by entering their win conditions (step 1). If a conflict among stakeholders' win conditions is identified, an issue schema is composed, summarizing the conflict and the win conditions it involves (step 2). For each issue, stakeholders prepare candidate option schemas addressing the issue (step 3). Stakeholders then evaluate the options, delay decision on some, agree to reject others, and ultimately converge on a mutually satisfactory option. The adoption of this option is formally proposed and ratified by an agreement schema, including a check to ensure that the stakeholders' iterated win conditions are indeed covered by the agreement (step 8). Experience also indicates that WinWin is not a panacea for all conflict situations, but generally increases stakeholders' levels of cooperation and trust [4, 11].

Agreement is not always guaranteed. There are often tradeoffs among win conditions that need to be balanced. CBAM provides a means to balance these tradeoffs, and a framework for discussion that can lead to resolution.

2.2 CBAM (Cost-Benefit Analysis Method)

The ATAM [13] uncovers the architectural decisions made in a software project and links them to business goals and QA (quality attribute) response measures. The CBAM [14] builds on this foundation by additionally determining the costs, benefits, and uncertainties associated with these decisions.

Given this information, the stakeholders can then decide how to address their important QA response measures. For example, if they felt that the system's reliability was not sufficiently high they could use the ATAM/CBAM methods to decide whether to use redundant hardware, checkpointing, or some other architectural decision addressed at increasing the system's reliability. Or the stakeholders can choose to invest their finite resources in some other QA—perhaps believing that higher performance will have a better benefit/cost ratio. A system always has a limited budget for creation or upgrade and so every architectural choice is, in some sense, competing with every other one for inclusion.

The CBAM is a framework and it does not make decisions for the stakeholders; it simply aids them in the elicitation and documentation of costs, benefits, and uncertainty and gives them a rational decision-making process.

When an ATAM is completed, we expect to have a set of artifacts documented as follows:

- a description of the *business goals* that are crucial to the success of the system
- a set of *architectural views* that document that existing or proposed architecture
- a *utility tree* which represents a decomposition of the stakeholders' goals, for the architecture. The utility tree starts with high-level statements of QAs and decomposes these into specific instances of performance, availability, etc. requirements and realizes these as scenarios
- a set of *risks* that have been identified
- a set of *sensitivity points* (architectural decisions that affect some QA measure of concern)
- a set of *tradeoff points* (architectural decisions that affect more than one QA measure, some positively and some negatively)

The CBAM builds upon this foundation of information by probing the architectural strategies (ASs) that are proposed in response to the scenarios, risks, sensitivity points, and tradeoffs. The steps of the CBAM are as follows. Each of these steps can be executed in the first (triage) and second (detailed examination) phases:

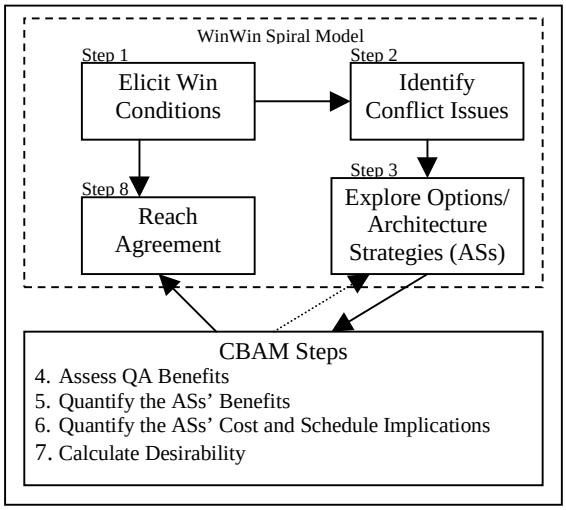
1. Choose Scenarios and Architectural Strategies
2. Assess QA Benefits
3. Quantify the Architectural Strategies' Benefits
4. Quantify the Architectural Strategies' Costs and Schedule Implications
5. Calculate Desirability
6. Make Decisions

3. The Steps of the Integrated Framework

The integrated framework shown Figure 1 begins with the WinWin process, which elicits what stakeholders need, identifies conflicts in these needs among the stakeholders, and explores the conflict-resolution options. CBAM is proposed here as a means to supplement the WinWin process of systematically evaluating and negotiating software architecture alternatives (as conflict-resolution options) by eliciting stakeholders' benefits and costs. The process may lead to agreement by itself (although this is not guaranteed). Reviewing each stakeholder' win conditions at

this final stage may further aid the next cycle of reconciliation or compromise in the WinWin Spiral Process model.

Figure 1: *The Integrated Framework*



The steps shown in Figure 1 are elaborated in the following subsections.

Step 1: Elicit Win Conditions

Each stakeholder identifies their win conditions. This step provides the basis for identification of ideal project features by stakeholders.

Step 2: Identify Quality Attribute Conflicts/Issues

The lists of win conditions are then reviewed to identify quality attribute conflicts. The identified conflicts are then categorized as being either a direct conflict or a potential conflict. This step may be accomplished manually, but future work may be able to incorporate software agents.

Step 3: Explore Architecture Strategies as Conflict-Resolution Options

Based upon the conflicts/issues generated in step 2, the stakeholders can now generate conflict-resolution options. It is best to generate a list of options which may emphasize those characteristics preferred by each stakeholder, but that include some balance representing needed conditions of all stakeholders. These options are called Architectural Strategies (ASs) in the CBAM. Where do such ASs come from? They can come from any number of areas: from the architects' experience, by borrowing from systems that have experienced similar problems in the past, or from repositories of design solutions, such as Design Patterns [10] or ABASs [15].

Step 4. Assess Quality Attribute (QA) Benefits

To aid in decision making, the stakeholders now need to determine both the costs and benefits that accrue to the various ASs. Determining costs is a well-established component of software engineering. This is not addressed directly by the CBAM—we assume that some methods of doing this exists in the organization. Determining benefits is less well-established and this is the province of the CBAM. As a means of determining the benefit of an individual AS, a benefit evaluation function is created. Benefit must be correlated with the degree to which an Architectural Strategy supports QA goals, which in turn relates back to the business goals for the system. These goals are both outputs of the ATAM.

