



Scaling New Heights

STRAW'03
Second International
Software Requirements to
Architectures Workshop

ICSE'03
International Conference on Software Engineering
Portland, Oregon
May 3-11, 2003

Table of Contents

Foreword by the Workshop Co-Chairs.....	iii
Daniel M. Berry, Rick Kazman, Roel Wieringa	
From Architectures to Requirements and Realizations	
Architectural Requirements Engineering: Theory vs. Practice	1
Robert W. Schwanke	
From Architecture to Requirements: A Success Story	9
Pamela Zave	
Architecture-Level Requirements Specification	14
Davor Svetinovic	
Architecture-Based Design of Computer Based Systems.....	20
Mark Denford, Tim O’Neill, John Leaney	
Patterns Approach to Building Software Systems	28
Satish Chandra, Satyendra Bhattaram	
From Requirements to Architectures	
Experience with Global Analysis: A Practical Method for Analyzing Factors that Influence Software Architectures.....	34
Robert L. Nord, Dilip Soni	
Tool Support for Scenario-Based Architecture Evaluation	41
Steffen Thiel, Andreas Hein, Heiner Engelhardt	
Experiences from the Architectural Change Process.....	46
Josef Nedstam, Even-André Karlsson, Martin Höst	
Pattern Oriented Software Development: Moving Seamlessly from Requirements to Architecture	54
M. S. Rajasree , P. Jithendra Kumar Reddy, D. Janakiram	
Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery	61
Nenad Medvidović, Alexander Egyed, Paul Gruenbacher	
Mapping Requirements to Software Architecture by Feature-Orientation.....	69
Dongyun Liu, Hong Mei	
Architecture as an Emergent Property of Requirements Integration.....	77
R. Geoff Dromey	
Detailing Architectural Design in the Tropos Methodology	85
Carla T. L. L. Silva, Jaelson F. B. Castro, John Mylopoulos	
Eliciting Architectural Decisions from Requirements using a Rule-based Framework.....	94
WenQian Liu, Steve Easterbrook	

Requirements, Architectures and Risks	100
James D. Kiper, Martin S. Feather	
From Goal-Oriented Requirements to Architectural Prescriptions: The Preskriptor Process	107
Manuel Brandozzi, Dewayne E. Perry	
Integrating Organizational Requirements and Socio-Intentional Architectural Styles	114
Lúcia R. D. Bastos, Jaelson F. B. Castro, John Mylopoulos	
Moving from Quality Attribute Requirements to Architectural Decisions	122
Felix Bachmann, Len Bass, Mark Klein	
Using Problem Frames with Distributed Architectures: A Case for Cardinality on Interfaces.....	130
Charles B. Haley	
Towards a Systems Engineering Pattern Language: Applying <i>i*</i> to Model Requirements-Architecture Patterns	134
Pete Pavan, Neil A.M. Maiden, Xiaohong Zhu	
Requirements–Architecture Integration	
An Experience-Based Approach for Integrating Architecture and Requirements Engineering.....	142
Barbara Paech, Antje Von Knethen, Joerg Doerr, Joachim Bayer, Daniel Kerkow, Ronny Kolb, Adam Trendowicz, Teade Punter, Allen Dutoit	
A Framework for Managing Traceability Relationships between Requirements and Architectures	150
Susanne A. Sherba, Kenneth M. Anderson	

STRAW '03

Second International Software Requirements to Architectures Workshop

Foreword by the Workshop Co-Chairs

Over the past 10 years, software requirements engineering and software architecture have been the topic of fastly growing research disciplines. Requirements engineering has seen the advent of

- goal-oriented approaches,
 - scenario-based requirements engineering,
 - sociology- and linguistics-based techniques, and
 - formal techniques
- for identifying and specifying requirements.

Architecture design has seen the advent of

- patterns research,
- architectural style research,
- attribute-based architecture design,
- architecture description languages,
- component-based approaches, and
- product-line architectures.

There is a clear relationship between requirements engineering and architecture design. However, for the most part, the two disciplines have evolved independently from each other, and promising areas of mutual interest remain to be explored. For example, an important type of design research consists of relating classes of problems to classes of solutions. In software engineering, there are interesting connections between software problem patterns and software solution patterns. Recent research in problem frames could therefore be extended by including architecture patterns and investigating relationships between the two kinds of patterns.

The patterns paradigm may be extended by including the wider business context, consisting of business processes, actors, and strategies. In this wider context, the problem is one of alignment of software architecture with business architecture. Here, domain knowledge may be codified using reference architectures.

A third area of potential fruitful interaction is that of component-based development. Assembling components into a system requires an architecture that mediates between the system requirements and the requirements on the components. More generally, when we extend our view from a single system to a hierarchy of systems, the

interplay between requirements and architectures is a central guiding principle in system design.

The goal of the Second International Software Requirements to Architectures Workshop (STRAW'03), to be held in Portland, Oregon, U.S.A. in conjunction with the 2003 International Conference on Software Engineering, is to bring together researchers from the requirements engineering and architecture communities to exchange views and results that are of mutual interest, and to discuss topics for further research. Topics of interest include, but are not limited to:

- deriving architecture descriptions in concert with requirements specification,
- attribute-based architecture design,
- tracing architectural decisions to requirements,
- evolving architectures and requirements,
- alignment between software architecture and business architecture,
- relating architecture patterns to requirements patterns,
- reference architectures,
- reuse of requirements and architectures,
- systems engineering approaches,
- formal foundations of the requirements–architecture relationship,
- requirements and architecture specification languages, and
- tools and environments for requirements engineers and software architects.

Participation in the workshop is limited to 30 people and is based upon the submitted papers, the best of which will be presented. All submitted papers will be distributed to the participants before the workshop starts and are included in these proceedings. Each presented paper will be assigned an discussant, who will lead the discussion about the paper. During the day, participants are expected to propose issues to be discussed at the end of the day. The workshop is expected to lead to the generation of several lists, of issues discussed, of disagreements identified, of conclusions reached, and of topics to be further researched.

It is important to remember that a workshop is not a formal conference. Rather, it is focused on discussion of current, on-going, and possibly incomplete work. Moreover, the papers appearing in these proceedings are position papers submitted both to suggest topics for discussion and to indicate the authors' interests in the subject of the workshop. They are not to be considered formal publications, and they may be sent in the future to more formal avenues of publication. Consequently, the organizing committee, in consultation with the program committee, decided to accept all papers submitted for publication in the proceedings. The organizing committee then selected a small number of these papers for presentation at the workshop. This small number would allow for more workshop-style discussion without the pressure to cover a whole lot of papers, each with only a short period for presentation and an even shorter period for questions and discussion.

The papers in these proceedings are divided into three groups,

1. about moving from architectures to requirements and realizations,
2. about moving from requirements to architectures, and
3. about requirements–architecture integration.

All papers, counterpoints, and discussion summaries will be made available electronically at the workshop website at

<http://se.uwaterloo.ca/~straw03/> soon after the workshop. The site aims to highlight outstanding issues that should be the focus of future research in the area.

This second workshop follows the first workshop of the same name, that was held in Toronto, Ontario, Canada, in conjunction with the 2001 International Conference on Software Engineering (ICSE'01). That workshop's website is

<http://www.cin.ufpe.br/~straw01/>.

We extend our thanks to all those who have participated in the organization of this workshop, particularly to the program committee who helped to design the call for papers and to solicit submissions and who provided advice to the organizing committee. They are:

Jaelson Castro, Brazil
Anthony Finkelstein, UK
Jaap Gordijn, Netherlands
Carlo Ghezzi, Italy
Manuel Kolp, Belgium
Jeff Kramer, UK
Axel van Lamsweerde, Belgium
Jeff Magee, UK
John Mylopoulos, Canada
Bashar Nuseibeh, UK
Dewayne Perry, USA

Finally, we hope that you will enjoy these proceedings.

Sincerely,

Daniel M. Berry, Canada,
Rick Kazman, USA, and
Roel Wieringa, the Netherlands
the Organizing Committee

Architectural Requirements Engineering: Theory vs. Practice

Robert W. Schwanke
Siemens Corporate Research, Inc.
robert.schwanke@scr.siemens.com

Abstract

This paper discusses how architectural requirements engineering fits into an overall software development process in the concept and definition phases of a project. It defines a reference process identifying the “ideal” artifacts and their interrelationships, describes some key technical activities that are useful for producing these artifacts, and captures some practical experience in commercial projects.

1. Introduction

Theory and practice are generally the same, in theory.
– Anonymous

This paper is an attempt to reduce the wide gap that so often occurs between the theory and practice of architecture requirements engineering in real software development projects. Too frequently, an organization fails to capitalize on a good software architecture, for reasons such as: the development process is not aligned to profit from it; the key stakeholders do not buy into it; or, it simply solves the wrong problem.

The “theory” aspect of this paper offers a reference process for architecture requirements engineering and related activities. The artifacts and dependencies are foremost in the process definition, because (in practice) most software analysis and design activities are artifact-driven and opportunistically scheduled, so modeling the data of the process gives more insight than trying to model control. These artifacts are sequenced within a simple phase-and-gate framework that shows the phases and decision points where the project can be cancelled, sent for rework, or approved to enter the next phase.

The central artifact, for the purposes of this paper, is the Global Analysis document, first introduced by Hofmeister, Nord and Soni [1]. The software architects at Siemens Corporate Research have used Global Analysis in half a dozen projects since the book was written. This paper gives a brief review of the approach, updated based on our experiences.

The “practice” aspect of this paper offers hints on doing software architecture effectively and efficiently. Doing it effectively means building stakeholder consensus and buy-in for both the technical design and the development plan, by obtaining agreement on the requirements and other constraints that they must satisfy, and convincing people that the design and the plan do satisfy the requirements and constraints. Doing it efficiently means focusing attention and other resources on the important issues, at the right times, while tracking, but living with, a large number of less-important inconsistencies, unsatisfied constraints, and other unknowns.

The “theory” and “practice” aspects are intermingled in the presentation, in hopes of reducing that gap.

2. An Architecture-Centered Process

The architecture group at Siemens Corporate Research provides technical and project management consulting services to a wide variety of software development groups within Siemens (primarily in the United States but occasionally in Europe). The process described here is our starting point: how we would like to do architecture if we could. Naturally, every real project has constraints that prevent this, such as the legacy process used previously, the legacy artifacts providing input to the project, and the skills and comfort zones of the key players. After presenting this idealized process we will discuss some of the adaptations that may be necessary to use it in a real project. (Hereafter, the word “we” usually refers either to the SCR architecture team or to the team and the readers of this paper, depending on context. “I” refers to the author.)

The process definition has four major parts: the artifacts produced, the dependencies between artifacts, the phases of the project, and the rules for coordinating artifacts. It does not specify any activities separately from artifacts, other than reviews, because most activities are artifact driven, anyway, and best discussed in the context of the artifacts they produce. This process definition also

does not describe how to assign artifacts to teams and how to coordinate teams; that would take another whole paper.

For brevity, this process description only covers the parts of the process most related to architecture requirements engineering. It assumes that the project has already completed its “idea phase”, and sufficient resources have been allocated to carry out the concept phase. It also assumes that the project is predominantly a software development project, and therefore does not address hardware design, manufacturing, or separate “system” artifacts. The principles presented here certainly apply to such systems, but would require a longer treatment.

3. Artifacts, dependencies, and activities

Figure 1 shows the artifacts and dependencies of the process. Each arrow represents a dependency: “ $X \rightarrow Y$ ” means “information in artifact X depends on (or is justified by) information in artifact Y.” Typically, each individual item in X, such as a specification, is annotated with references to specific items in Y, such as requirements.

Although many of the artifacts are familiar to the reader, a few comments are in order.

3.1. Stakeholder list

A stakeholder is an *accessible* person who *represents* a class of persons who will be significantly affected by architectural decisions. The stakeholder must be accessible to the project team to answer questions and review artifacts. Sometimes the stakeholder is a member of the class (e.g. a testing manager can speak for all testers), but sometimes he is appointed to represent the class from outside (e.g. a marketing analyst who speaks for the end user.) Every such class should be considered for representation, including such diverse classes as salespersons, buyers, end users (could be multiple classes), software testers, installers, trainers, and help desk attendants.

The stakeholder list clarifies exactly who cares about the project, why they care, and why that matters. As the list develops, it may go through several refinement steps. The first draft might identify all the candidate stakeholder classes, with stakeholder names, where known, and explanations of why each class is important. As the list stabilizes, the classes without named stakeholders become action items, either to find stakeholders or to explain why the class is not important enough to be represented. Later on, the list may also prioritize stakeholders or define different groups of stakeholders, typically for allocating stakeholders to artifact reviews.

If an organization has a well-developed business process model, showing all the actors in the product’s target business domain, many of these actors will require stakeholders to represent them. However, since such business process models are still uncommon in current practice, this software process does not assume that such an artifact exists.

3.2. Stakeholder requests

Stakeholder requests document the concerns of stakeholders. Some stakeholders produce artifacts that are defined in the company software process; others just write white papers, send e-mails, and attend reviews. For the purpose of this process definition, we assume that any input from a stakeholder can be documented as a stakeholder request. Since a stakeholder could request almost anything, we are usually only interested in *qualified* stakeholder requests, which have been reviewed and approved as being worth addressing.

Most stakeholders are “outside” the architecture team. The chief architect and the project manager are often also stakeholders. However, their requests should be qualified by someone outside the project, so that they do not appear to abuse their right to write requests.

3.3. Features

Requirements, in general, define properties of the product, in terms that external (outside the development project) stakeholders recognize and understand. Features are requirements at a coarse granularity, suitable for use in sales presentations and for allocating to product releases (in the Build/Release Plan). A feature could be a specific service that the product provides, but it could also be an attribute of the whole product, such as “fault-tolerant.”

The Features artifact should specify which stakeholders’ interests it represents. Often, it is limited to customer stakeholders, and becomes the “voice of the customer.” Eventually, each feature should be annotated with references to qualified stakeholder requests that justify the feature.

This process avoids using the terms “functional” and “non-functional” to characterize requirements and specifications, because these terms mean different things to different people.

3.4. Detailed requirements

Detailed requirements spell out what the feature level requirements mean in terms that are testable, but still in the stakeholders’ language. We often find that 15-30 detailed requirements are needed per feature, to be

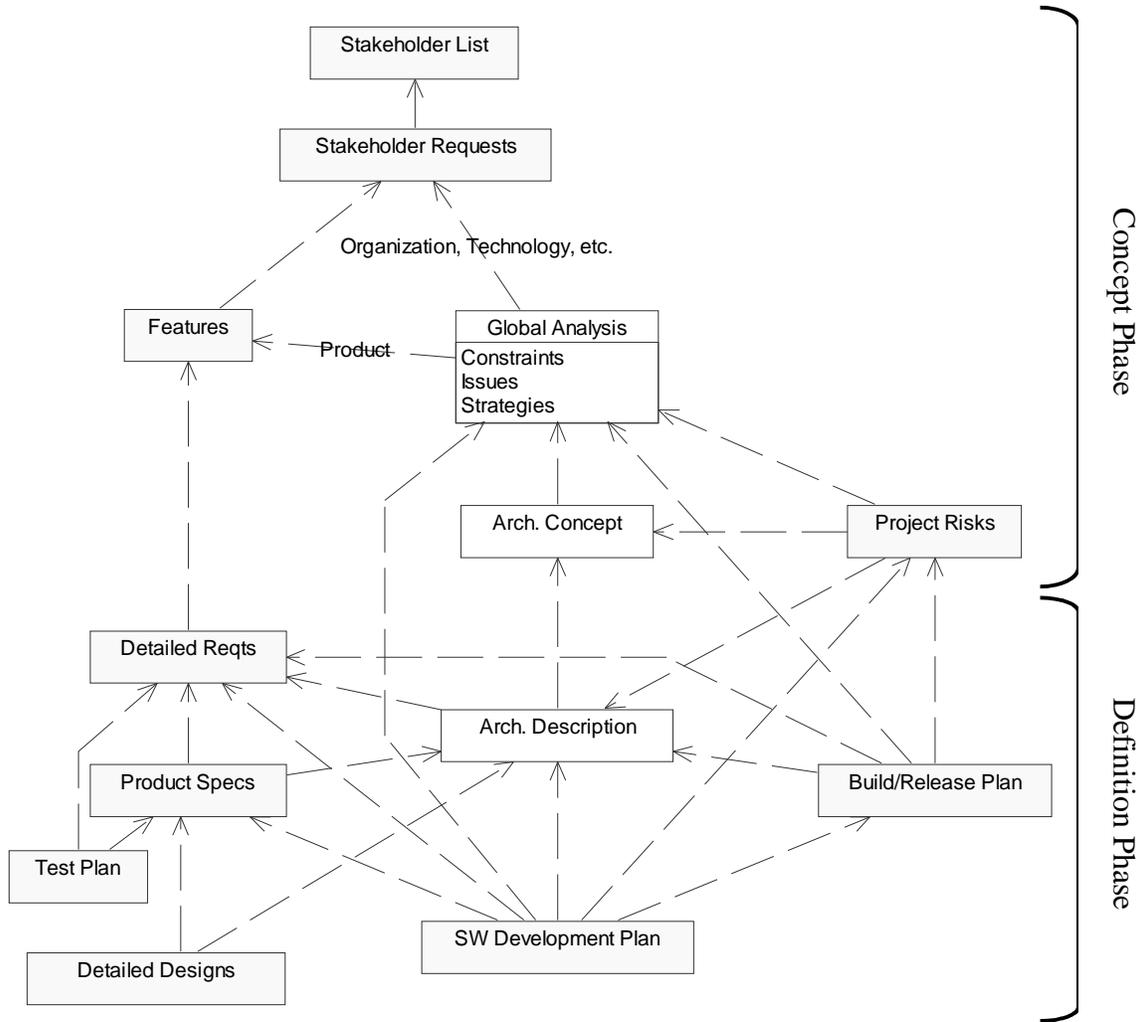


Figure 1. An architecture-centered process

complete and unambiguous. A single detailed requirement can support several features.

For user interface requirements, a UI prototype is strongly recommended, to capture both the intent and the details of features, particularly aesthetic features like “easy to use” and “common look and feel”. Eventually, the prototype can be captured in a conventional requirements artifact by copying screen shots into it, with accompanying text and models (e.g. state transition diagrams or message sequence charts) to nail down exactly what the product should do.

Some projects do not need both detailed requirements and product specifications. (Cf. Section 6.4)

3.5. Product specifications

Specifications define properties of the product and its parts, in technical terms that the developers, testers, documenters, project engineers, maintainers, and other “downstream” stakeholders understand. We often observe an expansion factor of 2-5 between detailed requirements and product specifications. A key, *theoretical* difference between detailed requirements and product specifications is that a requirement should state what the product should do, without reference to any particular implementation, whereas the product specification describes the externally-visible properties of the externally-visible interfaces identified in the architecture description.

When requirements and specifications are written by different teams, the product specifications may represent “push back” by the development team, conveying the message, “We heard what you said you wanted, but this is what we think we can build.”

3.6. Feasibility and global analysis

The architecture team has the responsibility to analyze everything that may affect the success of the project, determine what the critical issues are, propose strategies to address those issues, and then develop an architecture consistent with this analysis. It is called “global” analysis because it looks at the project from all directions (from the perspectives of all the stakeholders), and because the critical issues and strategies are typically also global, cutting across subsystem boundaries and appearing in more than one view of the architecture. The global analysis artifact contains three kinds of items: factors, issues, and strategies.

3.6.1. Factors. A *factor* is any fact that is likely to constrain or otherwise influence the architecture. Some factors can be written as requirements, but others cannot be so rigorously stated. Normally, we expect requirements to state properties of the product, and to be correct, unambiguous, and testable, whereas a factor is often unverified, ambiguous, or uncertain, and may describe something other than the product itself. Furthermore, we usually use a stylized language to write requirements, e.g. “The product shall cost no more than \$300 per floating license per year.” Imposing stylized language on factors would interfere with communication. For example, is it clearer to write, “The product shall be developed using programmers whose previous experience does not include ASP technology” or “Our programmers don’t know ASP”? The first alternative is verbose, passive voice, vague, and might actually be incorrect, if there is an option to hire a few ASP developers. The second alternative succinctly captures one fact that constrains the architecture.

Factors can come from anywhere. For convenience they are grouped into three categories: product factors (typically derived from features); technology factors, which involve the technologies available to implement the product; and, organizational factors, which involve properties of the company or other organization that is developing the product. These categories are further grouped into sub-categories, such as product performance, services provided, programming tools, technical standards, staff skills, schedule constraints, and so on. These categories and sub-categories should not be considered exhaustive; any stakeholder request might draw attention to a significant factor, whether or not it fits neatly into one of the categories.

A factor should have a standard structure. We typically record the following properties:

Category and sub-category

Name

Unique ID

Brief statement of the factor.

Flexibility (what sort of “wiggle room” is there in the factor today?)

Changeability (how might it change later on?)

Impact (how does it affect the architecture?)

Authority (what or who justifies this constraint?)

Owner (person responsible for text of this factor)

Status

Priority

Previously, we have tried to capture each factor as a row in a table of factors, but found several practical problems: the columns became too narrow, there was lots of white space, cross-referencing the factor name or number was awkward, and our usual word processing tool didn’t handle word-level change tracking very well in tables. So, I recommend organizing the factors into categories and subcategories, giving each constraint its own sub-sub-section within its sub-category, and using a standard text structure within the subsection. Figure 2 suggests a format.

1. Organizational Constraints

1.3 Management

1.3.5 Buy reporting subsystem

(Factor-37)

The reporting subsystem should be based on a commercial product, e.g. Crystal Reports

Flexibility. Previous reporting system was implemented in-house, so buying COTS is not a rigid requirement. But competitors are already doing this.

Changeability. Reporting features may become more specialized, making the “buy” option less advantageous.

Impact. Buying the market leading product has low development cost, risk, and time to market, but introduces licensing costs and reduces product differentiation.

Authority: Features 135, 136, and 139, and SR 174 from Jim Smith, who has interviewed customers concerning reporting features.

Figure 2. Textual presentation of a constraint

Although storing factors in an ordinary text document is often practical, we are also considering using a

requirements management tool to manage architecture factors, but have no experience yet.

Even when a factor is written in the form of an architecture requirement, there are two important differences between a marketing feature and an architecture factor: range and uncertainty. The range of the architecture factor is a way of capturing a set of similar requirements that vary only in certain dimensions. For example, “The architecture must support customers with transaction rates between 1 million and 100 million per day.” This factor does not say that any particular customer installation has to perform well across the whole range, nor does it even say that any particular release of the product has to handle the whole range. For example, there could be two variants of the product for low-volume and for high-volume customers.

The dimensions that factors frequently span include numerical ranges, members of a product family, successive generations of a product or family, and ranges of calendar time. For example, “In four years, the architecture must support GUIs on handheld devices.” This allows the architect to choose between designing the infrastructure for handheld GUIs now, or leaving a placeholder for them and designing them in two years, by which time the technology will have changed anyway. Note that variation over calendar time is different from the “stability” of a feature or a factor. In the example above, the factor is very stable, but is chronologically positioned four years in the future.

Allowing uncertainty in an architecture factor allows the architect to document the problem before the uncertainty is resolved. For example, now that Internet services are beginning to be offered aboard airplanes, the marketing department might envision the day when radiologists, traveling on airplanes, want to download medical images to their laptops. A factor might be written, “the product must be evolvable to support medical image viewing over low-bandwidth, high-latency Internet connections.” Such a statement would normally be disqualified as a requirement, because “evolvable” is a vague word. But, such statements are valuable to the architect, despite their vagueness. Note that “uncertainty” is also somewhat different from “stability”, because the statement of the factor explicitly captures the sense in which it is uncertain, whereas calling a requirement “unstable” has more to do with its status.

It may be useful to describe range and uncertainty as separate attributes of a factor, but we haven’t tried it yet.

Unlike requirements catalogs, the collection of architecture factors does not have to be complete. Global analysis prioritizes them, finds conflicts and tradeoffs between them, and finally reduces them to a set of key issues that shape the architecture. The less important factors will likely be ignored, for most purposes, so missing a few of them is not important.

3.6.2. Issues. An architectural issue is a potential conflict or tradeoff between two or more factors – usually many more! For example, the issue “Aggressive Schedule” might be stated as, “The project probably can’t be completed in 14 months if we have to train our programmers in Java, add new tools to our development environment, and implement all 75 major features, 7 of which require exploratory prototyping.” Normally, there are many potentially significant issues, but certain ones rapidly emerge as the most critical. Fortunately, because of the inherent uncertainty of many factors, it is not necessary to satisfy all of them. The architects must identify and prioritize the issues, so that the architecture and the project development plan can be designed to address the most critical ones. The others are managed as project risks, to be addressed later.

3.6.3. Strategies. A strategy is a decision that addresses one or more significant issues. The strategy may be technical, managerial, or a combination. For example, if the issue is “ASP programming is best done in Java, but our programmers only know C++”, the architect and project manager could choose to “retrain our programmers in JSP”, “buy an ASP development environment for C++”, or “use some C++ programmers to write C++ applets, and retrain others to write JSP.”

3.6.4. Putting it all together. The original description of Global Analysis[1] suggested using “Issue Cards”, where each card defines and discusses one issue, then defines and discusses strategies for addressing it. This approach doesn’t work well when a strategy addresses several issues – which many of them do. Instead, I recommend documenting issues and strategies by embedding them in a coherent presentation of the rationale for the architecture. The first part of the Global Analysis artifact should be a catalog of factors, as described above. The second part should present the significant issues and strategies for resolving them. Each issue should be documented in a format that is partly structured and partly informal. The structured part includes backward references to the most relevant factors and references to the most important strategies for dealing with the issue. Each strategy could be defined at the first place it is referenced in the text, perhaps in a sidebar or an inset box. The informal part of the issue description discusses how the factors interact to shape the issue, and how the proposed strategies would help to resolve the issue. The third part of Global Analysis should be a free-flowing, coherent rationale for the proposed architectural approach. This presentation technique emphasizes coherent argumentation more than cataloging and cross-referencing the issues and strategies, as we have sometimes done in the past.

3.7. Architecture concept

This artifact should not be confused with the conceptual view of the architecture. The architecture concept artifact is written for external stakeholders, is informal, and presents the essential concepts of the architecture in notations and words that are comfortable for the stakeholders. It is typically based on a paper “proof-of-concept”, which describes a slice of the system using the proposed architecture approach. It then uses portions of this system slice to illustrate the concepts it presents, depending on what is needed to educate and convince the stakeholders

3.8. Architecture description

This artifact is the complete description of the architecture, typically following the IEEE standard 1471-2000. Note that the architecture description depends on the detailed requirements, but the architecture concept does not. This is so because (a) the architecture concept should not be sensitive to small changes in requirements, and (b) the architecture concept usually needs to be relatively complete, reviewed, and approved before authorizing the expense of developing detailed requirements.

3.9. Project risks

This process does not specify how project risks are described and managed, but many risks are identified in the course of global analysis and architecture design. Any key issues that are not fully resolved by the strategies, as well as any major assumptions made while drafting the architecture description, become risks that must be managed.

3.10. Build Plan and Release Plan

The build plan defines a sequence of internal development milestones, or builds, with each module, product specification, and detailed requirement to be implemented in a specified build. We typically recommend that the individual builds be scheduled about 6 weeks apart, to provide rapid feedback on the effectiveness of the design and maintain a common understanding of the system across the development team. Some of the builds are designated as releases that the customer will see (although perhaps only as a demo).

3.11. Software development plan

The software development plan depends on the global analysis artifact for strategies and on the architecture description as the basis for a bottom-up cost estimate. At SCR we use an estimation methodology that annotates the module view of the architecture with development cost estimates, collecting the assumptions needed to make those estimates. The modules become tasks in the plan; the assumptions become risks to be managed. For more on architecture-centric software project management, see Paulish’s book of that title [2].

4. Project phases

Figure 1 divides the artifacts into two phases: the concept phase and the definition phase. This division signifies the phase in which each artifact receives its first critical review and sign-off. Of course, each artifact is revised in subsequent phases, as needed.

5. Coordinating artifacts and activities

Other than at the end of each phase, the process does not specify an order in which the artifacts are finished and reviewed, because this ordering varies widely between projects, depending on many “soft” factors. Instead, we expect that the artifacts will be written by different people, and will therefore evolve concurrently. In order to manage this efficiently, it is important to identify where information provided in one artifact is used in another, and to cross-review artifacts between teams. It is equally important to allow, but document and manage, incompleteness and inconsistency between artifacts.

5.1. Incompleteness and inconsistency

Recording incomplete links is especially valuable in the global analysis artifact. It is true that, *eventually*, every issue should be based on factors, and that those factors should derive their authority from other artifacts. However, global analysis frequently identifies potentially significant factors long before the relevant stakeholders have raised concerns about them. Rather than waiting to document the factor until the stakeholder writes a request, the architect should put a note in the authority field of the factor, describing where he expects the authority will come from. The note could even include a shortened draft of the item (e.g. a feature) that he would like to see added to some other artifact. (If necessary, the architect might have to write his own stakeholder request.) Similar techniques should be used wherever links between artifacts may appear. (Incomplete links are very much like

the “fat references” used in the Pattern Languages of Programming community.)

5.2. Cross-reviewing artifacts between teams

One of the most important heuristics for effective artifact review is, “Choose reviewers who depend on the information they are reviewing.” In this process, the dependency links between artifacts are an excellent guide for identifying reviewers. Consider, for example, the detailed requirements. The people who wrote the features (if different) will want to be sure that the detailed requirements accurately define the features. The people who will be writing product specifications will want to make sure they receive good-quality detailed requirements, to make their job easier. The people who have to write tests against the detailed requirements will want to be sure the requirements are testable.

Using the dependency links to identify reviewers also reduces the chances of “disconnect” in a project. Many of us have experienced projects where artifacts were “thrown over the wall” from one group to another, leaving both groups dissatisfied. Having such a wall between requirements engineering and development, for example, tempts developers to ignore the requirements they don’t understand or don’t like. By using cross-reviewing to strengthen communication and buy-in between teams, such problems can be reduced.

5.3. Reviewing links between artifacts

Whenever an artifact is formally reviewed, the links between it and other artifacts should also be reviewed. This includes both the artifacts on which it depends, and the artifacts that depend on it. *This is very important for building consensus!* When a requirements engineer signs off on the global analysis artifact, his signature should mean that, except for noted defects, (a) all relevant, previously documented features have been referenced in the right places in the analysis, (b) any relevant, not-yet-documented features have been discussed and given incomplete references in the analysis, and (c) *he agrees with the analysis of these features*. On the other side, when the global analyst signs off on the Features artifact, his signature means that every feature needed to justify significant factors, whether or not they have been published yet, appears either in the artifact itself or the review notes.

The review notes then become action items for resolving incomplete and inconsistent links. However, the resolution does not necessarily need to happen immediately. Some of the items may be very low priority, some may require further investigation, and some may not be resolvable until a later stage of the work.

5.4. Validation and Consistency

Each significant item in each artifact, such as a feature, an issue, or a specification, is subject to validation in the course of review. Part of the definition of consistency between artifacts is that a link from an item in artifact X to an item in artifact Y is only fully consistent when the item in artifact Y has been validated. Sometimes the validation is simply a yes/no decision on whether the item should be included in the artifact; in other cases, included items are further assigned to “buckets” that represent different development/release cycles. In the latter case, of course, the bucket assignments of X and Y must be compatible.

5.5. Phase reviews

At the end of each phase there is a review, often called a *gate*, whereby managers outside the project determine whether to continue funding the project. There are actually two separate questions to answer: “Is the project ready to move into the next phase?” and “Is the company ready to pay for it?” Some organizations actually have two separate reviews, because some of the decision-makers are different for these two questions.

Each phase review specifies the artifacts that will be considered at the review. In this process, each artifact is considered at each phase review after its introduction, if it is relevant to the decision. Naturally, these artifacts must have been reviewed individually prior to the phase review. However, they don’t have to be absolutely complete and consistent, as long as there is an action plan for resolving the inconsistencies.

This approach to handling incompleteness and inconsistency is especially valuable when the development organization is undergoing change to adapt to new or improved development processes. Often artifacts cannot be completed and reviewed in the same order as the chain of dependencies. Because the show must go on, explicitly documenting incompleteness and inconsistency for later resolution is often the best approach.

6. Merging the Processes

Because there is little standardization of software development processes across organizations, the process defined above will normally have to be adapted for use in the context of an organization’s existing process. This section describes some of the adaptations that are likely to be necessary, and some of the issues that may need resolving.

6.1. Enriching the concept phase

Many existing processes focus mainly on defining product features in the concept phase. If possible, one should insist on doing some feasibility analysis in the concept phase, before committing the resources necessary to do a complete high-level design. This feasibility analysis would then include global analysis and the architecture concept, as well as a UI prototype if the product has a user interface.

6.2. Regrouping information in artifacts

Sometimes it is necessary to combine logically separate artifacts into a single artifact, or, for reasons of scale, to divide a single logical artifact into a main artifact and several subsidiary artifacts. However, it can also be necessary to redefine an existing artifact so that it carries more architecture information than it has in the past.

For example, a process may define a “System Concept” artifact, typically due at the end of the concept phase, which has historically been a very informal document. This might be a good place to put the Architecture Concept.

6.3. Caring for stakeholders

Many existing processes do not address all the important stakeholders. For example, a Market Requirements artifact might be limited to addressing the logical functionality of the product, ignoring non-functional features. This typically arises from a focus on end-users, ignoring the needs of other stakeholders like system administrators, buyers, and commissioning engineers. The remedy might be to add another artifact to carry non-functional features, or to address quality attributes in the Global Analysis artifact.

More generally, the process should be adapted so that every important stakeholder has a “voice” in some artifact – in Global Analysis, if not elsewhere.

6.4. Detailed requirements vs. specifications

Although in theory there is a clear logical distinction between a detailed requirement and a product specification, in practice the two artifacts are frequently combined. We have found several reasons for this:

- Cost pressure: maintaining two descriptions of strongly related information is more expensive than maintaining one.
- Skill shortage: good requirements engineers are under-appreciated, and therefore in short supply!

- Process: without an architecture description, the only input to the product specification is the detailed requirements, anyway, so why not combine them?
- Disconnect: because of inadequate communication between those who write features and those who write specifications, it is not obvious that the detailed requirements are missing.
- Difficulty: it is actually quite difficult, in many instances, to write a good set of detailed requirements without referring to implementations, especially early in the definition phase when so many questions are unsettled.

One way often suggested to overcome these difficulties is to introduce a prototype, often as a controlled process artifact, whose purpose is to facilitate consensus-building between requirements analysts and developers. The most common types, of course, are the UI prototype and the proof-of-concept prototype. The detailed requirements and product specifications do not need to be written down until the prototype stabilizes and is reviewed. Then, both artifacts can be derived from it, if both are needed.

7. Future Work

We are currently investigating how to extend our process to effectively use rigorous models for domain analysis, requirements analysis, design, and testing.

Acknowledgements

The process diagram in Figure 1 was produced by rapid iteration based on feedback from my colleagues, each of whom contributed different expertise: project management (Dan Paulish), problem statements (David Laurance), architectural concept and consensus building (Dilip Soni), global analysis (Bill Sherman and Rod Nord), requirements engineering (Brian Berenbach), and user interface design (Nuray Aykin).

References

- [1] Hofmeister, C., R. Nord, and D. Soni, *Applied Software Architecture*, Boston: Addison-Wesley, 2000.
- [2] Paulish, D. *Architecture Centric Software Project Management: A Practical Guide*, Boston: Addison-Wesley, 2002.

From Architecture to Requirements: A Success Story

Pamela Zave
AT&T Laboratories—Research
Florham Park, New Jersey, USA
pamela@research.att.com

Abstract

User requirements for telecommunication systems are difficult to understand because they are obscured by a long history of ad hoc feature development and technological limitations. The presence of a viable modular architecture for telecommunication features gives us a fresh start. Working within this framework, we can discover desirable properties that ought to be requirements for all telecommunication systems.

1 Introduction

This paper tells a story. By the end of the story there is a close relationship between requirements and architecture in the telecommunication domain. The recognition of true requirements developed from the architecture, however, rather than shaping the architecture as might be expected.

Although this appears to be a success story, it is far from over, and there is much work yet to be done.

2 No telecommunication requirements

The life of the Public Switched Telephone Network (PSTN) began in 1875. Since the 1960s telephone switches have been controlled by software, which has enabled and encouraged addition of a steady stream of features of all kinds.

In this paper, “requirements” refers to user requirements, descriptions of the behavior of the system as a user observes it. Requirements concerning performance, reliability, resources, administration, *etc.*, are not discussed.

Beyond the most basic requirement of allowing people to talk to each other at a distance, the PSTN seems to have no requirements, or at least no requirements in the sense of desirable properties that are globally satisfied. Every desirable property one can think of has many exceptions. To give a

simple but rich example, consider this property: If the subscriber owning address x subscribes to a feature that blocks all calls to or from address y , then the owner of x is never talking to the owner of y through the telephone network.

This property might not be satisfied because x and y are associated with devices, and at least one of their owners is using a different device. This is typical of the ambiguity we see everywhere in the PSTN—addresses identify many things, but never what we really want to identify, which is people.

Alternatively, the property might not be satisfied because of an interaction among features. For example, the owner of y might cooperate with the owner of z so that calls to z are forwarded to x . If the forwarding feature sets the source of the call to z at the same time that it sets the target of the call to x (which is the most common behavior), and if x is not blocking calls from z , then the owner of y can call z and be connected to the owner of x . In this case the behavior of the call forwarding feature subverts the intention of the blocking feature.

The property could also be violated by interaction with a large-scale conference feature. Such conferences have their own addresses; participants can join the conference by calling the conference address, or can choose ahead of time to be called by the conference. Either way, the blocking feature of x cannot prevent its owner from joining a conference in which the owner of y is also a participant.

The lack of *satisfied* requirements in the PSTN is not surprising, given its long history, incremental development, technological limitations, and geographical and administrative distribution. There is also, however, a near-complete lack of *understood and agreed upon* requirements, whether satisfied or not. Simply put, we do not know how telecommunication systems should behave.

Beyond the obstacles to requirements already mentioned, the goals of subscribers often conflict, and there is no consensus about how to balance them. In addition, many people appear to believe that a telecommunication system should behave toward each subscriber exactly as that subscriber might wish during every moment of his life, without

acknowledging the impossible complexity of reaching such a goal.

There is now an industry-wide trend away from circuit-switched networks, and toward packet-switched IP networks. This change is removing many technological limitations, but it is not getting us any closer to understanding requirements for telecommunications. On the contrary, the IP community is much less aware of the issues than the PSTN community is. There is an unfortunate “Internet boom” arrogance that leads newer entrants in the telecommunications arena to believe that they have nothing to learn from the past.

3 The feature-interaction problem in telecommunications

As features are added to telecommunication software, they interact with old features, often in subtle, unpredictable, or disastrous ways. This *feature-interaction problem* makes telecommunication software extremely expensive to develop. PSTN software is not unreliable, but only because reliability is so important that switch manufacturers take heroic measures to ensure that failures are contained.

The feature-interaction problem has been recognized since the late 1970s, and there has been quite a bit of research attempting to solve it [2, 3, 4, 5, 9].

Ultimately, to manage feature interactions properly we need to understand what they are, prevent the bad ones, and enable the good ones. Unfortunately, distinguishing the bad ones from the good ones depends on having requirements for desirable global behavior, which is just what we do not have.