To do this we have each of the stakeholders assign a *Quality Attribute Score* (*QAScore*) to each QA system goal. We let the customer determine which stakeholders should be in a decision-making capacity. The stakeholders are instructed to choose these scores such that they total 100. For example:

Performance: 15
Security: 15
Modifiability: 30
Reliability: 25
Interoperability: 15

We also ask each stakeholder to describe the particular aspect of the quality attribute that caused them to make this score. For example, modifiability has a score of 30 and above, but it is *GUI modifiability* that is the primary determinant of this score.

Step 5: Quantify the Architecture Strategies' Benefits

We then use these scores to evaluate each of the ASs. Very rarely does an AS only affect a single QA. ASs will have effects on multiple QAs, some positive and some negative, and to varying degrees. To capture this, we ask the stakeholders to rank each AS in terms of its *contribution* (*Cont*) to each QA on a scale of –1 to +1. A +1 means that this AS has a substantial positive effect on the QA (for example, an AS under consideration might have a substantial positive effect on performance) and a -1 means the opposite. Based upon this information each AS_i can now be assigned a computed benefit score from –100 to +100 "Benys" using the following formula:

$$Benefit(AS_i) = Sum\ (Cont_{ij} * QAScore_j)$$

For example, given the QAscores listed above, we can calculate benefit scores for two hypothetical ASs as follows

(note that we only consider the QAs for which there is a non-zero contribution):

AS5: Performance (-0.2), Modifiability (0.6), Interoperability (0.3)

Benefit(AS5) = -0.2 * 15 + 0.6 * 30 + 0.3 * 15
= 20.5

AS6: Performance (0.8), Reliability(-0.2), Security(-0.4)

Benefit (AS6) = 0.8 * 15 + -0.2 * 25 + -0.4 * 15
= 1

This score allows us to rank the benefit of every architectural change that has been contemplated. But clearly this evaluation is fraught with uncertainty. We can capture this uncertainty by recording the variations in stakeholder judgements.

In the CBAM we use Kendall's concordance coefficient for the group as a whole as a measure of the uncertainty of the group, as described in [14]. The more highly correlated the group, the higher the concordance coefficient and hence the lower the uncertainty.

Step 6: Quantify the Architecture Strategies' Cost and Schedule Implications

Now that the benefits have been estimated by the stakeholders, we must capture two other crucial pieces of information about the various ASs: their costs and their schedule implications. We propose no special cost estimation technique here (although we do think that cost estimation methods that take architecture into account are a desirable and inevitable improvement to existing methods). We assume that an organization has some method (even if it is *ad hoc*) of estimating the costs of implementing new services and features. We simply need to capture these estimates, as they are associated with each AS.

In addition we need to capture any schedule implications of the ASs. For example, do several ASs require the use of the same critical resource (personnel, hardware, software)? If so then attempting to implement them simultaneously might be impossible even if their cost/benefit numbers indicate that this is the strategy that brings the organization the greatest profit. Similarly, we want to note cases where AS_i depends upon AS_j and so implementing AS_j first actually reduces the cost of implementing AS_i.

Step 7: Calculate Desirability

Given this information we are in a position to calculate a "Desirability" metric, as follows:

$$Desirability(AS_i) = Benefit(AS_i) / Cost(AS_i)$$

This metric indicates ASs that will bring high benefit to the organization at relatively low cost. In addition to calculating this metric, the absolute benefit and cost numbers need to be considered as does the uncertainty surrounding all of these numbers, as discussed in [14].

Step 8: Reach Agreement

At this point the negotiating among the stakeholders can begin in earnest. This negotiation will be informed, rather than simply a matter of opinion. The costs, benefits, and uncertainty of each of the ASs will be plain for each stakeholder to see, as well as the schedule implications and dependencies (if any). These ASs can be tied back to the business goals, and hence the win conditions of each of the stakeholders.

What results is much less an argument than a discussion about priorities, risk averseness, and the assumptions underlying the model. Stakeholders may still disagree about what direction to take the architecture and the system, but they will do so based upon a large base of facts and accumulated evidence and such disagreements can be more easily moderated than those which are simply based upon opinion and prejudice.

4. Further Research Challenges

The integrated decision-making framework offers useful tools to aid the stakeholder negotiation process from requirements to architecture evaluation. However, there are a number of challenges involved in its process. These challenges are given briefly here due to lack of space, but provide a great deal of fruitful scope for future research.

Exploration of Architecture Strategies: An important issue is how to sort these issues/conflicts (shown in Step 2) in order to explore Architecture Strategies as conflict-resolution options (shown in Step 3) reflecting the win conditions of the different stakeholders. That is, should several Architecture Strategies be explored for each issue/conflict, or per a set of issues/conflicts, or for all issues? Our feasible solution approach could be to classify (cluster) conflicting issues and identify a small set of Architecture Strategies for each set of the clustered issues/conflicts. One possible solution would use a cross-impact (or dependency) analysis technique, which would identify clusters of stakeholder positions. An ideal solution for each cluster could be used as the basis of an Architecture Strategy. This would imply the need to consider the entire set of criteria.

Determining Detailed Benefit-Cost Criteria: Agreeing on detailed criteria of benefits and costs among stakeholders is a challenging problem [7, 8, 9, 12, 15, 20]. An initial, complete list of criteria can be generated first. This long list of micro-criteria can then be reviewed and grouped by theme, yielding the macro-criteria. The identification of the macro-criteria is usually a natural grouping of micro-criteria representing a common theme.

Sensitivity to Uncertainty in Benefit, Cost Values: Objectively quantifying the benefits and costs of Architecture Strategies (shown in Step 5 and 6) is another challenging problem. Even though we can quantify them with popular cost and/or benefit models, the accuracy of the estimates is limited. Thus, the inherent uncertainty in the models must be taken into account [14].

Reaching Agreement from Desirability Results: Reaching agreement is a difficult task. One way to accomplish agreement is to let an arbitrator (e.g., the responsible manager) make the decision. Use of a group support system lets all stakeholders have the opportunity to provide their inputs. That alleviates some of the apparent arbitrariness usually perceived in dictatorial decisions. At the opposite extreme, the decision could be made by voting. As in political decision-making, this does not guarantee complete acceptance. Quite the contrary, all of the mechanisms that have been applied to reaching a decision can be applied in the proposed system. Hopefully, the opportunity to express win conditions, the use of group support systems, and objective cost-benefit evaluation modeling will create a decision-making environment that gains broader support from participating stakeholders.

5. Conclusions

In this paper, we have introduced an integrated framework for coordinating architectural decisions with requirements negotiation framework. This integrated framework has the following synergy compared to the requirements negotiation models [16, 17, 18, 19, 21] or to software architecture evaluation work (e.g., [13, 14, 15]):

- Enabling a more powerful multi-viewpoint analysis of architecture evaluation. Architecture alternatives (e.g., "Architecture Strategies") can be evaluated based on requirements elicited, explored, and negotiated from and by multiple stakeholders who have different roles, responsibilities, and priorities.
- Facilitating requirements negotiation in a more systematic way. Uncertainty in requirements negotiation can be clarified through the exploration, evaluation, and negotiation of architecture alternatives.

As a logical step, we will examine a real-world case study to obtain more insight of better way to overcome the future research challenges presented in Section 4.

In conclusion, we expect that the integrated framework provides a systematic, yet practical method for stakeholders to negotiate from requirements to architecture alternatives.

ACKNOWLEDGEMENTS

This work is partially supported by funding from NASA JPL under the contract C00-00443 with Texas A&M University. We would like to thank Drs. Barry Boehm, Robert Briggs, and Tom Rodgers for the helpful discussion on this topic.

REFERENCES

[1] Boehm, B., Bose, P., Horowitz, E., and Lee, M., "Software Requirements as Negotiated Win Conditions", in *First International Conference on Requirements Engineering* (ICRE94). Colorado Springs: IEEE Computer Society Press, 1994.

[2] Boehm, B., Bose, P., Horowtiz, E., and Lee, M., "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach", in *17th International Conference on Software Engineering* (ICSE-17). Seattle: IEEE Computer Society Press, 1995.

[3] Boehm, B., Egyed, A., Port, D., Shah, A., Kwan, J., and Madachy, R., "A Stakeholder Win-Win Approach to Software Engineering Education", *Annals of Software Engineering*, 1999.

[4] Boehm, B. and In, H., "Identifying Quality-Requirement Conflicts", IEEE Software, Vol. 13, No. 2, pp. 25-35, March 1996.

[5] Boehm, B. and Ross R., "Theory W Software Project Management: Principles and Examples", *IEEE Transactions on Software Engineering*, 1989. July: p. 902-916.

[6] Gruenbacher, Paul & Briggs, R.O. "Surfacing Tacit Knowledge in Requirements Negotiation: Experiences Using EasyWinWin", *Proceedings of the 34th Thirty Fourth Hawaii International Conference on System Sciences*, CLUS09, 2001

[7] Choo, E.U., Schoner, B., and Wedley, W.C., "Interpretation of Criteria Weights in Multicriteria Decision Making", *Computers and Industrial Engineering* 37, 1999, pp. 527-541.

[8] Dyer, J.S. and Sarin, R.K., “Measurable Multiattribute Value Functions”, *Operations Research* 27, 1979, pp. 810-822.
[9] Edwards, W. and Barron, F.H., “SMARTS and SMARTER: Improved Simple Methods for Multiattribute Utility Measurement”, *Organizational Behavior and Human Decision Processes* 60, 1994, pp. 306-325.
[10] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns, Addison Wesley, 1995
[11] In, H., Boehm, B., Rodgers, T., and Deutsch, M., "Applying WinWin to Quality Requirements: A Case Study", IEEE International Conference on Software Engineering (ICSE 2001), IEEE Computer Society Press, Toronto, Canada, May 12-19, (to appear)
[12] Keeney, R.L. and Raiffa, H., *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley, New York, 1976.
[13] Kazman, R., Klein, M., Clements, P., “ATAM: A Method for Architecture Evaluation”, CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, 2000.
[14] Kazman, R., Asundi, J., Klein, M., “Quantifying the Costs and Benefits of Architectural Decisions”, *Proceedings of the 23rd International Conference on Software Engineering (ICSE 23)*, (Toronto, Canada), May 2001, to appear.
[15] M. Klein, R. Kazman, L. Bass, S. J. Carriere, M. Barbacci, H. Lipson, “Attribute-Based Architectural Styles”, *Software Architecture* (*Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1))*, (San Antonio, TX), February 1999, 225-243.
[16] Klein, M., “Supporting Conflict Resolution in Cooperative Design Systems”, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 21, No. 6, pp1379-1390, Nov/Dec 1991.
[17] Lee, J., “SIBYL: A Qualitative Decision Management System”, In Artificial Intelligence at MIT: Expanding Frontiers, Edited by P. Winston and S. Shellard, MIT Press, Cambridge, 1990.
[18] Mylopoulos, J., Chung, L., Wang, H.Q., Liao, S., Yu, E. “Extending Object-Oriented Analysis to Explore Alternatives”, IEEE Software. February 2001.
[19] Nuseibeh, B. A., Easterbrook, S. M., and Russo, A., "Leveraging Inconsistency in Software Development", IEEE Computer, Volume 33, No. 4, Pages 24-29, April 2000.
[20] Olson, D.L., *Decision Aids for Selection Problems.* Springer, New York, 1996.
[21] Ramesh, B. and Sengupta, K., “Managing Cognitive and Mixed-motive Conflicts in Concurrent Engineering”, Concurrent Engineering, Vol. 18, No. 6, 1992, pp 498-519.