In the shorter term, there is plenty of value to being able to add and change telecommunication features easily, in a way that is modular and guaranteed not to break the system. This is a huge improvement over adding features by patching monolithic code, with all its attendant difficulties and dangers, even if feature interactions can still cause behavior that is undesirable to users. This short-term goal has been reached with an architectural approach.

4 An architecture for telecommunication services

Distributed Feature Composition (DFC) is a component architecture for telecommunication services [7, 8]. It was designed for feature modularity, feature composition, structured feature interaction, and generality within the telecommunication domain.

In DFC a request for telecommunication service is satisfied by a *usage*, which is a dynamically assembled graph of

boxes (components) and *internal calls*. A *box* is a concurrent process providing either interface functions (an *interface box*) or feature functions (a *feature box*). An *internal call* is a featureless, point-to-point connection with a two-way signaling channel and any number of media channels.

The fundamental concept of DFC is pipe-and-filter modularity [10]. Each feature box behaves transparently when its functions are not needed. Each feature box has the autonomy to carry out its functions when they are called for; it can place, receive, or tear down internal calls, it can generate, absorb, or propagate signals traveling on the signaling channels of the internal calls, and it can process or transmit media streams traveling on the media channels of the internal calls. A feature box interacts with other features only through its internal calls, yet does not know what is at the far ends of its internal calls. Thus each feature box is largely context-independent; feature boxes can easily be added, deleted, and changed.

In DFC there are exactly two mechanisms for feature interaction (or *component coordination*, to use a more architectural term). One is the signaling through internal calls, which is governed by the DFC protocol. The other is the DFC routing algorithm, which routes each internal call to a box, thus determining the configuration of boxes in each usage as it grows, shrinks, and reshapes itself. Feature boxes can influence the routing in specific ways, which is how routing becomes a mechanism for feature interaction.

DFC has been notably successful at reaching its goals. It is not easy to say how one could reproduce the success in another application domain, but here are a few observations that seem relevant:

- Michael Jackson and I began work on DFC at the beginning of 1997. At that time one or both of us had been studying the telecommunication domain, on and off, since 1982. Trying to tame the complexity of feature interactions, we had run up seemingly every possible blind alley.
- We were completely content to be domain-specific; we had no interest in any other domain, believing that telecommunications was more than enough challenge for us. More general applications of the ideas in DFC are just now beginning to emerge.
- In important ways, DFC is low-level: it is close to the true building blocks of telecommunication implementations. This accounts for its generality.
- At the same time, DFC is abstract enough to be formally defined in a few pages. This makes the application of formal methods to DFC tractable.

Since 1997 we have made a number of changes to the original DFC architecture [7]; these are documented in the manual [8]. Some changes are refinements, while others extend DFC to cover aspects of telecommunications not originally considered, for example multimedia. Nevertheless,

the central ideas of the original architecture are still present and essentially unchanged.

5 Experience with the architecture

Since 1999 we have been implementing and exploiting DFC within AT&T Research. Our BoxOS system [1] is an IP implementation of DFC with excellent interoperation capabilities; for example, it can be packaged as a SIP application server.

We have used this environment to create interesting voice-over-IP services. In 2002 alone we implemented features for personal mobility, mid-call movement from one device to another, switching, small-scale conferencing, transferring, augmenting a telephone with a graphical user interface on a nearby personal computer, call logging, voice mail, speed dialing, click-to-dial, voice signaling, reaching a representative of a group, and large prescheduled conferences.

This rapid feature development creates relentless pressure to understand feature interactions better. When a feature developer is faced with a seemingly arbitrary choice of feature behavior, he wants to know the consequences of each choice. Which choice will cause the feature to interact best with all other features, present and future?

Although the pressure to understand feature interactions returned us to the seemingly hopeless problem of discovering the requirements for telecommunications, we returned to it with some additional weapons in the arsenal. The improvement was due to the presence of a viable architecture. Because of the architecture, we could implement features quickly and plan ambitiously; this gave us a broader base of knowledge about features, how they can interact, and what people are trying to use them for. Also because of the architecture, we had a tractable formal framework in which to reason, without loss of generality, about features and their interactions.

6 Example: call forwarding

As an example of the subtleties of telecommunication behavior, let's return to the example of call forwarding as introduced in Section 2. When the owner of y calls z and the features of z forward the call to x by changing its target address to x , should the source address of the call also be changed to z ?

Most forwarding features make the source change, both in telephony and in electronic mail [6]. If some error occurs in attempting to reach the new target address x , then the error should be reported to the features or owner of z , which know about x . If the error is reported to the features or owner of y , the result may be confusion or a violation of

privacy, since y may know nothing about x . There is also a vague concern that if the source address is not changed and no trace is left of the role of z , there might be security problems.

On the other hand, changing the source address during forwarding has negative consequences. The source address is no longer a reliable indication of who the callee will be talking to when he answers the call, which is why the blocking feature is undermined. Also, x may have features that automatically return a received call, by placing a new call to its source address, under various circumstances. If one of these features is activated while z is still forwarding its incoming calls to x , then a forwarding loop will be created.

It might seem attractive to solve this problem by maintaining a complete address history within the signals of the call protocol, rather than just two addresses. Unfortunately, this also has many negative consequences.

Because address histories can grow quite long, they place a heavy burden on the infrastructure. In fact, the voice-over-IP protocol SIP, which maintains an address history for reasons other than the ones discussed here, is causing implementation problems due to very long headers. Address histories do not interoperate well with the existing telecommunication infrastructure, all of which is based on calls with two addresses.

Equally important, address histories can violate privacy. Consider a physician calling patients from home. He has a feature that allows him to change the source address of his calls to his office address. A complete address history would reveal to patients the address of his home telephone, which is the original source of each call. Yet the physician has a legitimate right to keep this information private.

7 Ideal address translation

It should be clear by now that if we accept the telecommunication domain as it exists today, the call-forwarding problem has no solution. Any choice we make about its behavior violates some desirable property that should be a global requirement.

One way out of this dilemma is to concentrate on an ideal version of telecommunications in which there are no legacy constraints, and both the infrastructure and the features behave in the right way. This gives us the freedom necessary to figure out what the right way might be.

Address translation is the function performed by a feature when it changes the source or target address of a call. Call forwarding performs address translation, as do many (perhaps even most) other features. For the feature interactions caused by address translation, a search for the ideal has succeeded, yielding two important and highly intertwined results [11]:

- Requirements that a telecommunication system should satisfy.
- Constraints on the infrastructure and on feature behavior that guarantee satisfaction of the requirements without sacrificing functionality.

The constraints on the infrastructure are architectural, and are inspired by DFC.

It is outside the scope of this paper to present the principles of ideal address translation. As a substitute, here is a brief, informal explanation of how the conflicts of the previous section can be resolved.

In the recommended infrastructure, a call is implemented by a chain of requests, feature modules, and interface modules as shown in Figure 1. Each request has source and target addresses. The chain has a *source region* in which there are (optional) feature modules associated with source addresses. For example, $s1$ might be the address of the physician's home telephone, and $s2$ might be the physician's office address. The source feature module of $s1$ changes the source address of the call to $s2$ at the physician's request.

The source region is followed by a *target region* in which there are (optional) feature modules associated with target addresses. For example, $t2$ might be z , and the target feature module of $t2$ might do call forwarding by changing the target address from $t2$ to $t1$.

There is an *authenticity* requirement that the source address of a request chain should reveal to the callee the entity at the other end of the call. Call forwarding is not changing the source of the call in any way, and therefore must not change the source address. If it does, the authenticity requirement will certainly be violated.

This constraint on the behavior of call forwarding and other target-region features is necessary but not sufficient for authenticity. For example, an unauthorized person might use the physician's home telephone, or might even program the features of his own telephone to set the source address of the call to $s2$!

The authenticity of $s2$ as a source address can only be secured if the source feature module of $s2$ contains an authentication feature that demands a password or other proof of identity. The infrastructure guarantees that any request chain containing $s2$ as a source address must pass through this module and therefore be subject to authentication.

There is a *reversibility* requirement that a target-region feature or targeted user should be able to call the source address of a request chain and thereby target the entity at the source of the request chain. Clearly this is another reason why call forwarding must not change the source address.

In this example, the source address $s2$ that reaches the target region identifies the physician in his role as a physician. It is more abstract than $s1$, which is only the address of a particular device. This is why the reversibility requirement is stated in terms of "the entity at the source of the

request chain" rather than "the device at the source of the request chain."

In the formal definition of the reversibility requirement, an abstract address such as $s2$ is considered to identify a truer source of the request chain than a concrete address such as $s1$. This has important consequences. If a patient misses the physician's call and calls back later, the physician may no longer be at home, and the $s1$ address would not reach him. However, his role address $s2$ can subscribe to a location feature that will locate him wherever he is now.

There is a *privacy* requirement that use of a more abstract address such as $s2$ effectively conceals a more concrete address such as $s1$. This requirement is also guaranteed by the constraints in [11].

Note that privacy and authenticity balance the conflicting goals of knowing and concealing. The effect of privacy is a person can conceal an address that he owns behind another address that he owns. The effect of authenticity is that an address can only be used by a person who owns it.

Errors are signaled back through the request chain. So if target address $t1$ turns out to be unknown, then the error signal will first reach the target feature module of $t2$, which should handle the error if possible, and conceal $t1$ if necessary.

8 Future work

There are many other areas of feature behavior and feature interaction besides address translation. It is important to attack them with the same weapons, in the hopes that they, also, will yield their secrets.

The infrastructure that supports ideal address translation is generally similar to all telecommunication protocols in use today. At the same time, it is different in crucial ways from all of them except DFC as implemented in BoxOS. So a gap has been opened between theory and practice that must be bridged in some way. This will require the utmost creativity, pragmatism, and patience.

Even though current telecommunication systems fall short of satisfying them, the requirements discovered so far are simple, compelling, and convincing. They would have been discovered long ago, except for the complications of a long history that has made them as difficult to see as to satisfy.

9 Acknowledgments

No success would have been possible without the many contributions of my colleagues Greg Bond, Eric Cheung, Healfdene Goguen, Karrie Hanson, Don Henderson, Michael Jackson, Gerald Karam, Hal Purdy, Chris Raming, and Tom Smith.

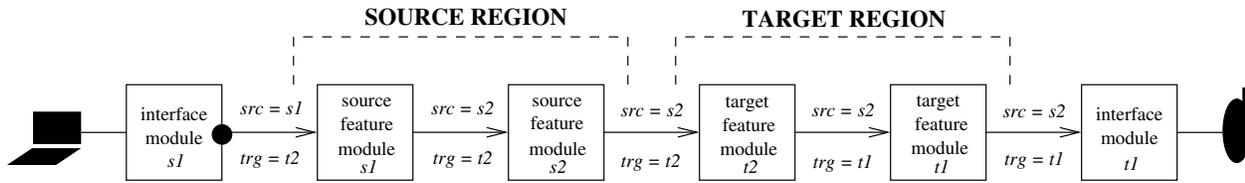


Figure 1. A request chain.

References

- [1] Gregory W. Bond, Eric Cheung, K. Hal Purdy, J. Christopher Ramming, and Pamela Zave. An open architecture for next-generation telecommunication service. Submitted for publication, 2002.
- [2] L. G. Bouma and H. Velthuisen, editors. *Feature Interactions in Telecommunications Systems*. IOS Press, Amsterdam, 1994.
- [3] M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI*, IOS Press, Amsterdam, 2000.
- [4] K. E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems III.*, IOS Press, Amsterdam, 1995.
- [5] P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interactions in Telecommunication Networks IV*. IOS Press, Amsterdam, 1997.
- [6] Robert J. Hall. Feature interactions in electronic mail. In [3], pages 67-82.
- [7] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunication services. *IEEE Transactions on Software Engineering* XXIV(10):831-847, October 1998.
- [8] Michael Jackson and Pamela Zave. The DFC Manual. AT&T Research Technical Report, August 2001. Available at <http://www.research.att.com/info/pamela>.
- [9] K. Kimbler and L. G. Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press, Amsterdam, 1998.
- [10] Mary Shaw and David Garlan. *Software Architecture*. Prentice-Hall, 1996.
- [11] Pamela Zave. Ideal address translation: Principles, properties, and applications. In *Feature Interactions in Telecommunications and Software Systems VII*, IOS Press, Amsterdam, 2003, to appear.

Architecture-Level Requirements Specification

Davor Svetinovic
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
dsvetinovic@uwaterloo.ca

Abstract

The large gap in the levels at which requirements are specified results in inadequate means for ensuring that business goals are properly supported. Architecture-level requirements specifications help us reduce this problem by providing necessary constructs and traceability mechanisms. Enhancing traditional requirements engineering approaches by incorporating architecture-level requirement specifications will facilitate business goals satisfaction and simplify the design of appropriate software architectures.

1. Introduction

Since its early days, software development has been implementation driven. Programming, still considered by many as the most important and difficult development activity, has attracted most of the research attention over time. While sufficient in some cases, programming has become a relatively routine activity compared to the other development activities in the development of today's large, complex, and constantly changing software systems. The main difficulty in today's development is not anymore *how* to build the system, but *what* to build and how to make it as adaptable to future change as possible [6].

Because of its early importance, implementation technologies and paradigms have influenced all development stages, even the early ones such as requirements analysis and design. For example, structured and object-oriented programming paradigms resulted in structured [12, 31, 30] and object-oriented analysis and design techniques [21, 9, 4]. This tradition continues with emergence of new methodologies such as aspect-oriented analysis [1], which has its origin in aspect-oriented programming paradigm.

The success of such approaches was mostly due to the fact that the traditional way of development focused on one product at a time [19]. A clear product-level requirements specification combined with low-level design, us-

ing structured or object-oriented concepts, was appropriate for a product development in relatively stable and well-understood problem domains.

Domain-level requirements analysis and specification appeared as a solution to a need for building software systems for large, difficult to understand, and changing problem domains. Goal-driven requirements engineering emerged as a leading approach for dealing with domain-level requirements for large systems [10, 23, 29]. The main emphasis of this approach was on making sure that software actually fulfills business goals. This goal fulfillment problem was one of the main weaknesses of the traditional product-level requirements engineering approach.

Now, with the emergence of new economic trends, the Internet as a business medium, software as a commodity, *etc.*, even small systems have become much more difficult to build and maintain. New software paradigms and technologies such as web services, agility, and product lines, emerged to solve this new wave of problems. In this new situation, both business systems and software systems change faster than ever before. Naturally, both domain and product-level requirements specifications become obsolete very quickly, in some cases even before the product is built [17, 19].

2. Agility, Web Services, and Product Lines

In this section I would like to emphasize the commonalities of agile development paradigms, web services, and product lines as related to requirements specification. Even though all three concepts seem to have contradictory goals, they do share and contribute many new common development principles.

First, they deemphasize product-level requirements specifications. The agile development philosophy states that a detailed up-front specification of the product level requirements is unnecessary [18, 3]. Rather, agile followers believe that product-level requirements are best discovered on the fly, *i.e.*, by developers who directly communi-

cate with clients and implement these features without documenting them or preserving their rationale. Web services elevate responsibility for product-level requirements from the application developer to the service provider. Application developers can choose and integrate many different services which vary in particular feature details. A product line stresses the development of several product at a time. The main focus of the product line development is on the establishment of a robust architecture that will support many kinds of different product-level variations. In all three cases, we have a shift from product-level thinking to a whole new way of thinking, which allows us to think in terms of future changes and how we can accommodate them as easily as possible.

Second, all three approaches take into account the constant change in the problem domain. This is similar to the changes at the product-level.

So in summary, now we have the situation that software systems have to adapt to constantly changing problem domains, and also to be easily adaptable to new problem domains. This leads us to the following main constraints that requirement specifications should satisfy:

- Requirements artifacts should be reusable and easily modified, *i.e.*, they should be *assets*.
- The focus should be on the clear separation between commonalities and variabilities, from the requirements perspective.
- Domain and product-level requirements should be secondary as they describe mostly variations in functionality, and low-level requirements should be left to developers.
- We should emphasize stable, change resistant requirements with a high architectural impact.

3. Requirement Abstraction Levels

Requirements are specified either directly or indirectly for many different purposes, and as part of many different engineering activities. For our purposes, we can sort them according to different levels at which they usually appear:

1. Business-level requirements specification: Business-level requirements are most often indirectly specified as the part of business reengineering activities [17, 11, 2, 27]. The most common concepts that appear at this specification level are business goals, processes, resources, and rules. It can be argued that this is probably the most important type of requirements specifications, as the goal of software systems is to ultimately satisfy and contribute to the fulfilment of these business processes, goals, *etc.*

2. Domain-level requirements specification: As mentioned previously, domain-level requirements are one of the traditional approaches to the requirements specification [10]. Newer, more systematic versions of domain-level requirements engineering have received a lot of attention recently [5, 7, 24]. Most of its applications are in the area of business systems, which are getting increasingly complex and difficult to adequately support by software systems [8, 16, 15]. The most common concepts that appear at this specification level are user goals, user tasks, domain input, domain output, *etc.* More recent trend is the incorporation of agent-based analysis as the part of domain modelling [26, 20, 25, 14].

3. Product-level requirements specification: Product-level requirement specifications are the most common type of requirement specifications. There exists an extensive body of knowledge about them, and most of the previous research focused on perfecting different techniques used to elicitate, specify, and validate this type of requirements. The most common artifacts and concepts that occur as the part of product-level specifications are features, use-cases, functional lists, data input, data output, *etc.*

4. Design-level requirements specification: This is another type of well understood and widely used type of specification. A lot of efforts were invested into its standardization through Unified Modelling Language (UML) [21, 13]. UML artifacts present the most common types of concepts and techniques used to capture requirements at this level. This level acts as a transition phase between product-level specification and code-level requirement specifications.

5. Code-level requirements specification: Lastly, usually considered as a part of programming activity, low-level algorithm and data structure specification makes what we refer to as the code-level requirements specification. This is the type of the specification which most programmers are familiar with, as it is inseparable part of coding. It focuses mostly on the implementation related issues and constraints. This is also probably the best understood requirements specification level.

From this discussion, we can observe that most of the current forms of requirement specifications focus on the specification of functionality at the different levels. This leads us to the definition of the problem I am aiming to solve.

4. Problem Statement

While new development technologies and paradigms stress structure and quality over changeable functionality, traditional requirements engineering techniques still focus on primarily capturing the low-level functionality of the system. Structure and quality requirements are often deemphasized and hidden within specifications. The requirement engineering artifacts must be adapted to support this new development reality and improve the return on investment in all possible ways. Therefore, the problem that we are dealing with is: *How should we organize and specify requirements in such way to emphasize structure, quality, and stable requirements, and at the same time provide a way for capturing changeable and variable requirements?*

In addition, as change is occurring in both, business system and supporting software system, we have to perform the analysis and specify structural and quality requirements of both systems. A software system has to be adaptable to support also several different business systems, and to allow the evolution of all of them.

In my opinion, the most promising way to deal with these issues at design, implementation, and maintenance stages of software development cycle is the effective use of software architecture principles and techniques. Nevertheless, the effectiveness of software architecture techniques, especially when one has to develop multiple architectures at the same time, is in my opinion limited, as they are based on requirement specifications which are tailored to emphasize different issues such as low-level functionality, one product focus, *etc.* The goal is to try to solve this problem by introduction of architecture-level requirements specifications.

5. Proposed Solution

The hypothesis is that architecture-level requirements specification provides more support for the development of software systems using web services, agility principles, and product lines, than traditional domain and product level-requirements specifications. This support reflects through an improved architecture for the system, clear identification of common structural elements and functionality, and identification of variation points and constraints on the future evolution of the system. Also, architecture-level specification lies conceptually between domain and product-level specifications, allowing clear definition and verification of the mechanisms through which product features help achieve the business goals. Providing this traceability is identified as one of the most important requirement engineering problems [22].

Therefore, my work will focus on the definition of different requirement specification levels, together with the analysis and adaptation of different requirement specification

techniques and artifacts to these levels. In particular, I will define a set of techniques and artifacts that can be used to capture architecture-level requirements. These include an architecture-level requirements specification method, which is based on the the focus shift from product and domain to their architectural properties, integration, and qualities.

6. Architecture-Level Use Cases

One of the already identified uses of architecture-level requirement specifications is the architecture recovery of software systems [28]. I successfully performed extraction and specification of architecture-level requirements in the form of *architecture-level use cases*.

Use cases — that is, narrative descriptions of domain processes — appear in different forms in all phases of a development cycle. They are typically used as the artifacts around which development cycles are organized. When used this way, all other activities and artifacts depend on them.

A use case describes the interactions between actors and system processes. A use case encapsulates responsibilities that are performed during a computation by actors and by system processes.

Architecture-level use cases are use cases that describe *logical processes* within the system. In my study, these use cases were not created by developers, but were generated using high-level responsibilities that were written as the part of the code documentation. The purpose of this generation was to document dynamic processes within the system. This was a technique used as the part of the logical architecture view in order to present dynamic interactions in a comprehensible format.

Requirements were discovered and abstracted from the method-level to the subsystem level. While module-level responsibilities provide a compact way to encapsulate and represent architecturally significant features, method-level responsibilities are used to understand and present module and subsystem interactions using architecture-level use cases. The advantage of architecture-level use cases over other presentations like sequence diagrams is that they present dynamic aspects in a comprehensible way while hiding low-level details.

Architecture-level use cases were built using navigational capabilities of several code-browsing tools in conjunction with documented responsibilities. The main value of this approach was not in a documentation of all possible use cases, but in an ability to recover them as needed. Although responsibilities were not required to be documented within source code, the advantage of having them documented there is that a transition from architectural level analysis to low-level design analysis is seamless. Below is an example of a fully developed architecture-level use case:

- *Name:* Play Song
- *Actors:* End-user
- *Stakeholders:*
 1. End-user
 2. Music provider
- *Event:* User pushes play button
- *System:* xmms, libxmms, input, output, visualization
- *Purpose:* Describe collaboration among subsystems to accomplish “play” functionality
- *Priority:* 10/10-core business process, crucial for business operation
- *Overview:* Input stream is processed to produce wanted output (song playing or streaming to a file on hard disk)
- *References:* None
- *Related Use-Cases:* Setup
- *Responsibility:* Play media stream or write it to a file
- *Preconditions:* Play-list was configured, Setup use-case successfully performed
- *Postconditions:* System stops playing, after input stream end, if Repeat option is turned off.
- *Invariants:* None
- *Main Scenario:*
 1. xmms: User interface component signals “play” event is raised.
 2. xmms: Signal to input subsystem to start processing data
 3. xmms: Connector between input and output is established
 4. input, output, visualization: Start processing data streams
 5. input: If end of the stream signal xmms and stop
 6. xmms: If “Repeat” option turned on signal input to start processing again, else “stop” signal to output and visualization
- *Alternatives:*
 - step 4: data stream disconnected before end of it (file deleted, network connection went down, etc.) — raise exception and inform user

- step 4: special effects events raised — activate appropriate plugin, which alters output

- *Quality Attributes:*
 1. Responsiveness — events are handled without delays
 2. Reliability — user is informed within 1 second if system stops due to data stream problems
- *Technology:* Network access support for network streams
- *Special Requirements:* For low-end systems, output processes have higher priority over visualization and affect subsystem’s processes
- *Open Issues:* None

A single column format was used to document this particular use case. One could also use multiple column format to emphasize subsystems and modules. The second option has a drawback that it is harder to format text properly thus increasing production and maintenance time.

7. Current Work and Open Questions

Currently, I am involved in an exploration of the following topics:

- Identification of architecture-level requirements: properties and patterns. This topic involves analysis of which properties of the requirements have a significant architectural impact. This knowledge can be used to discover them and isolate from the different software requirement specification documents.
- Recovery of the architecture-level requirements from code, UI, and deployment properties. This recovery is a process of abstracting and combining requirements all the way up to the business level. Its value is in being able to analyze how software impacts the business. This analysis is important in situations in which new software is acquired and business is tailored to it.
- Analysis of the architecture-level requirements change. This analysis is an observational study of several systems to try to discover the evolution patterns and properties of the requirements that actually change over time.
- Techniques for the architecture-level requirements specifications. There are two main techniques:

- Proposal of a new way to organize software requirement specifications. The aim is to structure the requirement specifications in order to preserve and emphasize business and software architecture requirements and concepts.
- Architecture-level use cases for capturing dynamic properties and functional requirements at the subsystem level.

8. Conclusion

This paper has attempted to emphasize the importance of the conceptual shift from the traditional domain and product-level requirement specifications to multiple level specifications and to architecture-level requirements specifications, in particular. The main purpose of this shift is to accommodate the development using new software paradigms. Also presented were some parts of the work that was done as a part of a study in software architecture recovery.

References

- [1] *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, Mar. 2002.
- [2] W. Aalst and K. Hee. Framework for business process redesign. In J. R. Callahan, editor, *Proceedings of the Fourth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 95)*, pages 36–45, Berkeley Springs, 1995. IEEE Computer Society Press.
- [3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston, Massachusetts, first edition, 1999.
- [4] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Boston, Massachusetts, second edition, 1994.
- [5] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Modeling early requirements in tropos: A transformation based approach. In *AOSE*, pages 151–168, 2001.
- [6] F. J. Brooks. No silver bullet. *Computer*, 20(4):10–19, April 1987.
- [7] J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. *Lecture Notes in Computer Science*, 2068:108–??, 2001.
- [8] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: The tropos project. *To Appear in Information Systems, Elsevier, Amsterdam, The Netherlands*, 2002.
- [9] P. Coad and E. Yourdon. *Object Oriented Analysis*. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.
- [10] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.
- [11] T. H. Davenport. *Process Innovation – Reengineering Work through Information Technology*. Harvard Business School Press, Boston, first edition, 1993.
- [12] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, first edition, 1978.
- [13] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, Massachusetts, second edition, 1999.
- [14] P. Giorgini, A. Perini, J. Mylopoulos, F. Giunchiglia, and P. Bresciani. Agent-oriented software development: A case study. In *Proc. of the 13th Int. Conference on Software Engineering & Knowledge Engineering (SEKE01)*, Buenos Aires, Argentina, 2001.
- [15] F. Giunchiglia, J. Mylopoulos, and A. Perini. The tropos software development methodology. *Technical Report No. 0111-20, ITC - IRST. Submitted to AAMAS '02. A Knowledge Level Software Engineering 15*, 2001.
- [16] F. Giunchiglia, A. Perini, and F. Sannicolo. Knowledge level software engineering. In *Springer Verlag, Editor, In Proceedings of ATAL 2001, Seattle, USA. Also IRST TR 011222, Istituto Trentino Di Cultura, Trento, Italy*, 2001.
- [17] M. Hammer and J. Champy. *Reengineering the Corporation: a Manifesto for Business Revolution*. Nicholas Brealey P., London, first edition, 1995.
- [18] R. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley, Boston, Massachusetts, first edition, 2000.
- [19] P. Knauber, D. Muthig, K. Schmid, and T. Wide. Applying product line concepts in small and medium-sized companies. *IEEE Software*, Vol. 17, Iss. 5, pages 88–95, 2000.
- [20] M. Kolp, P. Giorgini, and J. Mylopoulos. A goal-based organizational perspective on multi-agent architectures. In J.-J. Meyer and M. Tambe, editors, *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages (ATAL-2001)*, pages 146–158, 2001.
- [21] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, Englewood Cliffs, N.J., second edition, 2001.
- [22] S. Lauesen. *Software Requirements: Styles and Techniques*. Addison-Wesley, Boston, Massachusetts, first edition, 2002.
- [23] J. Mylopoulos, L. Cheung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of ACM*, Vol. 42, No. 1, 1999.
- [24] J. Mylopoulos, L. Chung, S. Liao, H. Wang, and E. Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, /2001.
- [25] J. Mylopoulos, M. Kolp, and J. Castro. UML for agent-oriented software development: The tropos proposal. *Lecture Notes in Computer Science*, 2185:422–??, 2001.
- [26] J. Mylopoulos, M. Kolp, and P. Giorgini. Agent-oriented software development. In *SETN*, pages 3–17, 2002.
- [27] S. R. Schach. *Classical and Object-Oriented Software Engineering With Uml and Java*. McGraw-Hill, fourth edition, 1998.
- [28] D. Svetinovic. Agile architecture recovery. Master’s thesis, School of Computer Science, University of Waterloo, 2002.
- [29] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, Vol. 26, No. 10, 2000.

- [30] E. Yourdon. *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, N.J., first edition, 1988.
- [31] E. Yourdon and L. Constantine. *Structured Design : Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, Englewood Cliffs, N.J., first edition, 1979.

Architecture-Based Design of Computer Based Systems

Mark Denford, Tim O’Neill, John Leaney
Architecture-based Engineering Research Program
Institute for Information and Communication Technologies
University of Technology, Sydney
[Mark.Denford | Tim.ONeill | John.Leaney]@uts.edu.au

Abstract

This paper presents a practical approach to architecture-based design of computer based systems. The approach is discussed in relation to other existing methods of performing discovery, abstraction, refinement and evolution of systems’ architectures. It has also be shown that this approach can be supported by formal methods of refinement. The approach assists the designer to maintain a strict focus of reasoning about the architecture and its qualities.

1. Introduction

The importance of architecture in the engineering of computer based systems is widely recognised [19, 22, 34, 38]. Given a strong architectural model [30], the architectures of systems can be visualised [9], reasoned about [4, 8, 27, 32], and evolved [36]. These activities are part of any process of architecture-based design, which we call A-Design. This paper particularly investigates the area of (formal) design traditionally called refinement.¹

It is important to consider architecture-based design in a practical engineering context and in particular to address the practical concerns of the engineering effort involved in developing (short term change) and evolving [36] (long term change) a given Computer Based System (CBS). One must always be sure that in designing a system the non-functional requirements of the system are satisfied [5, 11]. Additionally, no system is ever built from nothing; in practice the designers will have suggestions for the system at every level of abstraction [3, 29, 41].

The latest IEEE Requirements standard recommends that requirements are hierarchical [21] and always have some associated implementation restrictions. As a result there will be a need to place these restrictions at whichever level of abstraction is most appropriate [5, 11, 31]. In addition to this need to be able to interact at which ever

level of abstraction is most appropriate, the practicing designer will often as a first step need to discover the architecture of an as-built system. That is, they will need to develop a concrete architecture of a system and abstract away details until the underlying architecture of the system is exposed [14, 24].

This paper presents a practical approach to architecture-based design of CBSs. The approach is discussed in relation to other existing methods of performing discovery, abstraction, refinement and evolution of systems’ architectures. It will also be shown that this approach (whilst generally being more practically based than other more formal methods) can be supported by these formal methods of refinement in particular. The approach also helps to maintain a strict focus of reasoning about the architecture and its qualities.

General architectural definitions are presented in section 2. Section 3 presents a discussion of related work on refinement in general and section 4 builds upon this work with our concept of architecture-based design. Finally a practical approach to architecture-based design is presented in section 5. The paper finishes with a discussion of future work and a conclusion (section 5.1).

2. Architecture

This section presents some general architectural definitions to establish the vocabulary of the paper. From the IEEE [22] and extended by the ECBS Architecture Working Group [37] and UTS [26] the following definitions are provided;

System: A set of interrelated entities which display a specified behaviour while interacting with the system’s particular environment.

Architecture: Any well defined form of a system’s essential, unifying structure defined in terms of components, connections and constraints along with the system’s interaction with its environment.

Architectural description (A-Description): A product which documents an architecture and consists of zero or more architectural models, including rationale for and relationships between the models and views chosen.

Architectural models (A-Models): Any formal description of a system which describes the system’s

¹ If refinement is the process of taking an architecture from the abstract to the concrete, then “abstracting” is the reverse process - taking an architecture from the concrete to the abstract. For the purposes of this paper we refer to both the concepts of “abstracting” and traditional refinement as (part of) design.

architecture. Typically A-Models are formulated using a specific A-Style while embodying (or portraying) one or more A-Views (refer to [26]).

Architectural Model Elements (A-Elements): The constituents of a system’s architectural model which represent the components (Cp), connections (Cn) and constraints (types, implementations, properties, etc) of the proposed system architecture.

Please note: For reasons of brevity and concentration, the concepts of Architectural Styles (A-Style), Architectural Patterns (A-Patterns), Architectural Principles (A-Principles) and Stakeholders concerns are extensively discussed in other places, including our working group [26] but omitted here.

In Figure 1, UML notation [40] is used as an extended entity relationship diagram, to compare and contrast the meanings of the definitions.

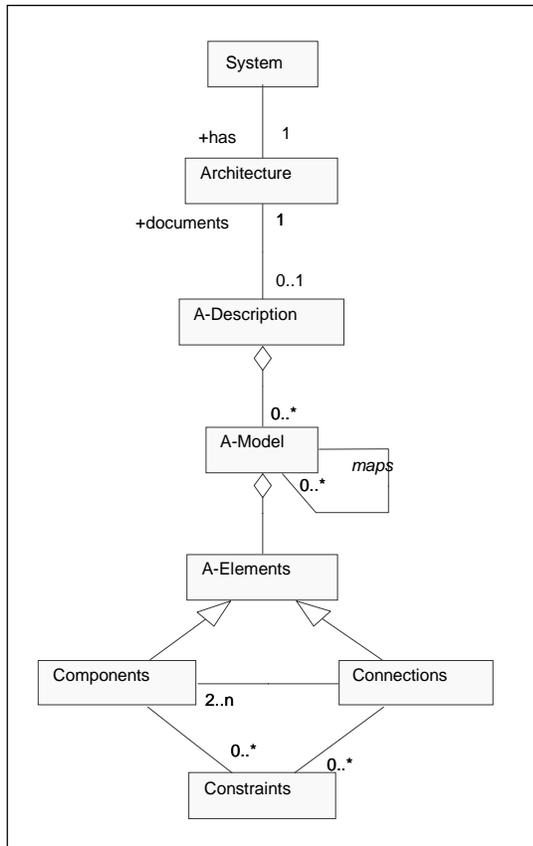


Figure 1 – Interrelationship between key Architectural terms [26]

3. Related Work

This section discusses the history of refinement methods in general from it’s origins in early program proving work to the emergence of architectural refinement methods in the 1990’s. Additionally, existing architectural refinement approaches are discussed, and, finally the strong

relationship between requirements and refinement is argued with a recognition of the importance of hierarchical requirements to a refinement approach.

3.1 History of Refinement Methods

Existing refinement methods have their origins in the program proving work of Dijkstra [10] and Hoare [17]. The initial work on stepwise refinement by Dijkstra [10] appears to be the first to use logic in the construction of a program from a design, as specified by pre/post conditions. Refinement of programs from models gained a boost with the work on the Vienna Development Method (VDM) [23] and Z [15] and B [2]. These methods specify systems, and refine them to programs using predicate calculus and proof obligations. Z has been used on large practical systems to reverse specify and then, by proof techniques, understand and re-specify industrial systems, including the IBM CICS system [42]. They have also been used to model and prove the correctness of systems. Another approach, exemplified by the LARCH project, [25] makes use of axioms and rewriting logic to model a system and prove the correctness of an implementation.

All these methods have one major weakness in that there is no concept of design attributes being satisfied. For example, the design requirement that the system be maintainable could require the application of the principles of coupling and cohesion. One exception is Object-Z [7] which implicitly applies the principles of coupling and cohesion since it forces an information hiding paradigm on the modeller.

3.2 What is Architectural Refinement

Architectural refinement methods have been developed since the early 1990’s. Broadly speaking, these methods can be classified into predicate logic reasoning and refinement methods [4], and methods focussing on rewriting logic or mapping architectural patterns and styles. [5, 31].

In [6], Büchi discusses refinement from a formal specifications perspective. He refers to refinement to be that “the implementation actually complies with its specification, or, more precisely, is a refinement thereof”. This perspective is similar to the classical refinement approach which uses the notion of “behavioural substitutability”. That is, the concrete representation should not show any behaviour not observable in the abstract representation [13]. This is the approach used in CSP [18]. Moriconi et al [31] argue that behavioural substitutability may not be sufficient and introduce “conservation extension”. That is, if a feature is not explicitly included in the abstract then it is implicitly claimed not to exist [13]. According to the above authors, refinement is thus the process of ensuring that these

conditions of “behavioural substitutability” and “conservation extension” are met.

Garlan [13] introduces his own perspective on refinement by claiming “that there is no single definition of refinement. Rather, refinement rules must be specific about what kinds of properties they are preserving in the refined design”. This sort of definition is moving towards the notion that refinement can be thought of in terms of ensuring the non-functional properties of a system such as evolvability and performance.

3.3 Refinement and Requirements

Egyed et al [11] imply a strong relationship between requirements (both functional and non-functional) and refinement. They mention the “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).” This perspective is also supported by Bolusset et al [5] who state that refinement is used to ensure that the system’s concrete implementation still meets its requirements.

We propose to extend Egyed et al and Bolusset et al’s concepts of architectural refinement by including the concept of hierarchical requirements, supported by the latest IEEE standard on System Requirements Specifications [21]. It argues that requirements are assembled “into a hierarchy of capabilities where more general capabilities are decomposed into subordinate requirements”. The implication of this to architecture-based design is that at each level of abstraction a certain subset of the overall system requirements will be addressed. This is further discussed in the following section.

4. Architecture-based Design

This section defines the architecture-based design of Computer Based Systems (CBSs). Additionally two types of architecture-based design (*horizontal* and *vertical*) are identified and discussed with regard to the differing reasons for using each one. The section concludes with a diagrammatic summary of the terms and relationships introduced in this section. This diagram provides the basis for the proposed practical design approach (section 5).

4.1 Definition

The more recent interpretations of refinement [5, 11] and the concept of hierarchical requirements [21] influences the definition of architecture-based design used within this paper. Within the context of the architecture-based design approach discussed herein and the general architectural definitions of the UTS [26] (summarised in section 2) architecture-based design (A-Design) can be defined as;

A-Design: the addition (or removal) of A-Elements to the A-Model to ensure a larger subset of the overall requirements are met.

As is evident in Figure 2, A-Design can be seen in terms of developing another A-Model that satisfies more of the overall systems requirements. The initial A-Model satisfies a certain set of the system’s requirements, R_1 . After the refinement step, the final A-Model satisfies the set of requirements, R_2 . Given that R_1 is a proper subset of R_2 , the final A-Model also satisfies all of the requirements originally satisfied by the initial A-Model.

In practice, R_1 may not be a proper subset of R_2 after the first attempts at refining. The actual refinement method must detect this and ensure that the condition is met before the refinement step is considered complete. This is discussed further in section 5.

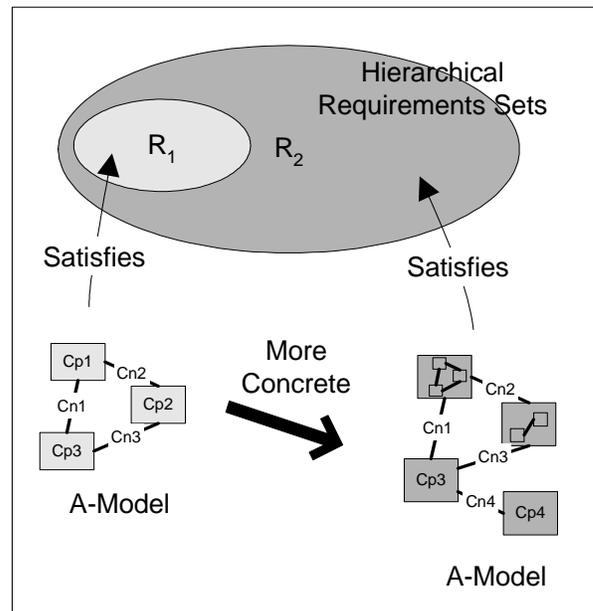


Figure 2 - Relationship between A-Design and Requirements

4.2 Types of A-Design

Following the general definition of A-Design given in section 4.1, more specific definitions can be given for two types of A-Design, *horizontal* and *vertical*, that are often discussed, though not entirely agreed upon, in literature.