Transforming Goal Oriented Requirement Specifications into Architecture Prescriptions

Manuel Brandozzi
Center for Advanced Experimental Software Engineering Research
The University of Texas at Austin
mbrandoz@ece.utexas.edu

Dewayne E. Perry
Center for Advanced Experimental Software Engineering Research
The University of Texas at Austin
perry@ece.utexas.edu

Abstract

In this paper we propose a new method to transform the requirements specification for a software system into an architectural specification for the system. In the introduction we illustrate the needs for this new method in the context of the software development process and we explain the concept of architecture prescription. Then, we give a brief overview of KAOS, the goal oriented requirements specification language we used as a starting point. We characterize the APL (Architecture Prescription Language) and show how to use it to derive an architecture prescription from the KAOS requirements. We then illustrate our technique with a practical example, namely the meeting-planning problem. Finally, we discuss related work and indicate future directions of our research.

1. Introduction

At present, there are many different software development processes used in industry. These processes may be more or less suited for the particular application being developed. A common characteristic of most of the development processes is that there is feedback from one phase to another one. Let's consider the earlier phases of a generic development process with feedback.

The process starts with the "requirements analysis and specification" phase. In this phase, the requirements engineer has to understand the user's needs and to document them, either formally or informally.

The second phase is the "architectural design" phase. In this phase, the system architect selects the architectural elements, their interactions and the constraints on these elements and interactions to achieve a basic framework to satisfy the requirements specified in the previous phase.

Then there is the design phase. During this phase the designer decides how to decompose the elements described in the architectural design into low level modules, which already existing components might be reused to implement these modules, or which algorithms and data structures should be used to implement the modules.

The later phases of the process include coding, testing, integration, delivery and maintenance. Each of the front-end phases can be viewed as the implementation of the previous one.

The process is iterative because, typically, either to make the implementation of a phase feasible, or more efficient, we return to previous phases, one or more times, and modify the relevant software artifacts of those phases. Coming back from one phase to a previous one constitutes considerable work and time overhead because it's generally very difficult to understand what has to be modified in the previous phase and these modifications may have side effects. So, there is a need to minimize the number of iterations, or to make them easier to perform and easier to identify their side effects whenever they are necessary. Iterations in the process happen also when it is discovered that the system doesn't do what the user really wanted. So, we feel we can improve the process by using better methods to specify the requirements and by using rigorous techniques to pass from each phase of the development process to the next one.

There are further advantages of passing from a phase to the next one using formal techniques. By doing this, we achieve reusability of part of the artifact of each phase. The earlier the phase the artifact belongs to, the higher is the gain obtained. Let's suppose, for instance, that we have to develop a new system for which only a small part of the requirements differ from those of a system already developed. With our approach that maps the requirements to the components derived from them, we know exactly what has to be changed in the architecture of the system already developed in order to obtain an architecture of the new system. The same

applies when some of the requirements of a system are changed, for whatever reason, during or after the development of the system.

As experience shows, in traditional approaches the modification of requirements might have very subtle effects on the architecture and a brand new architecture may be needed each time requirements are modified if we are to achieve a reliable system. The method we introduce in this paper, by providing requirements to architecture traceability, enables us to reuse parts of the architecture, and hence to reuse all the derived artifacts that implement the architectural components. It enables a development team to save both time and resources.

Our work has been focused on finding a method for the first of the transitions from one phase to the next: the transition going from the requirements specification to the architectural design, i.e. the one that has the highest leverage. Traditionally, this transition has been one of the most difficult aspects of software engineering. The primary problem in software development is transforming *what* we want the system to do into a basic framework for *how* to do it. Our method takes as input goal oriented requirement specifications and returns as output an architecture prescription. An architecture prescription is an alternative way to specify an architecture. We chose goal oriented specifications because we think they are, among all the kinds of requirements specifications, those more near to the way human thinks and are easy to understand by all the stakeholders. Another reason is that they are particularly suitable to be transformed into an architecture prescription. In the next section we'll give a brief description of KAOS, the goal oriented specification language that we used in our example that has been first introduced by Axel van Lamsweerde et al. [1].

Let's now explain what we mean by an architecture prescription, a concept introduced by Dewayne E. Perry and Alexander L. Wolf [3]. An architecture prescription lays out the space for the system structure by restricting the architectural elements (processes, data, connectors), their relationships (interactions) and constraints that can be use to implement the system. The main advantages of an architecture prescription over a typical architecture description are that it can be expressed in the problem domain language and it's often less complete, and hence less constraining with respect to the remaining design of the system. An architectural prescription concentrates on the most important and critical aspects of the architecture and these constraints are most naturally expressed in terms of the problem space (or business domain, the domain of the problem). An architecture description, on the other hand is a complete description of the elements and how they interface with each other and tends to be defined in terms of the solution space rather than the problem space (or in terms of components such as GUIs, Middleware, Databases, etc, that are used to implement the system). Since an architecture prescription is expressed in the domain language, it makes it easier to create a mean of transforming requirement specifications into architectural specifications. The two kinds of specifications can make use of a common vocabulary to relate the requirements' goals to the architectural constraints.

The purpose of our work is twofold: to propose architecture prescriptions as a way to specify the architectures of software systems, and to design a technique to transform the requirements specifications into prescriptive specifications.

The rest of the paper is structured as follows: in section 2 we give an overview of KAOS, in section 3 we show how to derive from a KAOS specification the architecture prescription whose architectural elements, and the way these elements interact, are defined via application specific constraints. In section 4 we'll give a practical example of the method using the meeting-planning problem and, finally, in section 5 we will summarize the contribution of our work and illustrate its future directions.

2. Overview of the KAOS Specification Language

KAOS is a goal oriented requirements specification language [1]. Its ontology is Composed of:

- *Objects* - they can be:
  - *Agents*: active objects
  - *Entities*: passive objects
  - *Events*: instantaneous objects
  - *Relationships*: depend on other objects

- *Operations*: they are performed by an agent and change the state of one or more objects. They are characterized by pre-, post- and trigger-conditions.