Bolusset et al [5] refer to horizontal refinement as inducing a specification modification where there is no change of abstraction level. We build on this, and within our approach and architectural definitions of the UTS [26] (summarised in section 2), define;

Horizontal A-Design: the addition (or removal) of A-Elements at the **same level of abstraction** to satisfy an additional subset of the overall requirements – both functional and non-functional.

In [5] the process of vertical refinement is referred to as moving “closer to the implementation by going from a first architecture description language, to a second one”, that is, moving from one level of abstraction to a second one. “Details are added to remove a part of indeterminism or to facilitate implementation”. This interpretation of vertical refinement as involving a transition of levels of abstraction is generally well understood.

Whilst discussing the vertical case with respect to their refinement method, Moriconi et al [31] comment that “we are guaranteed that the most concrete architecture in the hierarchy meets the requirements of the most abstract architecture in the hierarchy.” Thus vertical refinement involves a transition of levels of abstraction.

We build on these concepts and within our approach and architectural definitions of the UTS [26] (summarised in section 2) define:

Vertical A-Design: the addition (or removal) of A-Elements at a **more concrete or less concrete level of abstraction** to satisfy an additional subset of the overall requirements – both functional and non-functional.

Of particular note is the “more concrete or less concrete” portion of the definition which reflects the concept of abstracting (from concrete to abstract) being the reverse process of refining (from abstract to concrete).

4.3 Levels of Abstraction

A fundamental aspect of our architecture-based design (A-Design) approach is the concept of levels of abstraction for A-Models that originates from the work of Ward and Mellor. In [41] they introduce the concepts of an *Essential Model* and an *Implementation Model*:

“Given that a system must function in a specific environment, and given that it has a purpose to accomplish, it is possible to describe what it must do (the essential activities) and what data it must store (the essential memory) so that the description is true regardless of the technology used to implement the system...an essential model.

It is also possible to describe a system as actually realised by a particular technology...an implementation model.

The implementation model is defined as an elaboration of the essential model that contains enough detail to permit a successful implementation with a particular technology.” [41]

We extend these concepts to the domain of systems architecture by introducing the *essential architecture* and the *implementation architecture*. In addition to these two architectures we also introduce the concept of many

intermediate architectures. In all cases, these architectures are actually represented using the UTS [26] terms summarised in section 2. Each of the essential, intermediate and implementation architectures represent differing levels of abstraction in modelling the system.

4.3.1 Essential Architecture

The *essential architecture* is the most abstract representation that a particular project will use of the architecture. It utilises abstraction to help highlight key system properties, architectural components and component interaction. The essential A-Model(s) are the primary model(s) about which one can reason about to ensure that the system is capable of meeting its requirements. Further essential modelling concepts may be drawn from systems engineering and systems theory literature to help establish the essential A-Model [3, 29, 41]. Our approach is used to formalise the concepts associated with the essential A-Model and then provides a basis on which to reason about the feasibility of the proposed architecture.

The most important aspect of an essential architecture is that, by definition, it contains no implementation details. As Ward and Mellor state, the system as described by an essential architecture could equally be implemented by humans manually executing the required processes as it could by a computer system [41]. The essential architecture should give no indication of what technology should be used in the final implementation.

4.3.2 Intermediate Architecture

Depending on the complexity of the system being modelled, there may be multiple *intermediate architectures*. The intermediate architectures describe the system in successively more detail than the (essential) requirements driven essential architecture. The intermediate A-Models are a primary tool in refining the system down to a subsequent implementation architecture that is capable of meeting the system’s functionality, performance and quality (including evolvability) requirements. Our approach is used to develop the intermediate A-Models and provides a basis on which to continue to reason about the feasibility of the proposed architecture, and its relation to the essential architecture. In relation to the essential architecture, we are especially interested in the intermediate architecture being a correct refinement, which shows promise in meeting the non-functional requirements.

One important point to raise here is that by having intermediate architectures one can approach the A-Design of a system at whatever level of abstraction the designers are comfortable with, or have data with which to populate the A-Model [3, 29, 41]. The merits of this flexibility concept in our A-Design approach will be further discussed in section 5.

4.3.3 Implementation Architecture

A system’s essential and intermediate A-Models are important abstractions, however they do not consider the system solution sufficiently with respect to implementation issues – it is the final *implementation architecture* that delivers the final, specified functionality and capability [39, 43]. Our approach is used to ensure that the architectural solution chosen is appropriate and feasible given the skills and technology available.

The final implementation architecture is not the end of the detailed design process, it is in fact, the beginning. The implementation architecture components, connections and constraints are now ready to be implemented as (e.g.) collections of classes.

4.4 A representation of A-Design

Given the definitions for architecture-based design (A-Design) and the essential, intermediate(s) and implementation A-Models as discussed above, an approach for A-Design can now be presented that facilitates the mapping between each of the A-Models.

Figure 3 illustrates the important concepts in our approach for performing A-Design. It can be seen that A-Design can occur in two “dimensions”: *horizontal* and *vertical*, and that each of these dimensions is bi-directional. Each level of abstraction (essential, intermediate(s) or implementation) satisfies a certain subset of the system’s requirements. The requirements shall either be fulfilled directly at that level of abstraction, or indirectly via the fact that the architecture at the current level of abstraction is a faithful interpretation of those architectures at a higher level of abstraction. Thus, at the implementation architecture, all requirements shall be fulfilled [31].

It is important to note that while each vertical refinement of the A-models in Figure 3 is shown as a graphically similar A-model this is not necessarily the case. As represented graphically and described in [12], for successive A-models “a given element from one space can map to zero, one, or more elements in the lower level space”.

5. A Practical A-Design Approach

This section presents a practical approach to architecture-based design (A-Design). Firstly, the requirements for A-Design (the necessary capabilities of a A-Design method) are presented. The details of a practical A-Design method follow. The section finishes by discussing how this practical A-Design method satisfies the requirements of A-Design.

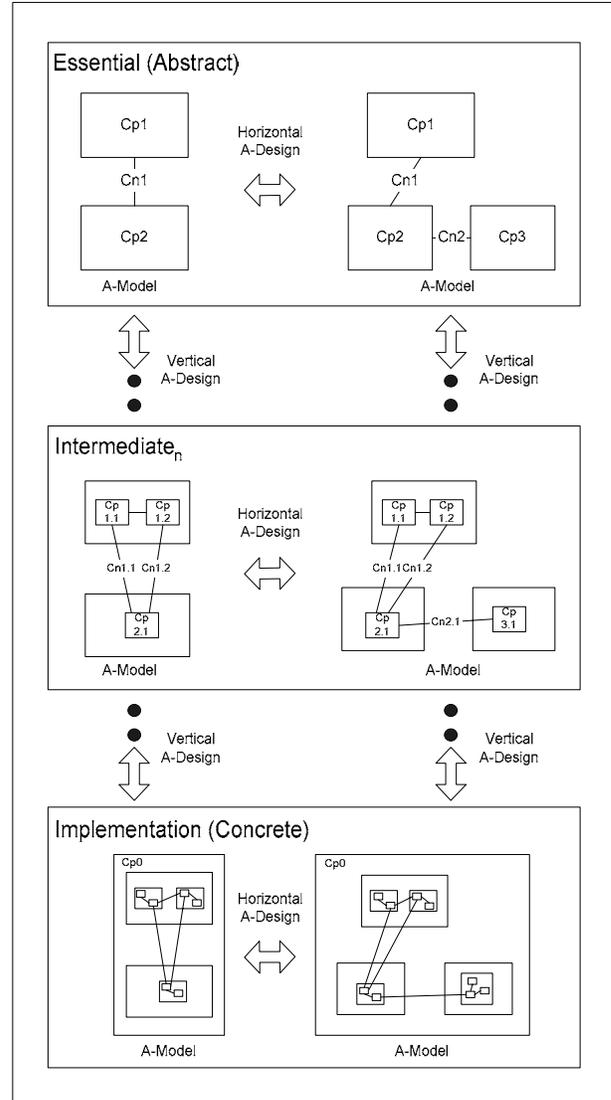


Figure 3 – Architecture-based design (A-Design) using architectural models (A-Models) of different levels of abstraction

5.1 Requirements of A-Design

Following from the previous discussion, and definitions, we propose that the following requirements need to be met in order to produce a practical A-Design process that will aid the designer in both the development and the evolution of Computer Based Systems. The A-Design approach must:

- 1) Be scalable and practical for large, heterogeneous systems.
- 2) Provide the ability to begin design with an A-Model at any level of abstraction.
- 3) Provide the ability to design in any dimension and direction (see section 4).
- 4) Support “long term design”, that is, evolution
- 5) Ensure that both functional and *non-functional* requirements are met.

- 6) Be rigorous, yet flexible, with no specified ordering of horizontal or vertical A-Design steps.

5.2 Approach

This section describes in steps a process for practical A-Design. The input to the A-Design process is an A-Model, and the output is simply another A-Model that has been refined or abstracted depending on the dimension in which refinement is occurring.

There are two “pre-steps”, or essentially tasks that are assumed to have been done prior to use of the practical A-Design approach, as follows:

Pre-Step A (A-Model Population / A-Discovery):

This pre-step involves gathering the appropriate “information” from the most appropriate available sources. Examples include requirements documents, especially constraints and specified equipment, source code, design documents and interviews with system architects / designers. Once this information is parsed (either manually or automatically) a collection of elements (but not necessarily A-Elements) will exist. A-Discovery is concerned with reasoning about the gathered elements and deciding which of those are legitimate A-Elements and which are not. Thus, after gathering the elements they must be filtered so as to keep only the A-Elements. Once this is done we have the initial architecture model, designated A-Model_i. A-Model_i may represent the system at any level of abstraction (from Essential to Implementation).

Pre-Step B (Requirements):

A requirements analysis has to have been completed before the A-Design can commence. The requirements need not be completely defined from the outset, however when the A-Design takes place it will simply work off whichever requirements are defined at that stage. As such the approach could be used in many different system development life cycles, from traditional waterfall to evolutionary, all of which have a different notion as to what stage and level of completion the requirements analysis shall be completed before embarking on the subsequent life cycle stages.

Once these pre-steps have been completed, the main steps of the architecture-based design approach can commence. The input to the approach is a certain A-Model, A-Model_i. The output from an iteration of the approach is the next A-Model, A-Model_{i+1}. The composition of the new A-Model depends on which direction and dimension A-Design is taking place. For example if one is abstracting (vertical A-Design moving upwards) the A-Model might typically contain fewer A-Elements and be generally simpler, whereas if one is refining (vertical A-Design moving downwards) the A-Model might typically contain more A-Elements and be generally more complex.

The main steps of the approach are shown diagrammatically in Figure 4 and are as follows:

Step 1 (Initial Evaluate and Prove):

This step is important as it sets the baseline “status” for A-Model_i. This step gives the approach the ability to know where one is coming *from*, in order to guide where one is going *to*. For example, assume at a high level of abstraction, an architecture exhibited a strong peer-to-peer architectural shape, however an implementation requirement is the specification of a particular client-server (shaped) database. The evaluation of the architecture when the client-server implementation detail is added would be flagged as a “mismatch”.

Essentially, the aim is to prove whether the functional requirements (R_F) are met, and to evaluate whether the non-functional requirements (R_{NF}) are met (for more details see step 4). Regardless of whether the requirements are met or not, this evaluation, or performance index (PI), is fed to the A-Principles “knowledge base”. This knowledge base is used for the generation of new A-Models, that is, it is used to gauge how successful previous modifications to the A-Models have been, and to then guide the proposal of new A-Models.

Step 2 (Propose New A-Model):

This step involves generating a proposed alternative to the current A-Model_i, designated A-Model_{i+1}. The generation of the next A-Model_{i+1} is guided by the A-Principles knowledge base (as discussed above). Given A-Model_i, the PI guides the designer in making the appropriate aggregations, substitutions and decompositions depending on the dimension and direction being refined (see section 4).

Step 3 (Evaluate and Prove):

This step is a repetition of the activities of step 1 and involves checking A-Model_{i+1} against the requirements, both functional and non-functional. It must be proven that the functional requirements are met. It is at this point in the approach that existing methods of architectural refinement could be used. The specific method used would vary depending on how the functional requirements had been expressed [5, 31]. Typical non-functional requirements evaluated in this step are performance, evolvability and openness.

Once these proofs and evaluations are complete, we can see whether A-Model_{i+1} satisfies the requirements. The results of the evaluation are fed back into the A-Principles PI so as to enhance the “knowledge” contained. Should the evaluation prove that the requirements have been met, the fact that the changes from A-Model_i to A-Model_{i+1} resulted in a successful refinement are incorporated into the knowledge base. The same is true for the reverse case - the changes did not result in the requirements being met, and this is also fed back into the knowledge base. If the answer is “No” then the process is repeated from step 2 again until we are successful in meeting the requirements and we have the refined A-Model, designated A-Model_{i+1}.

It is evident, via step 1 of the approach, that we have the ability to enter the A-Design process at whichever level of abstraction is most appropriate. Obtaining the initial A-Model_i from source code will result in a refinement process that begins at a much more concrete level of abstraction than one that begins with a high level architectural description.

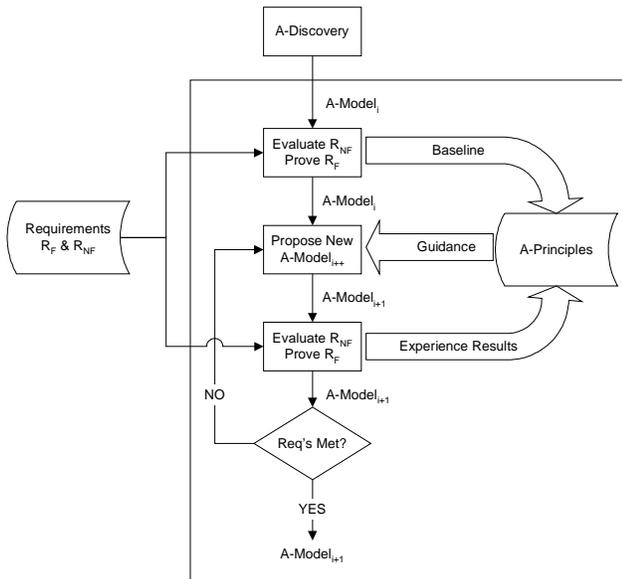


Figure 4 – An Iteration of the Architecture-based design (A-Design) Approach

Within this approach, A-Design can proceed in any dimension and any direction (horizontal moving left and right, and vertical moving up and down respectively). This is partly due to the evaluation method described in step 1 and 3 which allows any new model, A-Model_{i+1} to be evaluated against the functional and in particular the non-functional requirements of the system.

The benefit of this to the designer is that it gives them the ability whilst typically developing a new system, to design vertically and down in order to approach a more concrete model. In addition, for an as-built system which is required to evolve in a certain way, it gives the designer the ability to enter the A-Design process with a typically more concrete model, and to abstract vertically and upwards until the model is sufficiently abstract to reason about and make the required changes. Once this reasoning is complete and satisfactory, the A-Design process can proceed in the reverse direction (that is, vertically and down) until a suitably concrete model is again obtained. This “round trip” of abstracting up, reasoning, making alterations and then refining down is an approach that is very well suited to the long term evolution of a system.

Consequently, the approach is very flexible in allowing the designer the freedom to enter at any level of abstraction, and to move in any dimension and direction as

required by the designing task at hand. However, at the same time, the approach is rigorous in that at each A-Design step the new model is evaluated against the overall requirements of the system to ensure that no unsuitable or unwanted changes have been made.

Non-functional requirements and their evaluation are fundamental to this approach and it is in this area that the practical A-Design approach described here differs the most from existing refinement methods [5, 18, 31].

The approach is tied heavily to the satisfaction of requirements, functional and importantly non-functional. Consequently, it gives the designer confidence that they are not only producing an architecture that is a valid refinement or abstraction of their starting point, but additionally that it is a “good” architectural solution.

6. Future Work and Conclusion

There are three aspects to the future work. Firstly, to incorporate related work such as co-design [35], and other methods such as MASCOT. Secondly, to include into the approach the idea of evolution as a third dimension of A-Design. We also need to develop a formal method for evolution. Finally, we need to incorporate the whole approach into the ABACUS tool suite [1].

In conclusion, we have developed a practical approach to the architecture-based design (referred to as A-design) which aims to simultaneously satisfy the functional and non-functional properties of a system. This approach is based upon the various architectural refinement calculi.

7. Acknowledgements

This work was undertaken as part of an Avolution Pty Ltd sponsored project and funding was also furnished in part by the Australian Research Council (ARC) in the form of an Australian Postgraduate Award (APA) scholarship for Mark Denford.

8. References

- [1] ABACUS, <http://www.avolution.com.au>.
- [2] Abrial, J-R., *the B-book – assigning programs to meanings*, Cambridge University Press, 1996.
- [3] Alexander C., *Notes on the Synthesis of Form*, Harvard University Press, Cambridge MA, 1964.
- [4] Allen, R., J., *A Formal Approach to Software Architecture*, PhD Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997.
- [5] Bolusset, T., Oquendo, F., *Formal Refinement of Software Architectures Based on Rewriting Logic*, International Workshop on Refinement of Critical Systems: Methods, Tools and Experience, Grenoble, France, 2002.
- [6] Büchi, M., Sekerinski, E., *Formal Methods for Component Software: The Refinement Calculus Perspective*, Second Workshop on Component-Oriented Programming, Jyväskylä, June 1997.

- [7] Carrington, D., Duke, D., Duke, R., King, P., Rose, G., Smith, G., *Object-Z: An Object-Oriented Extension to Z*. FORTE 1989: 281-296.
- [8] Chaudet, C., Oquendo, F., π -Space: A Formal Architecture Description Language Based on Process Algebra for Evolving Software Systems, Proceedings the Fifteenth International Conference on Automated Software Engineering (ASE'00), Grenoble, France, September 2000.
- [9] Denford M., O'Neill T., Leaney J., *Architecture-based Visualisation of Computer Based Systems*, Proceedings of ECBS'02, Lund, Sweden.
- [10] Dijkstra, E. W., *Why Correctness Must Be a Mathematical Concern*, in Boyer, R. S. and J. S. Moore (eds.), *The Correctness Problem in Computer Science*, Academic Press, 1981.
- [11] Egyed, A., Grünbacher, P., Medvidovic, N., *Refinement and Evolution Issues in Bridging Requirements and Architecture – The CBSP Approach*, First International Workshop From Software Requirements to Architectures (STRAW'01), Toronto, Canada, May, 2001.
- [12] Egyed, A., Medvidovic, N., *Consistent Architectural Refinement and Evolution using the Unified Modeling Language*, First Workshop on Describing Software Architecture with UML, Toronto, Canada, May, 2001.
- [13] Garlan, D., *Style-based Refinement for Software Architectures*, Proceedings of the Second International Software Architecture Workshop (ISAW2), San Francisco, October 1996.
- [14] Guo, Y.G., Atlee, J.M., Kazman, R., *A Software Architecture Reconstruction Method*, Proceedings of First Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA, February, 1999.
- [15] Hayes, I.(ed.), *Specification Case Studies*, Prentice-Hall, 1987.
- [16] Heales J., *Evolutionary and Revolutionary Maintenance of information Systems: A Theoretical and Empirical analysis*, PhD Dissertation, Dept. of Commerce, The University of Queensland, Australia, 1998.
- [17] Hoare, C. A. R., *The Axiomatic Basis of Computer Programming*, Communications of the ACM, 12 (10), October, 1969
- [18] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1985.
- [19] Horowitz, B.M., *The Importance of Architecture in DOD Software*, The MITRE Corporation, Report M91-35, July 1991.
- [20] IBM Pty Ltd, *IBM Insurance Application Architecture*, http://www-1.ibm.com/industries/financialservices/solution/SOLUTIONS_30269.html, 2002.
- [21] *IEEE Standard 1233: IEEE Guide for Developing System Requirements Specifications*, IEEE Computer Society, 1998.
- [22] *IEEE Standard 1471: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE Computer Society, 2000.
- [23] Jones, C. B., *Software Development - A Rigorous Approach*, Prentice-Hall, 1979.
- [24] Kazman, R., O'Brien, L., Verhoef, C., *Architecture Reconstruction Guidelines*, CMU/SEI-2001-TR-026, Technical Report, Software Engineering Institute, Carnegie Mellon University, August 2001.
- [25] Larch, <http://nms.lcs.mit.edu/Larch/>
- [26] Leaney, J. et al., *Current Developments in the Definition and Description of System Architectures - A University of Technology, Sydney (UTS) Architecture-based Engineering Research Program Architecture Working Group (AWG) Position Paper*, Internal UTS report 2002-R-01.
- [27] Leaney, J. et al, *Measuring the Effectiveness of Computer Based Systems: an Open System Measurement example*, Proceedings of ECBS'01, pp 179-189.
- [28] Lehman M.M. & Belady L.A., "A model of large program development", *IBM Systems Journal*, vol. 15, no. 3, 1976, pp 225-251.
- [29] McMenamin, S.M. & Palmer J.F., *Essential systems analysis*, Yourdon Press, Englewood Cliffs (N.J.), 1984.
- [30] Medvidovic N., Taylor R., *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, Volume 26, Number 1, 2000, pp. 70-93.
- [31] Moriconi, M., Qian, X., Riemenschneider, R. A., *Correct Architecture Refinement*, IEEE Transactions on Software Engineering, Volume 21, Number 4, 1995, pp. 356-372.
- [32] O'Neill, T., Leaney, J., Martin, P., *Architecture-Based Performance Analysis of the COLLINS Class Submarine Open System Extension (COSE) Concept Demonstrator (CD)*, Proceedings 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, Edinburgh Scotland, 2000, pp. 26-35.
- [33] Perry D.E., "Dimensions of Software Evolution", in *Proceedings of 1994 Conference on Software Maintenance*, IEEE, 1994, pp. 296-302.
- [34] Rehtin, E., "Systems Architecting : creating and building complex systems", Prentice Hall, 1991
- [35] Schulz, S., Rozenblit, J.W., Mrva, M. and K. Buchenrieder, *Model-Based Codesign*, IEEE Computer, 13(8), 1998.
- [36] Rowe D., Leaney J., Lowe D., *Defining systems evolvability – a taxonomy of change*, Proceedings of ECBS'98, Jerusalem, Israel.
- [37] Rowe et al., "IEEE ECBS'99 TC Architecture Working Group (AWG) Discussion Paper" Proceedings of ECBS'99, pp. 359-364.
- [38] Simpson, H.R., "Architecture for Computer Based Systems", IEEE Publication 0-8186-5715-4/94, 1994.
- [39] Thome B. (ed), "Systems engineering: principles and practice of computer-based systems engineering", John-Wiley and Sons, Chichester (N.Y.), 1993.
- [40] Unified Modeling Language, Rational Software Corporation, most recent updates of UML are available via WWW <http://www.rational.com>
- [41] Ward, P.T., Mellor, S.J., *Structured Development for Real-Time Systems: Volume I*, Yourdon Press, Englewood Cliffs (N.J.), 1985.
- [42] Wordsworth, J. B., *The CICS application programming interface definition*, Z User Workshop, Oxford 1990
- [43] Wymore A.W., *Systems Engineering Methodology for Interdisciplinary Teams*, John Wiley and Sons, New York, 1976.

PATTERNS APPROACH TO BUILDING SOFTWARE SYSTEMS

Satish Chandra
Consultant
Satyam Learning Center
Satyam Computer Services Ltd.
Hyderabad – INDIA

Satyendra Bhattaram
Technical Leader
Knowledge Initiative Group
Satyam Computer Services Ltd.
Hyderabad – INDIA

{satish_chandra, satyendra_bhattaram}@satyam.com

ABSTRACT

This position paper suggests an approach for building software systems using patterns, right from business architecture to software architecture. Further, the approach incorporates a concurrent and iterative development process to ensure that the business architecture and software architecture are aligned, end to end. Usage of patterns leads to reuse of various artifacts, involved in the software development life cycle.

1. INTRODUCTION

It is highly desirable to start the development of a software system with most of the system requirements being captured in the form of use cases. However, in practice systems evolve. New requirements crop up when the business on which the system is based keeps growing/changing. Also, development of a system gives rise to new ideas, which could be incorporated in the system. Thus, in practice, activities take place concurrently, rather than sequentially.

Nuseibeh [15] mentions that 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 alternative design solutions and states that, in practice, software development starts from either requirements or software architecture [15].

It is obvious that the requirements, both functional and nonfunctional, are derived from business architecture. The underlying business architecture may be either implicit or explicit. Hence, we propose that the approach suggested by Nuseibeh could be extended to

business architecture, as well. It is apparent that the effect on requirements would have a cascading effect on the business architecture, and vice-versa. Sometimes, the effect of the software architecture, which led to modification of requirements, could even lead to business process re-engineering.

This article illustrates the linkages that would be there between various artifacts used in the development of a software system. We also suggest a concurrent and iterative process to software development.

2. BUSINESS ARCHITECTURE

The role of architecture in building any type of structure is well defined. A well-designed architecture makes it possible to thoroughly understand the structure being built, to plan the actual construction, and to estimate costs; it serves as the basis for the blueprints of the structure. Once construction has been completed, a good architecture remains as the documentation of the process and the result, making it possible to understand, maintain and, if so desired, to extend the structure [18].

An architecture captures the vital parts of a structure in an organized manner and is a practical tool for managing a complex system, such as a software system or a business. Business Architecture defines the business structure, so modeling this architecture is key to understanding the business and how it functions [18].

Eriksson and Penker [18] propose a business architecture description using four views: business vision, business process, business structure, and business behavior.

They state "The knowledge and information in a business architecture is used to define the software architecture. This isn't a one-to-one mapping, and there is no simple algorithm to convert the business model into a software model. They are two different models that serve different purposes. The business model describes a business or a specific part of a business; not all of the business goes into the software systems. To

define a software architecture, the business architecture is used to:

- Identify suitable support systems.
- Identify functional requirements.
- Identify nonfunctional requirements.
- Act as basis for analysis and design.
- Identify suitable components.”

Creating a business model before the software models, then using the information in that business model for the creation of software models, will increase the quality of the software systems. Systems that better support the business of which they are a part will be the result [18]. They have also catalogued some business patterns.

At times, it may so happen that while establishing the business goals it may be realized that some of the business processes may have to be changed, event to the extent of having a business process reengineering.

3. SOFTWARE ARCHITECTURE

A Software Architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. The software architecture of a system is an artifact. It is the result of the software design activity [8].

Buschmann [25] states: “According to its definition, a pattern system for software architecture should support building concrete software systems with help of patterns. Fortunately, many well-described patterns for software architecture already provide steps and guidelines that specify their implementation [GHJV95] [POSA1] [POSA2]. Many such patterns also provide information about their refinement and combination with other patterns. Whenever another pattern is referenced, its implementation steps can be applied: they thus complement and complete the implementation steps of the original pattern.” He also sees some drawbacks in application of patterns to certain areas like partial design and suggests improvements in the application of patterns in building software architecture.

4. FROM BUSINESS ARCHITECTURE TO SOFTWARE ARCHITECTURE

UML is a de-facto as well as de-jure standard for modeling object-oriented systems. The business architecture could be developed by using UML, as suggested by Eriksson and Penker. The software architecture could also be modeled using the UML or any Architecture Description Language. Between the two ends we could consider the use cases for capturing and documenting requirements, the class diagrams to capture the analysis models as well as design models. The other diagrams of the UML could be used to support the development process.

This paper suggests that the conceptual class diagrams derived from the use cases be refined. It is suggested that the conceptual class diagrams be refined using the classes that can be derived from the analysis patterns of the specific domain or the generalized analysis patterns which apply to the domain under consideration. It may be mentioned that a good number of analysis patterns have been documented covering domains like insurance, virtual libraries, oil & refinery and so on [21,22,23].

It would involve some experience or training to understand that a transaction would give birth to an association class, which has to be identified and documented as an analysis class. Similarly, analysis patterns help in identification of additional classes. This would help in refining the conceptual class diagrams.

The other suggestion is with regard to the design class diagrams. The design class diagrams which are identified in the system could be refined using the design patterns. Apart from the design patterns documented by Gamma and others there are design patterns specific to technologies like EJB, J2EE and so on. These would help in refining the class diagrams.

Taking into consideration the existing architectural patterns/styles could refine the identified software architecture.

Thus, the software architecture, which is a composition of patterns, is derived from the business architecture.

Each of the artifacts would affect the other, as shown in figure 1 (Appendix IV).

5. SOFTWARE DEVELOPMENT PROCESS

The Unified Process describes process workflows as: business modeling, requirements, analysis & design, implementation, test and deployment. It describes configuration & change management, project management and environment as the supporting workflows.

The waterfall methodology describes the various phases of software development cycles as; requirements gathering, analysis, design, coding, testing and maintenance.

The phases of the waterfall model or the workflows suggest sequencing of activities. However, in practice we experience that activities in the phases or workflows happen concurrently.

We strongly feel that the concurrent and iterative development approach presented in [14] and [15], using the Twin Peaks, is close to reality. We suggest that the approach be extended to Three Peaks, with the addition of the business architecture as the third peak. The modification is as given in figure 2 (Appendix V).

6. A Case Study

We had studied a knowledge management system, developed in-house, to identify and discover analysis and design patterns in the system [1]. The identified and discovered analysis and design patterns are presented in Appendix II and Appendix III respectively.

Further, study of the system in identifying the business patterns led to the identification of the business patterns presented in Appendix I.

As regards the solution architecture of the system, our study shows that the System follows the MVC architecture. We have used the Microsoft's ASP technology.

7. CONCLUSIONS

We advocate the continuing of use cases for capturing requirements. However, use case driven approach leads to identification of classes, with boundary, control and entity stereotypes. The conceptual class diagram, which has thus been arrived at, would be similar to a design class diagram without the application of design patterns. Hence, the analysis patterns of the domain or generalized analysis patterns could be used to refine the conceptual class diagram. This would enable creation of rich conceptual class diagrams. The conceptual diagrams could lead to the questioning of business processes and requirements.

The design class diagrams could be refined using design patterns. This would lead to development of systems, which would address the nonfunctional attributes like flexibility, scalability, maintainability, etc..

Some of the issues that need to be addressed are:

Training the various stakeholders.

Analysis patterns for more domains have to be developed.

This methodology has to be validated.

8. REFERENCES

1. Satish Chandra, Suri Ponnada and Satyendra Bhattaram: Pattern Oriented Architecture for a Knowledge Management System – In proceedings of Informatica 2003, Hyderabad, India, January, 2003.
2. Geyer-Schulz and Hahsler: Software Reuse with Analysis Patterns – In proceedings of AMCIS 2002, Dallas, TX, August 2002
3. Fowler, M. *Analysis Patterns: Reusable Object Models*, Object Technology Series. Addison-Wesley Publishing Company, Reading, 1997.
4. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company, New York, 1995.
5. Martin Fowler: Patterns of Enterprise Application Architecture. www.martinfowler.com
6. Moore, M.M., "Software Reuse: Silver Bullet?" *IEEE Software* (18:5), September/October 2001, p. 86.
7. Ulrich Frank, Knowledge Management Systems: Essential Requirements and Generic Design Patterns. Proceedings of the International Symposium on Information Systems and Engineering, ISE'2001, Las Vegas: CSREA Press 2001.
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture, A System of Patterns*, John Wiley & Sons Ltd, Chichester, 1996.
9. Al Williams, Design Patterns for Web Programming Do you need MVC?, New Architect June, 2002.
10. Dick Stenmark, Turning Tacit Knowledge Tangible. Proceedings of the 33rd Hawaii International Conference on System Sciences – 2000.
11. Neighbors, J.M., "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions of Software Engineering* (10:5), September 1984, pp. 564-574.
12. Francisco Montero, María Lozano, Pascual González, Isidro Ramos: Designing Web Sites by Using Design Patterns.
13. Paola Inverardi & Henry Muccini & Patrizio Pelliccione: Checking consistency between architectural models using using SPIN. STRAW 2001.
14. Jon G. Hall, Michael Jackson, Robin C. Laney, Bashar Nuseibeh and Lucia Rapanotti: Relating Software Requirements and Architectures using Problem Frames. STRAW 2001.
15. Bashar Nuseibeh : Weaving the Software Development Process Between Requirements and Architectures. STRAW 2001.
16. Paul Evitts : UML Pattern Language. Techmedia publications.
17. Martin Fowler & Kendall Scott :UML Distilled. Addison Wesley.
18. Hans-Erik Eriksson: Magnus Penker Business Modeling with UML: Business Patterns at Work. John Wiley & Sons. 2000.

19. Grady Booch; James Rumbaugh; Ivar Jacobson : The UML User Guide. Addison Wesley. 1999. of understanding and optimizing this communication.
20. Philippe Kruchten: The Rational Unified Process, An Introduction. Second Edition. Addison Wesley. 2000.
21. Prashant Jain and Michael Kircher: Leasing. PLoP 2000.
22. Wolfgang Keller: Some Patterns for Insurance Systems. PLoP 98.
23. Lei Zhen and Guangzhen Shao: Analysis patterns for oil refineries. PLoP 2002.
24. Ivar Jacobson, Martin Griss, Patrik Jonsson: Software Reuse – Architecture, Process and Organization for Business Success. ACM Press. Pearson Education Asia. Addison Wesley Longman. 2000.
25. Frank Buschmann. Building Software with Patterns. EuroPloP 99.

APPENDIX I - Business Patterns

Resource and Rules Patterns:

1. Actor-Role: Provides guidelines for using actor and role concepts, including how they should be separated and how they can be combined.
2. Organization and Party: Used to create flexible and qualitative organizational (processes)charts in object-oriented models.
3. Product Data Management: All businesses have many products and/or documents that must be organized and structured. Capturing the structure of the relationship between documents and products is a difficult but common problem in all businesses.
4. Thing – Information: Eliminates the focus-shifting that occurs during the modeling process by referring to two frequently used foci(thing focus and information focus) in business modeling and how they are related to each other.
5. Title-Item: Helps modelers to simplify the design process for systems that involve objects that exist in multiple copies or instances. It separates the information about the title from the information about individual instances of that title.
6. Type-Object-Value: Models the relationships between a type, its Object, and value.

Goal patterns:

7. Business Goal – Problem: Used to identify the connection between business goals and their related problems in order to correct the problems and achieve the goals

Process Patterns:

8. Action Workflow: A tool for analyzing communication between parties, with the purpose

APPENDIX II - Analysis Patterns

The following analysis patterns have been identified in the System:

1. Recurring event pattern: (Fowler [3]).
2. Individual instance method (Fowler [3]).
3. Effectivity analysis pattern (Fowler [3]).
4. Range analysis pattern (Fowler [3]).
5. Structured Pin Board (Hahsler [2]).

The following analysis patterns have been discovered in the System:

1. Push Pull Analysis Pattern and
2. Collaborative Problem Solving Analysis Pattern.

APPENDIX III - Design Patterns

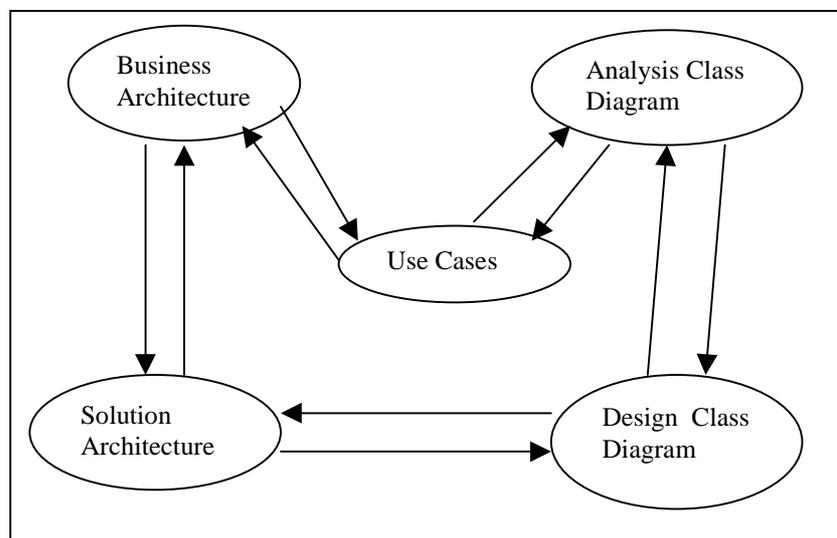
During our literature survey, we had come across two types of design patterns. The first dealing with the user interface and the second dealing with the functionality of the System. We have used all the UI Design Patterns mentioned in [12]. We have used 3 design patterns of GoF [4] and 12 design patterns from Fowler [5].

The identified design patterns from the Gang of Four are: Decorator, Iterator and Facade.

The identified design patterns from Fowler are: Front Controller, Two Step View, Server Session State, Gateway, Mapper, Service Layer, Recordset, Data Access Object, Transaction Script, Domain Model, Table Module and Active Record.

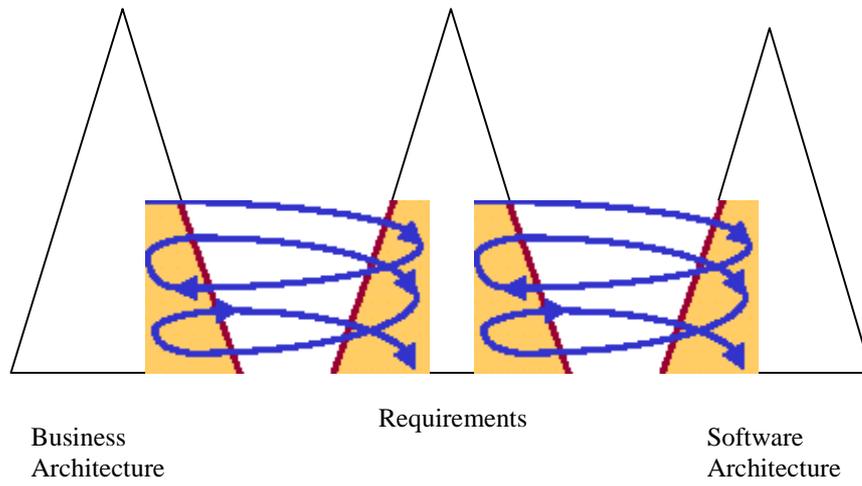
APPENDIX IV

Figure 1 – Relationship between the various artifacts:



APPENDIX IV

Figure -2: Three Peaks – a model of the concurrent development of business architecture, requirements and software architecture



Experience with Global Analysis: A Practical Method for Analyzing Factors that Influence Software Architectures

Robert L. Nord¹, Dilip Soni
Siemens Corporate Research
755 College Road East
Princeton, New Jersey 08540 USA
rn@sei.cmu.edu, dilip@scr.siemens.com

Abstract

A practical method for analyzing the factors that influence software architectures is presented. Factors include organizational context and constraints, available technologies, and product requirements. Analyzing the factors uncovers a small number of issues that drive the design of the architecture. These issues arise from the factors that have little flexibility, a high degree of changeability, and a global impact on the system. The result of the analysis is a set of global strategies that guide the architecture design.