- *Goal*: it's an objective for the system. In general, a goal can be AND/OR refined till we obtain a set of goals achievable by some agents by performing operations on some objects. The refinement process generates a refinement tree.

- *Requisites, requirements and assumptions*: the leaves obtained in the goal refinement tree are called requisites. The requisites that are assigned to the software system are called requirements; those assigned to the interacting environment are called assumptions.

How are requirements specified? The high-level goals are gathered from the users, domain experts and existing documentation. These goals are then AND/OR refined till we derive goals achievable by some agents. For each goal the objects and operations associated with it have to be identified. Of course more than one refinement for a goal may be possible, and there may be conflicts between refinements of different goals that can be resolved as proposed in [2]. It's up to the requirements engineer to choose the best refinement tree. A refinement tree could be modified afterwards in case there are problems implementing the artifacts of a latter phase of the development process.

In exhibit 1. there is an example of a goal specified using KAOS.

Goal *Achieve*[MeetingRequestSatisfied]
InstanceOf SatisfactionGoal
Concerns Meeting, Initiator, Participant
ReducedTo SchedulerAvailable,
ParticipantsConstraintsKnown,
MeetingPlanned,
ParticipantsNotified
InformalDef Every meeting request should be satisfied within some deadline associated with the request. Satisfying a request means proposing some best meeting date/location to the intended participants that fit their constraints, or notifying them that no solution can be found with those constraints.

Exhibit 1. example of a goal specification in KAOS

The *Goal* keyword denotes the name of the goal; *InstanceOf* declares the type of the goal; *Concerns* indicates the objects involved in the achievement of the goal; *ReducedTo* traces into which sub-goals the goal is resolved. Finally, there is informal definition of the goal followed by an optional formal definition. *FormalDef* is the optional attribute; it contains a formal definition of the goal that can be expressed in any formal notation.

3. From Requirements to Architecture

3.1 From KAOS entities to APL entities

How is it possible to transform a KAOS requirements specification into an architecture prescription for the software system? Exhibit 2. shows the correspondence we found between KAOS entities that refer to a subset of the system specification and the Architecture Prescription Language (APL) entities that describe the constraints on the software architecture. The subset of the overall system specification considered is the subset concerning the software system specification.

KAOS entities	**APL entities**
•Agent	•Process component / Connector component
•Event	•Event
•Entity	•Data component
•Relationship	•Data component / Relationship among components
•Goal	•Constraint on the system or on a subset of the system •One or more additional processing, data or connector components

Exhibit 2. Mapping KAOS entities to APL entities

Each *object* in the requirements generally corresponds to a *component* in the architecture. More specifically, an *agent object*, an active object, corresponds to either a *process* or a *connector*. By definition, a process (thread, task) is an active component. What might not be immediately apparent is that also a connector can be an active component. An example of this type of connector is a software firewall. A software firewall is an active entity that checks whether the processes that want to interact satisfy some conditions or not, and allows or denies the interaction among them accordingly.

The *events* relevant to the architecture of the system are those either internal to the software system or those in the environment that have to be taken into account by the software system. The receiving of a message by a process is an example of internal event. The triggering of an interrupt by a sensor is an example of external event. An event is generally associated to a connector.

An *entity*, or passive object, corresponds to a *data* element, which has a state that can be modified by active objects. For example, the speed of a train is a variable (entity) that can be modified by a controller (agent).

A *relation* corresponds to another type of *data* element that links two or more other objects and that can have additional attributes. An example of relation data is a data structure whose attributes are the type of train, its

current speed and its maximum speed (additional attribute).

A *goal* is a constraint on one or more of the components of a software system. Additional components may be derived to satisfy a non-functional goal. An example of a constraint deriving from a goal is that a component of the software system of an ATM has to check if the password typed by the user matches the password associated in the system to the ATM card inserted.

3.2 The Architecture Prescription Language

Appendix A shows an abstract example of the refinement tree for the goals (on the left), and of the refinement tree for the corresponding architecture prescription components (on the right). As the example shows, the trees don't have the same shape. It would be a pure coincidence if they did have it.

The goal refinement tree is obtained as we explained in section 2. All the refinements are pure "and" apart from the refinement of goal G1. G1 is obtained by achieving requirement R1.1 and either requirement R1.2 or goal G1.1 (the arch between R1.2 and G1.1 denotes an "or" refinement). The sub-goals/requirements refining goal Gi are denoted as Gi.j, with j varying from 1 to the number of sub-goals/requirements. We use an analogous notation for the subcomponents.

In the component refinement tree, the root component C is the software system itself. The software system is viewed as a component of the bigger system that may include hardware devices, mechanical devices and human operators. We want to note here that also for these other kinds of systems we could design an architecture prescription language. Ours, anyway, is tailored to the software sub-system. The first refinement of C is obtained by considering the components directly derived by the KAOS specification by using the methodology we explained in section 3.1. Note that we may provide further refinements or even redo existing refinements due to non-functional requirements such as performance and reliability or from reusability considerations.

Exhibit 3. shows how the APL describes all the attributes of the components in the refinement tree. Please note that the Composed of relationship is the only one that can be deduced directly from the tree.

Component C:
KAOS spec.: S
Type: Software System
Constraints: R1.1, (R1.2 or (R1.3.1, R1.3.2)), R2.1, R3.1, R3.2
Composed of: C1, C2, C3, C4
Uses: /

Component C1:
KAOS spec.: S
Type: Processing
Constraints: R1.1, R3.1a
Composed of: C1.1, C1.2, C1.3
Uses: C2 *to interact with* {C3}

Component C2:
KAOS spec.: S
Type: Connecting
Constraints: R3.1b
Composed of: C2.1, C2.2
Events: E1, E2, E3
Uses: /

Component C3:
KAOS spec.: S
Type: Data
Constraints: R1.2, R2.1
Composed of:/
Uses: C2 *to interact with* {C1, C4}

Component C4:
KAOS spec.: S
Type: Processing
Constraints: R1.1, R3.2
Composed of: C4.1, C4.2
Uses: C2 *to interact with* {C3}

Component C1.1:
KAOS spec.: S
Type: Processing
Constraints: R1.1
Composed of:/
Uses:/

Component C1.2:
KAOS spec.: S
Type: Connector
Constraints: R3.1a
Composed of:/
Uses: C2 *to interact with* {C3}

Component C1.3:
KAOS spec.: S
Type: Processing
Constraints: R1.1
Composed of:/
Uses:/

Component C2.1:
KAOS spec.: S
Type: Data
Constraints: R3.1b.1

Composed of:/
Events: E1, E2
Uses:/

Component C2.2:
KAOS spec.: S
Type: Processing
Constraints: R3.1b.2
Composed of:/
Events: E2, E3
Uses:/

Component C4.1:
KAOS spec.: S
Type: Processing
Constraints: R1.1
Composed of: /
Uses: C2 *to interact with* {C3}

Component C4.1:
KAOS spec.: S
Type: Data
Constraints: R3.2
Composed of: /
Uses: C2 *to interact with* {C3}

Exhibit 3. APL prescriptions

The attribute *KAOS spec.* denotes the specification from which the component is derived. In our example we called this specification S.

Type specifies the type of the component. The possible types for components are: *Software System*, *Processing*, *Connecting* and *Data*.

Constraints is the most important attribute of a component. It denotes which requirements the component satisfies. For example, the root component C, i.e. the software system, must achieve all the goals. The subcomponents in the first layer of the tree, instead, have to satisfy only a subset of the system requirements. The union of the requirements achieved by the leaves components is the complete set of requirements.

Also, a component may be only contributing in achieving a goal without being able to achieve it alone. This may happen in case of non-functional requirement such as security. When a component cannot achieve a requirement only by itself, we represent it in our prescription language by appending a different lower case letter to the name of that requirement in each of the components involved in achieving it. In our example, this happens with C1 and C2. In order to achieve goal R3.1, goals R3.1a and R3.1b have to be achieved by C1 and C2 respectively.

The same requirement can be achieved by more than one software component. One reason for such a redundancy might be a reliability goal; another reason might be that the achievement of a goal may best be done cooperatively. In refinements successive to the first one (in which all the components are directly derived from the requirements specification) the constraints themselves can be further refined in order to better allocate them to different subcomponents. In the next paragraph we'll explain the reasons for such subcomponents. So, it may happen what we show in our abstract example. In our example C2.1, a subcomponent of C2, whose constraints are requirements R1 and R3.1.b, has as constraint requirement R3.1b.1 (R3.1b.1 is one of the two sub-requirements that and-refine R3.1b). The other subcomponent of C2, C2.2 has R3b.2 as constraint.

Composed of identifies the subcomponents that implement the component. The subcomponents of the root component are obtained directly from the KAOS specification. The subcomponents in the next layers of the components refinement tree are designed by the software architect in order to make the software system achieve other desirable characteristics, such as better performance or greater reusability of the components (even across different domain). For example, a component directly derived from the KAOS specification might be too big; it could have too many requirements as constraints. If the component implements many requirements many users/software components might use it. This would lower the system performance. Also, reducing the constraints on a component will make it easier to modify the component in case one of the requirements is removed or changes during the software development process. To achieve this purpose the component could be split into many subcomponents that have fewer requirements or even only a part of a requirement. As we said in the previous paragraph, some requirements might be further refined after the specification phase, even though they are already directly achievable by some agents. Having to satisfy only a sub-requirement may make the subcomponent more reusable. For example, a requirement on a component for an intelligent house software system might have to take into account both inputs from a smoke detector and a heat sensor to detect a fire. Even though the requirement can be directly achieved by a single component, to make the fire manager components more portable (to a system that has a smoke sensor only, for example) as well as better maintainable, we can split the requirement into smoke detection and heat detection sub-requirements and assign them to different components.

The attribute *Events*, generally assigned to connector components, indicates the events the component has to handle.

The last attribute, *Uses*, indicates what are the components used by the component. Since interactions always happen through a connector, the *Uses* attribute has

the optional keyword *to interact with* that indicates which components the component interacts with using that connector.

4. An Architecture Prescription for the Meeting-Planning Problem

Now, we will know show how to obtain an architecture prescription in practice. For this purpose we will consider the meeting-planning problem. At the highest abstraction level there are two goals that every meeting planner has to achieve. They are (in KAOS notation):

Achieve[MeetingRequestSatisfied]
Maximize[NumberOfParticipants]

We already showed the specification of the first goal (exhibit 1.). From this goal specification we obtain three agents one of which is software: Scheduler. From the sub-goal Achieve[SchedulerAvailable] we deduce that the root component of the architecture prescription must be parent also of other two components: SchedulerManager and MConnector. Scheduler Manager finds an available scheduler, communicating to the existing schedulers by MConnector, and in case no Scheduler is available it builds a new one. We call the root component MeetingWizard.

The second goal translates in an additional constraint on Scheduler.

Without going into the details of the KAOS specification for the meeting planner, some of which are in [1] and [2], in exhibit 6 we show some of the components of the meeting planner architecture prescription that illustrate many of the characteristics we discussed before.

Component MeetingWizard:
KAOS spec.: MeetingPlanner
Type: Software System
Constraints: {the complete set of requirements}
Composed of: Scheduler, SchedulerManager, MConnector
Uses:/

Component Scheduler:
KAOS spec.: MeetingPlanner
Type: Processing
Constraints: {the complete set of requirements} \ Achieve[SchedulerAvailable]
Composed of: PlanningEngine, ParticipantClient, MeetingInitiatorClient, ResourcesAvailableRepository, SecureConnector1, SecureConnector2
Uses:/

Component SchedulerManager:
KAOS spec.: MeetingPlanner
Type: Processing
Constraints: Achieve[SchedulerAvailable]
Composed of:/
Uses: Scheduler, MConnector *to interact with* {Scheduler}

Component PlanningEngine:
KAOS spec.: MeetingPlanner
Type: Processing
Constraints: {subset of the set of requirements}
Composed of: Planner, Optimizer
Uses: SecureConnector *to interact with* {ParticipantClient, MeetingInitiatorClient}

Exhibit 6. APL sample prescription for the meeting planner

5. Conclusion

In this paper we have illustrated the advantages of formal techniques to go from a phase of the software development process to its next one. We focused on what we consider the most important of these transitions: the one from requirements to architecture. To make a formal transition between these two phases easier we have introduced an architecture prescription language (APL), that specifies the structure of the software system and its components in the language of the application domain. This higher-level architecture specification can be then easily translated, if necessary, in an architecture description, in the solution domain. We took advantage of the characteristics of KAOS as a requirements specification language.