A two-phase approach for analyzing factors and developing architecture design strategies is given. Experience has been gained with this approach in three ways: (1) developing the approach during the design of an imaging system; (2) using the approach to analyze four systems in retrospect; (3) using the approach in new software development projects.

Introducing global analysis into the software development process resulted in a new global analysis specification document that helped bridge the gap between requirements and architecture design and provided a place to explicitly record design rationale.

1. Introduction

Global analysis analyzes factors that globally influence the architecture design of a system. Factors include organizational context and constraints, available technologies, and product requirements. This analysis focuses on key issues that transcend boundaries between development activities, subsystems, and architecture views. The result of the analysis is a set of global strategies that guide the architecture design and improve its adaptability with respect to changes in the factors.

Successful projects prepare for change by noting the flexibility of influencing factors and their likelihood of change, characterizing interactions among the factors and their impact, and selecting cost-effective design strategies to reduce the expected impact of the changes [8].

Three categories of influencing factors are considered during global analysis: organizational, technological, and product.

Organizational factors arise from the business organization. Organizational factors constrain the design choices while the product is being designed and built. They are external to the product, but influence it. Their influence is important because if they are ignored, the architecture may not be buildable.

External technology solutions are embedded or embodied in the product. These factors are primarily hardware and software technologies and standards. These technological factors are external to the product being designed. Unlike the organizational factors, however, they can affect the product throughout its lifetime. Further, they can change over time, so the architecture should be designed with this changeability in mind.

Product factors are used to describe the product's requirements for functionality, the features seen by the user, and nonfunctional properties. The product factors are also subject to change over time, so the architecture should be designed to support such changes.

In this paper, we present the concept of global analysis, a practical method for analyzing factors that influence software architectures. We demonstrate its role in software architecture design and discuss its relationship to other software development activities. We present our experience with developing the method and its use by others in new software development projects. We conclude with lessons learned about the method's value and where further improvement is needed.

¹ Current Address: Software Engineering Institute, 4500 Fifth Avenue, Pittsburgh, Pennsylvania 15213USA.

2. Related Software Development Activities

Figure 1 shows the relationship of global analysis to software architecture design and project planning activities.

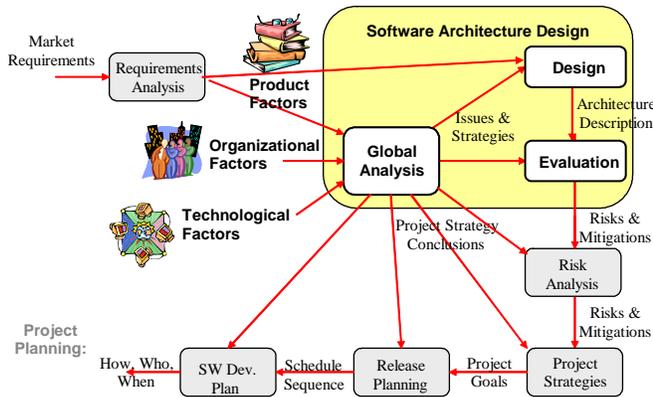


Figure 1: Software Architecture Design and Project Planning Activities

Global analysis complements requirements analysis tasks. Global analysis helps focus on the important architecture requirements; these are the quality attribute requirements. But global analysis goes further than just examining requirements; it includes organizational and technological factors that are not typically included in the requirements document.

The method helps bridge the gap between requirements and architecture design by analyzing the impact of requirements on important technical and business issues that affect design. Global analysis records rationale and provides traceability as requirements are linked to strategies that guide design.

The description of requirements is often textual, but more rigorous requirements analysis methods may employ some combination of feature modeling [6], use case modeling, or object modeling [5]. If such an approach is used, then the artifacts will provide useful input to the global analysis method. Features will be put in global analysis factor tables for further analysis. Use cases show a specific interaction between a stakeholder and the system and provide a means to evaluate the impact of the design decisions in providing a solution to the design issue. Objects encapsulate system responsibilities and will inform the choice of conceptual components in the global analysis strategies that guide the design.

Global analysis generates issues and strategies that guide architecture design and provide input to architecture evaluation. Global analysis begins as the architecture is defined and continues as the design decisions are made. Figure 2 shows the iterative nature between global analysis and the design tasks for any given architectural

view. Global analysis guides design decisions. As design decisions are made, additional constraints may arise that are in turn analyzed and in turn guide additional design decisions.

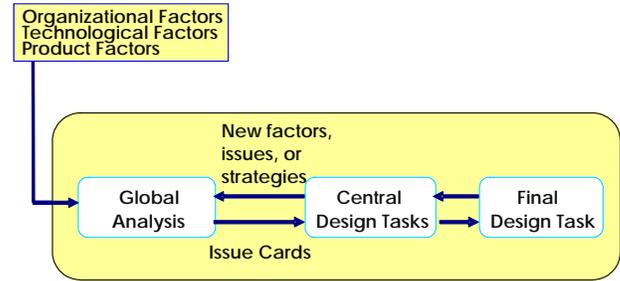


Figure 2: Architecture Design

Global analysis complements architecture evaluation tasks, such as the Architecture Tradeoff Analysis Method (ATAM) [3]. Often, much time is spent at the beginning of the evaluation capturing information about relevant business drivers, quality attribute requirements, and architectural approaches. Rather than record these after the fact, the best time to capture them is as they are made during the design activity. Global analysis captures this information and provides design strategies and their rationale that can be reviewed during the ATAM. ATAM will uncover risks for which additional strategies may need to be developed.

Global analysis provides input to project planning and management activities. It is used to generate project strategy conclusions that help define project goals [10].

3. Global Analysis Activities

The global analysis method consists of two phases: Analyze the factors and Develop issues and strategies.

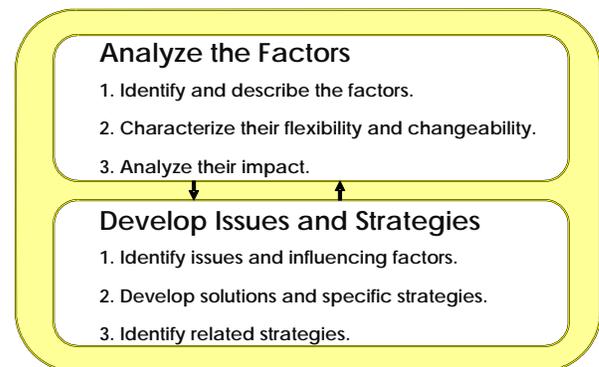


Figure 3: Global Analysis Activities

The process is iterative and may start with either phase.

Phase 1: Analyze the Factors: The first phase analyzes the factors using three steps: (1) Identify and describe the factors; (2) Characterize the flexibility or the changeability of the factors; and (3) Analyze the impact of the factors.

Identify and describe the factors: Consider factors that have a significant global influence, those that could change during development, those that are difficult to satisfy, and those with which you have little experience. Can the factor’s influence be localized to one component in the design, or must it be distributed across several components? During which stages of development is the factor important? Does the factor require new expertise?

Characterize the flexibility of the factors: Describe what is negotiable about the factor. Is it possible to influence or change the factors so that it makes your task of architecture development easier? Use this information when factors conflict or for some other reason become impossible to fulfill.

Characterize the changeability of the factors: Describe what could change about the factor, both in the near and more distant future. In what way could the factor change? How likely is it to change during or after development? How often will it change? Will the factor be affected by changes in other factors?

Analyze the impact of the factors: If the factor will change, which of the following would be affected and how: other factors, components, modes of operation of the system, other design decisions.

Phase 2: Develop Issues and Strategies: The second phase develops strategies for the architecture design using three steps: (1) Identify issues; (2) Develop solutions and specific strategies; and (3) Identify related strategies.

Identify issues: An issue may arise from factors in many ways:

- limitations or constraints (e.g., Aggressive Schedule)
- reducing the impact of changeability (e.g., Changes in Software Technology)
- difficult-to-satisfy product factors (e.g., Easy Addition and Removal of Features)
- common solution to global requirements (e.g., Implementation of Diagnostics)

Develop solutions and specific strategies: Discuss a general solution to the issue, followed by a list of associated strategies. The solution description records analysis-based rationale that illustrates that the strategies satisfy the issue. Strategies should address the issue and one or more of the following goals:

- reduce or localize the factors’ influence (e.g., Buy rather than build)
- reduce the impact of the factors’ changeability (e.g., Use a pipeline for image processing)
- localize required areas of expertise (e.g., Map independent threads of control to processes)

- reduce overall time and effort (e.g., Use incremental development)

Identify related strategies: When a strategy belongs to more than one issue, describe it in one place and reference it as a related strategy in the other issues where it applies.

4. Experience with Developing the Method

We developed the approach informally while designing the architecture of an image acquisition and processing system. After the conclusion of the project, we developed a more rigorous description of the method and provided an example of its use in terms of a fictional system we call IS2000, inspired by this and other systems we studied [4]. The IS2000 system consists of a probe that takes sensor readings that are processed according to the type of acquisition procedure selected by the user. The results of the first phase are documented in a factor table. We illustrate the factor table with an excerpt from IS2000.

<i>Factor</i>	<i>Flexibility/ Changeability</i>	<i>Impact</i>
O4.2 Schedule Feature Delivery		
Features are prioritized	Negotiable	Moderate impact on the schedule
T2.1 Domain-specific Hardware Probe Hardware		
Hardware to detect and process signals	Upgraded every three years as technology improves	Large impact on image acquisition and processing components
P1.1 Features Acquisition Types		
Acquire raw signal data and convert into images	New types of acquisitions may be added every three years	Affects UI, acquisition performance, and image processing

The organizational factor (O4.2) shows there is flexibility in delivering features according to their priority. For other systems these kinds of factors may not affect the architecture, but in this system they will have a significant impact. The technological feature (T2.1) shows that change to the probe hardware is likely and will have a large impact on the imaging components. The product factor (P1.1) shows new types of acquisition algorithms may be added during the lifetime of the system.

The results of the second phase are documented in an issue card. We illustrate an issue card from IS2000.

<p>Issue: Easy Addition and Removal of Acquisition Procedures</p> <p>There are many acquisition procedures. Implementation of each feature is quite complex and time consuming. There is a need to reduce complexity and effort in implementing such features.</p> <p>Influencing Factors</p> <p>O4.1: Time to market is short</p> <p>O4.2: Delivery of features is negotiable</p>

P1.1: New acquisition procedures can be added every three years.
P1.2: New image-processing algorithms can be added on a regular basis.
...
<p>Solution Define domain-specific abstractions to facilitate the task of implementing acquisition and processing applications. Strategy: Use a flexible pipeline model for image processing. Develop a flexible pipeline model for implementing image processing. Use processing components as stages in the pipeline. This allows the ability to introduce new acquisition procedures quickly by constructing pipelines using both old and new components. Strategy: Introduce components for acquisition and image processing.</p> <p>...</p> <p>Strategy: Encapsulate domain-specific data.</p> <p>...</p>
<p>Related Strategies See also Encapsulate domain-specific hardware.</p>

We performed a retrospective analysis on four systems with the aid of the architects who designed the systems [4][9]. We interviewed the architects to understand the process they used to go from requirements to design. We solicited feedback on the approach to ensure that the artifacts captured the design rationale of their systems.

These systems come from domains such as instrumentation and control, signal processing, central monitoring, and communication. They vary in size, complexity, and have different system characteristics that influenced the architecture design such as fault tolerance, multiprocessing, safety critical, real-time performance, interoperability, distribution, heterogeneity.

The following table lists typical categories of influencing factors based on our observations. Within each category there will be a number of factors. For example, the schedule (O4) will record the time to market and how features are to be delivered; performance (P3) will record latency and bandwidth considerations.

<i>Organizational</i>	<i>Technological</i>	<i>Product</i>
O1: Management	T1: General-purpose Hardware	P1: Features
O2: Staff Skills	T2: Domain-specific Hardware	P2: User Interface
O3: Development Environment	T3: Software Technology	P3: Performance
O4: Schedule	T4: Architecture Technology	P4: Recovery
O5: Budget	T5: Standards	P5: Diagnostics

The following table gives an indication of the kinds of strategies we found in the systems we examined.

<i>Organizational</i>	<i>Technological</i>	<i>Product</i>
Reuse existing components	Encapsulate hardware	Use feature-based components
Build rather than buy	Separate processing, control, and data	Separate the user interaction model
Make it easy to add or remove features	Use vendor-independent interfaces	Separate time-critical components

5. Experience with Using the Method

We have taught the global analysis method in courses and have observed its use as it has been applied to four additional systems as part of a forward-engineering software development process.

	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>
Application	data mgt.	image mgt.	business mgt.	automation mgt.
Factors				
Org.	14	9	28	28
Tech.	8	7	22	14
Product	7	11	28	25
Issues	11	3	19	23
Strategies	24	21	100	64

System A is representative of the way global analysis was applied. System A is a software system for acquiring and processing meter data from electrical, gas, and water meters [10]. System A performs calculations on the meter data and the results are sent to a utility's billing system. A global analysis specification was produced.

Factor tables were adopted as is. They are recorded in tables in a global analysis specification document. Columns record the factor name, description, flexibility and changeability, and impact.

Experience with System A provided evidence of the generality of the original collection of factors and categories. The author of the global analysis document was able to cut and paste many of the factors from the IS2000 system and make minor modifications to adapt the analysis to his situation. An example of such a technological factor was the database system. Although marketing specified Oracle 8 be used it was known that it would change over time. New database versions would become available and some customers would prefer databases from other vendors. The strategy for dealing with this factor was to design a layer in the architecture to isolate and encapsulate the database so that the effect of

changes could be localized and accommodated in the future.

Experience with System A reinforced the importance of considering organizational factors in addition to traditional requirements and enhanced the collection of project management strategies. An example of such an organizational factor was that company management wanted to get the product to market as quickly as possible. Since the market was changing rapidly, it was important to provide users with a subset of features so that they can provide feedback. The strategy employed to address this factor was to develop products incrementally so that scheduled release dates could be met.

Experience with System A suggested improved support for additional topics such as product lines. An example of such a product factor was to support a product line in the market place. The graphical user interface must accommodate many types of users for different applications. A web-based GUI was employed so that additional flexibility could be achieved as new applications are added and location independence achieved for the various user populations. The performance of the system must scale for higher-end applications so a scalable distributed platform was necessary to meet these more stringent calculation time requirements.

A summary of issues and strategies was documented. The summary provided a listing of the issue name with a short description, factor cross-reference by number, and strategy name. Issue cards were not documented.

The strategies have implications for the project management. Strategies were analyzed and consolidated to develop project strategy conclusions about how the system should be designed and developed. This short list of major project strategies served as guiding principles for all the development team members. These project strategies helped define the project goals and risks that must be mitigated for success.

System B is similar in scope to System A and yielded similar conclusions. Systems C and D continued to expand our repertoire of factors and strategies; but the large number of factors and strategies that needed to be considered challenged us to think about new ways of managing and ordering this information. We address this in the following section where we discuss lessons learned.

6. Lessons Learned

What value did global analysis add that wasn't present before global analysis was used?

Introducing global analysis into the software development process of new projects resulted in a global analysis specification document that helped bridge the gap between requirements and architecture design and

provided a place to explicitly record design rationale. The process of global analysis also can be used to build stakeholder consensus. In one case, a global analysis workshop was held to elicit feedback from stakeholders, discuss conflicting stakeholder requests and possible tradeoffs, and prioritize the factors.

Global analysis strategies advocated the adoption of an architectural pattern or style, provided design guidelines (encapsulation, separation of concerns), placed constraints on elements of the systems, or introduced additional structure. In essence, the strategies yielded a set of constraints on the architecture design in terms of prescribing a collection of component types and their patterns of interaction. These building blocks were developed from software engineering principles and the experience of building previous products. Component types, their relationships, properties, and constraints define an architectural pattern or style. As experience grows these patterns may be codified and the architect could select common patterns from a repository. The patterns embody a set of predefined design decisions. Constraints that emerge during global analysis could be used to select the appropriate ones.

Another benefit is improved documentation of the system. Design decisions between and within views of the architecture and the supporting rationale are recorded. The strategies are linked backward to requirements and forward to design decisions to provide traceability and validation [2].

In addition to guiding architecture design, it was not surprising to see the outputs of global analysis used by project management, since architecture plays a central role in software development activities. Issues and strategies provide input for project strategies that are used in release planning and scheduling in the software development plan. Issues also capture risks that the project manager is interested in tracking. Global analysis helps identify project and technical risks and suggest strategies for mitigating them.

What should be changed as a result of using global analysis in practice?

Many of the systems we examined had characteristics of product lines. Global analysis takes on an even more prominent role in product line design. The architect must characterize how the influencing factors vary among the products within a product line. The architect develops and selects strategies in response to these factors to make global decisions about the architecture that allows the developers of the products to make uniform decisions locally. Guiding the developers in this way ensures the integrity of the architecture. This is an iterative process. During the design, certain decisions feed back into the global analysis, resulting in new strategies.

Since product lines focus on variations among products, it would be advantageous to have separate

columns for flexibility, changeability, and variation so that more guidance can be offered and the characterization and its type of impact can be more precisely captured.

Strategies suggest solutions for addressing a problem highlighted by an issue. As the architect selects a strategy, it is being evaluated in a continuous activity that we call global evaluation. Later on, these decisions could be evaluated during an architecture evaluation exercise. It would be beneficial while the issue is being articulated to also link it to an evaluation technique such as scenarios that would provide criteria for successfully meeting the requirement. It makes sense to do so as the issue is being formed and input gathered from the architect and relevant stakeholders rather than being captured after the fact during an evaluation exercise.

What wasn't used from global analysis and needs better elaboration?

Issue cards were not explicitly documented. The information they were meant to capture is therefore missing: text describing the problem and explaining tradeoffs and the degree of difficulty, text describing the factors in relation to the problem, and the solution statement.

Instead of the issue cards, a summary of issues and strategies table was used. This could be because the first time global analysis was used the document was written by the project manager. This experience showed the need for two views of the global analysis information. Using the summary of strategies served the project management view well, but trying to use it for the architecture view in lieu of the issue cards resulted in a number of problems. This was seen in a subsequent project where an architect used the global analysis document of the first as a template.

A problem with not using issue cards is that the summary table is not easy to read, especially the factor numbers. Instead of using numbers, it would be more readable to include the factor name with a link to the factor description and analysis. Issue cards help cross-reference information among the factors relevant to particular issues. Without their use, the factor table is used to pick up the slack. But because it was not designed for this purpose, the global tradeoffs and issues are more difficult to discern. For example, factor tables are used to address tradeoffs, such as schedule vs. quality and function. The impact column is used to address analysis and the solution. Issues tend to get grouped into factor categories instead of being cross-cutting across factors.

What needs further study for improving the global analysis method?

Issue cards were inspired by design patterns [7]. Further study and codification of the artifacts is needed to see them effectively adopted in practice.

A catalog of common factors, issues, and strategies is emerging. The original list of factors and categories was

not meant to be exhaustive but illustrative. These factors were inspired by standards such as ISO/IEC 9126, the SEI taxonomy on software development risks, and our experience with numerous case study systems. Some of the additional factors we have seen include: legacy systems, global development, project engineering (for product lines), internet architecture technology (e.g., middleware, clients, and servers), scalability, and usability.

Similarly the list of issues and strategies were meant to be illustrative. Strategies are drawn from software engineering principles (loose coupling and high cohesion, separation of concerns, encapsulation), heuristics, patterns, and styles. As experience grows these strategies may be codified [1].

It would be useful to identify a core set of factors, issues, and strategies applicable to all systems. They could be used to derive a global analysis checklist used in conjunction with a template that the architect would use as an integral part of design and not be viewed as an extra documentation obligation.

A better articulation of the solution field in the issue card is needed, explaining the dependencies and tradeoffs among the strategies and how they might be used separately or in conjunction with one another.

There is value in creating a global analysis document at the beginning of architecture design to support management functions. However, global analysis is not meant to be a static document but one that evolves as the architecture is designed. The architect needs better support in this iterative process.