Other researchers in the past have tried to find techniques to pass from requirements to architecture. Nenad Medvidovic et al., in [4], developed a technique to pass from requirements specified in WinWin to an architectural model for the system. Their technique, while providing a framework to pass from requirements to architecture, is not formal and still leaves a lot of choices to the architects. This due in part the big gap between the requirements specification, specified in the problem domain language, and architectural design, described in the solution domain language. Other researchers have designed object-oriented techniques to pass from requirements to architecture. These techniques, though,

are still very informal with little guidance for the architect on to decompose the architecture into classes and which attributes and methods to assign to those classes. Furthermore, this approach is tailored to an object oriented design.

Our goal is to design a formal technique to pass from the requirements to an architecture prescription that can be refined afterwards. The formality is necessary to make it sure that none of the requirements are neglected, and that we don't introduce any useless component or constraint. The generality of our approach allows the architects to choose their favorite ADL (architecture description language) to describe an architecture prescribed in APL.

Future areas of our work will include: define and perform experiments that tests our method, further research or even redefine the components of the APL to achieve non functional properties such as better performance and reusability, and build supporting tools that take the requirements for a software system and some other parameters and transform them into an architecture prescription for the system.

6. References:

[1] Anne Dardenne, Axel van Lamweerde and Stephen Fickas, "Goal-directed Requirements Acquisition", Science of Computer Programming, Vol.20, 1993, pp. 3-50

[2] Axel van Lamweerde, R. Darimont, and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", IEEE Transactions on Software Engineering, IEEE Computer Society, November 1998, pp. 908-925.

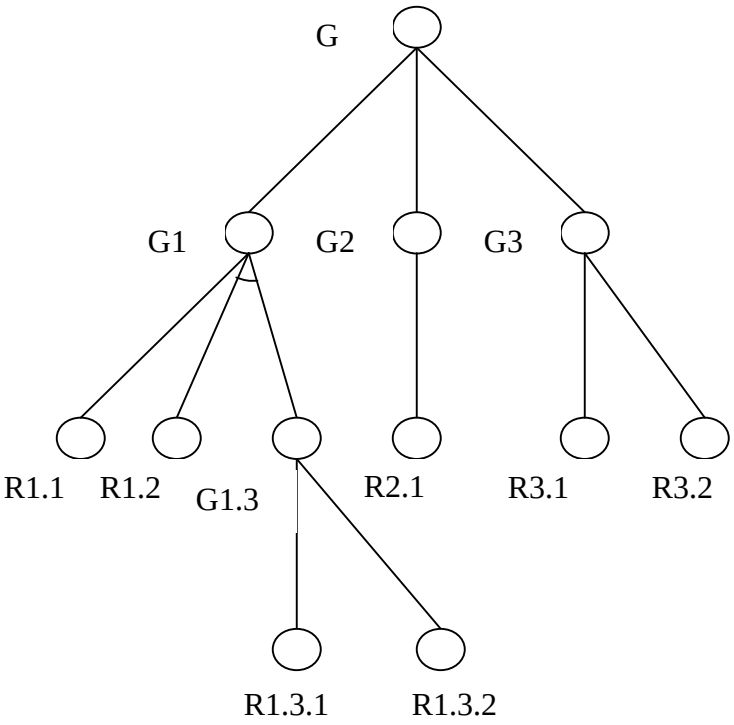
[3] Dewayne E. Perry, Alexander L. Wolf, "Foundations for the Study of Software Architecture", Software Engineering Notes, ACM SIGSOFT, October 1992, pp. 40-52

[4] Nenad Medvidovic, Paul Gruenbacher, Alexander F. Egyed, and Barry W. Boehm, "Bridging Models across the Software Lifecycle", Technical Report USC-CSE-2000-521, University of Southern California

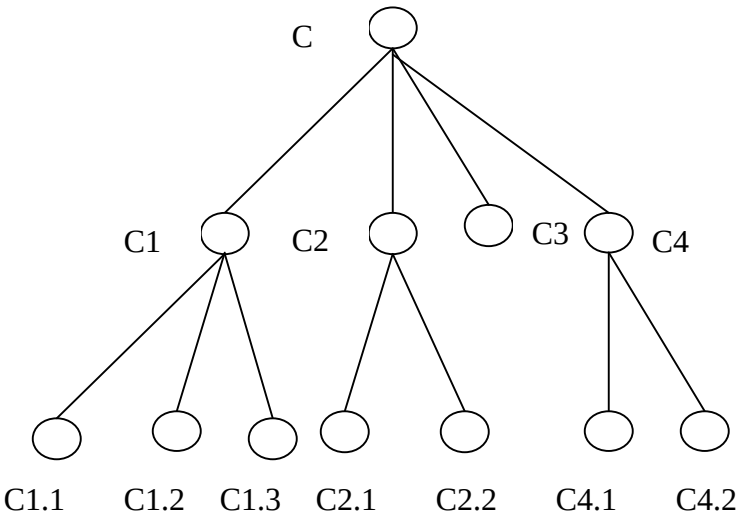
[5] Axel van Lamsweerde, Robert Darimont, and Philippe Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt", Proceedings RE'95 – 2nd IEEE Symposium on Requirements Engineering, York, March 1995, pp. 194-203

Appendix A.

Goal Refinement Tree



Component Refinement Tree



Checking consistency between architectural models using SPIN

Paola Inverardi & Henry Muccini & Patrizio Pelliccione
Dipartimento di Matematica
Universitá 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

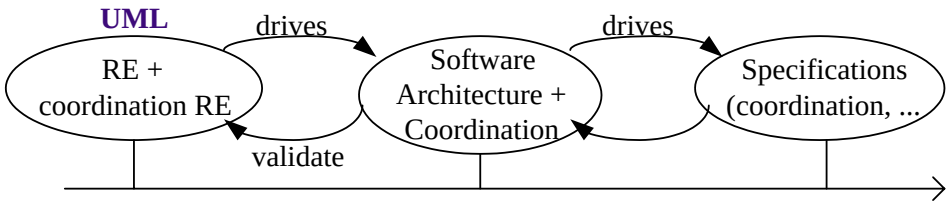


Figure 1. Requirements, Software Architectures and Coordination models

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

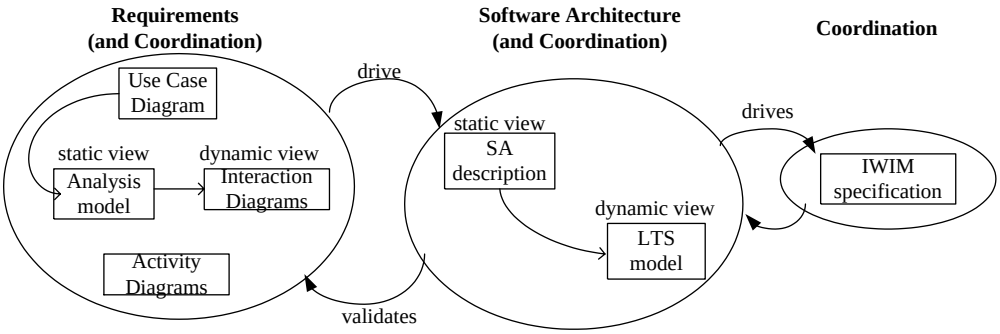


Figure 2. Coordination Requirements and SA: The Approach

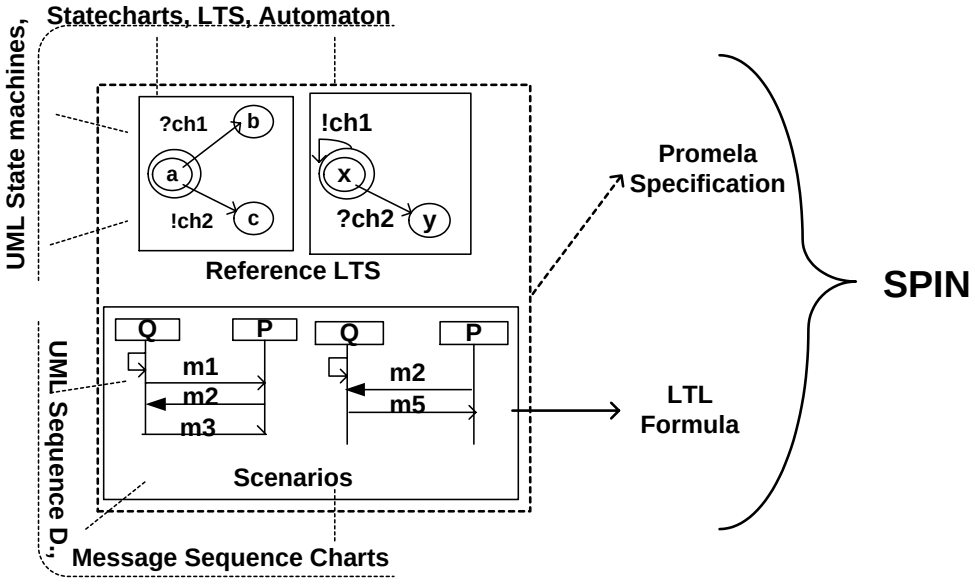


Figure 3. Consistency Checking between models: The Approach

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

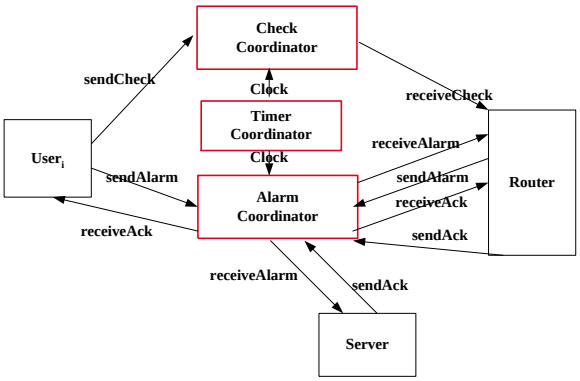


Figure 4. The TRMCS SA topology

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

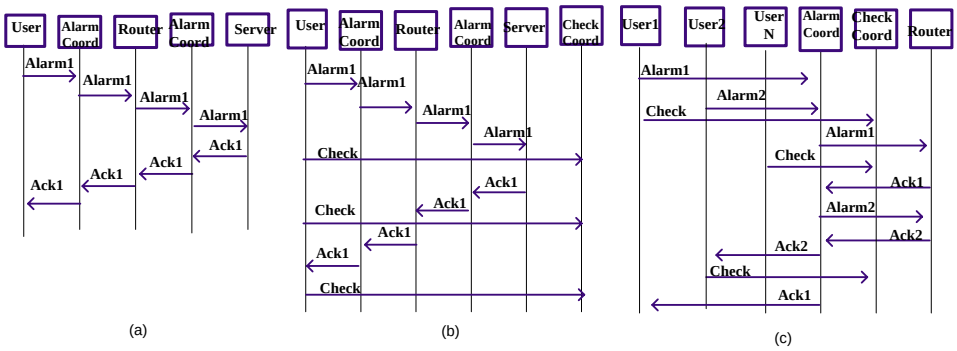


Figure 5. Coordination Requirements at the SA level of abstraction

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 Assuarance Systems Enginering, 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 Verifyng 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.jtmpl>.

[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>.