The global analysis data needs to be presented in different ways to different stakeholders. For example, we saw examples of how strategies were grouped by issues, by project recommendations and by architecture structure that they influence.

7. Conclusions

This paper has presented our experiences with a practical approach for analyzing the factors that influence software architecture. Approaches we have observed tend to focus on the functional requirements. But it is the quality attributes and constraints from the organization and the underlying technology that most strongly shape the architecture. These organizational, technological, and product factors are analyzed in global analysis. We have presented examples of factors based on experience and see a role for a catalog of such factors.

Global analysis helps the architect make the conceptual leap from the requirements to architecture design. Global analysis identifies factors that influence the architecture and yields a set of constraints on a collection of architecture design element types and their patterns of

interaction. Global analysis also helps the architect record design decisions made between and within views of the architecture and the supporting rationale.

These factors are constantly changing. We found that successful architects analyze factors that have a global influence to produce an architecture that localizes the effects of change. Global analysis aids the architect in designing for change and building flexibility into the software.

To help the architect in this process, we have provided a two-phase approach for analyzing factors and developing strategies. The process is iterative and may start with either phase. We have provided factor tables and issue cards to capture the information.

We have validated and gained experience with this approach in three ways. First we developed the approach informally while designing the architecture for an image acquisition and processing system. Second, we did a retrospective analysis of four existing systems, interviewing the architects to understand the process they used to go from requirements to design, and getting their feedback on the resulting global analysis approach and the artifacts captured for their systems. Third, global analysis is being taught in courses and used in new software development projects. The result is the production of global analysis documents that are used by the architect, project manager, and other stakeholders. The benefits they have realized include: documented factors and design strategies that guide the architecture design; inputs for developing project strategy conclusions, goals, and risks; and improved documentation of the architecture. These applications give us confidence that the approach is practical and helpful.

8. References

- [1] Bass, L., M. Klein, F. Bachmann, "Quality Attribute Design Primitives and the Attribute Driven Design Method." 4th Conference on Product Family Engineering. Bilbao, Spain, 4 October 2001.
- [2] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, 2002.
- [3] Clements, P., R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, Boston, MA, 2002.
- [4] Hofmeister, C., R. Nord, D. Soni, *Applied Software Architecture*, Addison-Wesley, Reading, MA, 2000.
- [5] Jacobson, I., M. Griss, and P. Jonsson, *Software Reuse: Architecture Process and Organization for Business Success*, Addison Wesley Longman, New York, NY, 1997.
- [6] Kang, K.C., S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson, *Feature-Oriented Domain Analysis Feasibility Study* (CMU/SEI-90-TR-21), Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
- [7] Meszaros, G., and J. Doble, *A Pattern Language for Pattern Writing*, 1997. URL: <http://www.hillside.net/patterns/>
- [8] Nord, R.L., C. Hofmeister, D. Soni, "Preparing for Change in the Architecture Design of Large Software Systems," Position paper accepted at the *TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, 1999.
- [9] Nord, R.L., "Meeting the Product Line Goals for an Embedded Real-Time System," In *Proceedings of the 3rd International Workshop on the Development and Evolution of Software Architectures of Product Families*, 2000.
- [10] Paulish, D.J., *Architecture-Centric Software Project Management: A Practical Guide*, Addison-Wesley, Boston, MA, 2002.

Tool Support for Scenario-Based Architecture Evaluation

Steffen Thiel, Andreas Hein, and Heiner Engelhardt

Robert Bosch Corporation
Corporate Research and Development, Software Technology (FV/SLD)
Eschborner Landstr. 130-132, D-60489 Frankfurt, Germany
+49 (69) 7909 – { 518 | 512 | 501 }
{ steffen.thiel | andreas.hein1 | heiner.engelhardt }@de.bosch.com

Abstract

Architectural considerations play a key role in the success of any software-based development project. Architecture evaluation is an early risk reduction method for identifying risks that prevent a system or product line meeting the organization's business goals and customer needs. This paper introduces a tool that supports architecture evaluation. It gives an overview on its information management capabilities and discusses development issues as well as the underlying data model.

1. Introduction

Architectural considerations play a key role in the success of any software-based development project. Architecture evaluation is an early risk reduction method for determining whether the system or product line will satisfy the desired business and quality requirements. An important prerequisite to achieve this is getting an understanding of the consequences of architectural decisions with respect to those requirements.

Unfortunately, requirements specifications are often not definitive enough in practice, neither for architectural design nor for evaluation. As a consequence, requirements must be made explicit in order to be useful for development. Scenarios are practical in this respect since they allow to describe concrete interactions between the stakeholders and the system. They are, for example, useful in understanding run-time qualities such as performance and reliability. This is because scenarios specify the kinds of operations over which the qualities need to be measured, and the kinds of failures the system will have to withstand. Therefore, scenario-based methods have been proven useful in practice for evaluating architectures during a review [1, 3].

This paper describes a tool that supports scenario-based architecture evaluation. In Chapter 2, we sketch

basic steps performed during an evaluation. Chapter 3 introduces AET, a tool developed by the authors that supports the evaluation team during a review. In Chapter 4 development issues of the tool are discussed. Finally, Chapter 5 concludes with a short summary and gives an outlook on further development activities.

2. Architecture Evaluation

The goal of an architecture evaluation is to identify risks that prevent the system or product line to be successful. Successful systems meet the organization's business goals and satisfy the customer needs.

The Software Technology department of Robert Bosch Corporate Research and Development performs architecture evaluations for business units [2, 3]. Some of these evaluations are based on [1]. Typical activities of [1] are the following:

Step	Description
Presentation	
1	<i>Present method:</i> The evaluation team describes the evaluation method to the assembled stakeholders (typically, architects, managers, marketing, integrators, testers etc.).
2	<i>Present business drivers:</i> The marketing representative describes what business goals are motivating the development effort and hence what will be the primary architectural drivers (e.g., high availability or high security or time-to-market).
3	<i>Present architecture:</i> The architect describes the proposed architecture, focussing on how it addresses the business drivers.
Investigation and Analysis	
4	<i>Identify architectural solutions:</i> Determine the central mechanisms (e.g., architectural styles or patterns) used in the architecture.

5	<i>Brainstorm and prioritize scenarios:</i> The stakeholders elicit scenarios to make business drivers and important requirements more concrete. The scenarios are then prioritized according to a ranking scheme (e.g., market importance and effort/cost).
6	<i>Analyze architecture:</i> Evaluate the architectural decisions made to achieve the high-priority-scenarios. This is supported by examining the architectural solutions from step 4 and by identifying those design elements that are affected by the scenarios.
Reporting	
7	<i>Present results:</i> Present the findings (e.g., risks) to the audience and summarize them in a written report.

In the past, the results of an evaluation have been documented in prose. As a consequence, the access on the results of those evaluations was quite inefficient and unsatisfactory. This motivated us to start the development of a database application. One goal was to create an experience repository of architecture evaluations, another to speed up information access and report generation. In the next chapter we describe the current status of the tool.

3. Architecture Evaluation Tool

The Architecture Evaluation Tool (AET) is a research tool developed by the authors at the Software Technology department of Robert Bosch. It supports a review team in documenting results and managing information during an architecture evaluation. AET makes reporting more convenient and allows exploring the content of an evaluation.

AET uses two different databases to store information: one for general data and one for project data. The general database contains static data – this means, data that does not depend on a specific system context. General data such as general analysis questions or scenarios can then be used to support the evaluation of a particular system.

The project database contains project-specific information obtained during an evaluation (dynamic data). It includes data such as qualities, scenarios, architectural approaches, and risks that have been identified and examined during evaluation.

In the following we describe how AET can be applied in practice during an architecture evaluation.

3.1. Requirements and Qualities

During the presentation of business drivers and the architecture (step 2 and 3), a lot of information about requirements and quality attributes is usually obtained

from stakeholders. This information can be recorded in AET for later exploration and reference. Business goals, functional requirements, and design constraints can be put into a requirements list. Quality attributes can be documented in a quality tree. A sample quality tree for a fictitious Embedded Vehicle Control System (EVCS) is shown in Figure 1.

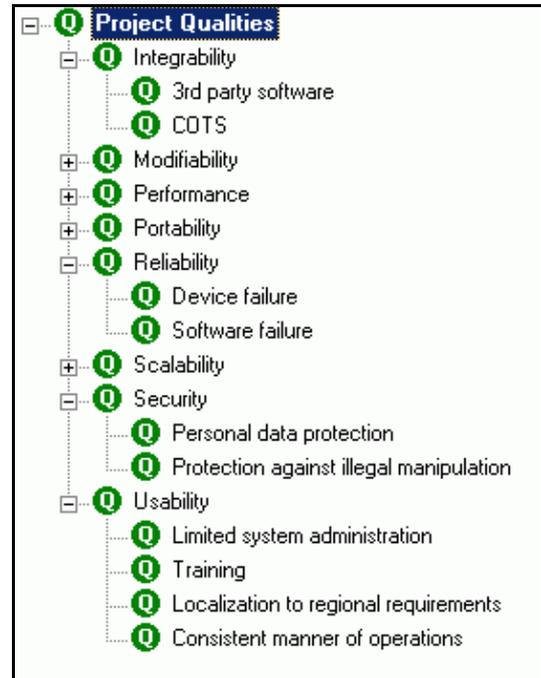


Figure 1. Quality Attribute Tree

Optionally, a starter set of typical quality requirements and scenarios for the type of systems under evaluation (e.g., embedded automotive systems) can be generated from the general database.

Each quality attribute may contain one or more sub-factors, as shown in Figure 1. Sub-factors describe specific stakeholder concerns of the quality. For example, in Figure 1 “personal data protection” is of specific concern for security. Note that each item in the quality tree can be moved easily. This allows quick modifications during an architecture evaluation.

3.2. Scenarios and Prioritization

AET allows to record the scenarios gathered in step 5 in a scenario list, as illustrated in Figure 2. Scenarios can also be described in more detail. For example, you can document potential stimuli and responses in order to make the expected behavior more concrete. In addition, you can link scenarios to a particular quality attribute or business goal in order to document that it contributes to that attribute or goal. In Figure 2, the selected scenario

contributes to “quick start up” which is a sub-factor of performance.

Furthermore, you can assign stakeholder priorities to each scenario, as defined in step 5. The scenario priorities then drive the further analysis. In the example of Figure 2, we use two dimensions (business importance and architectural difficulty) and three values (High, Medium, Low) for prioritization.

However, AET allows to adapt the dimensions and priority scale to fit individual needs of the evaluation, as shown in the lower right of Figure 2. Scenarios can easily be sorted according their priority such that the most important ones (high importance, high difficulty) appear at the top of the list.

3.3. Scenario Analysis

Each scenario can be analyzed in AET. Usually, you start with the most critical scenarios. There is room for a detailed description of the analysis results, including text and pictures. For example, you can describe the

architectural elements that contribute to a particular scenario. You may also like to document how the architecture would need to be changed to accommodate a scenario. The description is stored in HTML-format in the database for post-processing.

Furthermore, AET allows to classify important findings of the analysis. Important findings are, for example, risks or tradeoff points. Risks arise from architecturally important decisions that have not been made, yet. A tradeoff point occurs when multiple quality attributes are differently affected when changing one architectural parameter. For example, improving throughput may result in reduced reliability.

When recording a finding, AET directly links the finding to the scenario under analysis. This is very useful since it allows you to easily trace back risks, tradeoffs, issues etc. to its source – the scenario. You may follow the trace to obtain a more detailed description of the analysis. Storing traces also supports statistics, for example, about which scenarios are most critical for the success of the system.

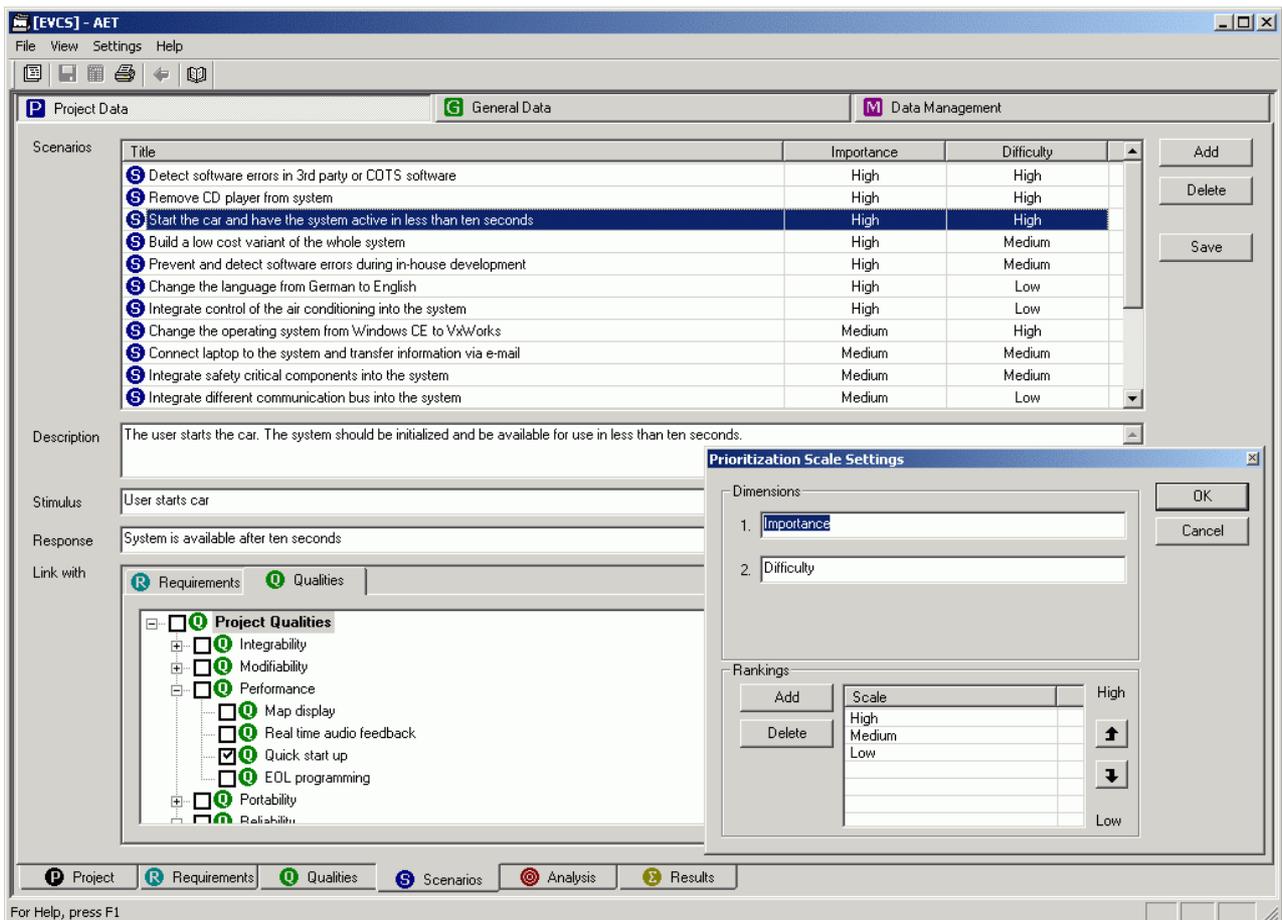


Figure 2. Classification of Scenarios in AET

3.4. Analysis Results

Since the number of risks identified during an evaluation can be high, AET allows to classify them in risk themes [1]. Risk themes summarize key architectural issues that pose potential future problems for the success of the system. For each risk theme, AET allows to assign one or more findings. In addition there is room for a detailed discussion of the risk theme. Risks and risk themes can be clearly arranged in a “result tree,” as shown in the lower part of Figure 3. This tree is automatically generated by AET based on the relationships between findings and risk themes.

AET can also generate a “utility tree,” as illustrated in the upper part of Figure 3. This tree represents a summary of the elicited scenarios and priorities together with the respective sub-factors and quality attributes. As shown in Figure 3, four scenarios have been documented to address the modifiability concern of supporting “multiple customers.”

Finally, the results documented in the AET project database can be included in a written report. The different tree views visualize the results of the evaluation in a concise form. This supports clarity and understandability of the documentation.

4. AET Development and Database Model

AET is an easy-to-use application. It is implemented in C++ and runs on Microsoft® Windows operating systems. It uses a commercial database system for storing and retrieving data which has reduced the development effort drastically. Furthermore, AET deals efficiently with its resources. It is thus a suitable companion for a mobile application at the customer site.

From the architectural perspective, AET is organized in three layers: presentation, application, and data management. The presentation layer is responsible for user interaction and data presentation. Data post-processing such as scenario sorting or combining data from different database tables is done in the application layer. The data management layer provides low-level services to access and maintain the database.

Figure 4 shows a simplified model of the project database. For each evaluation project you can record individual requirements, scenarios, architectural decisions, findings, and risk themes. Priority dimensions and scales are global to a project. Each scenario can have a ranking. The ranking must conform to the global scheme defined for the project.

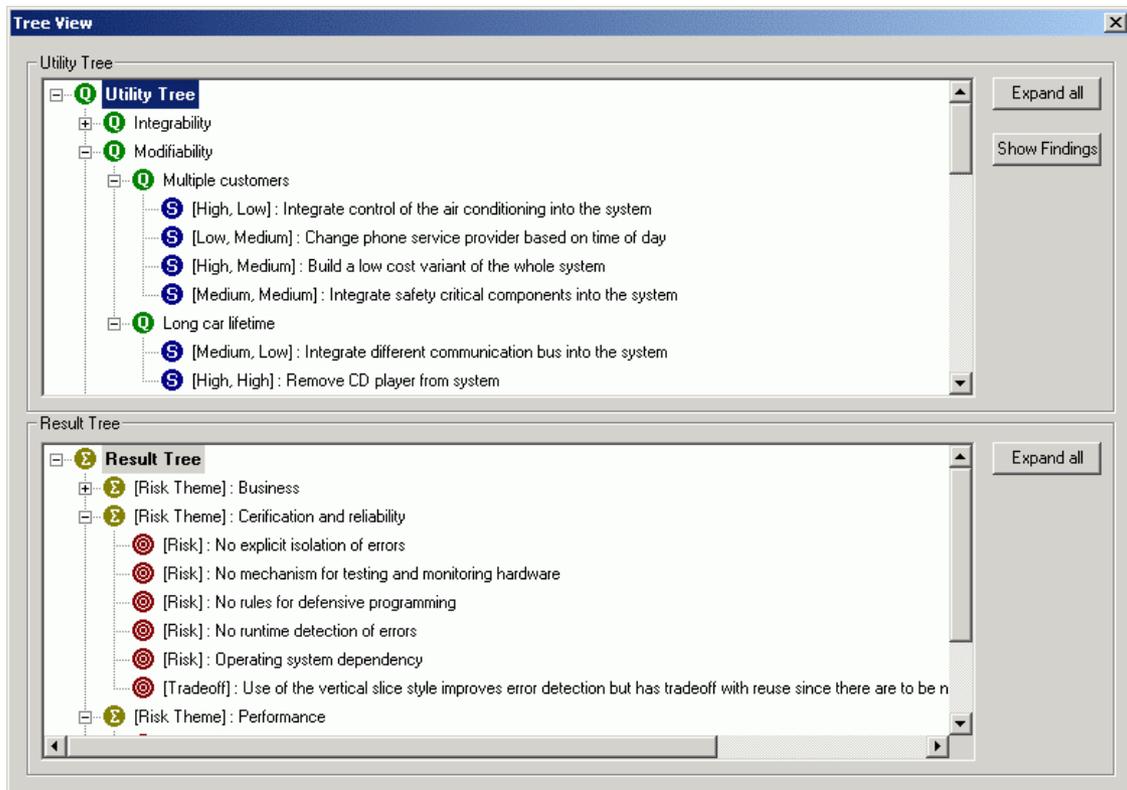


Figure 3. Tree View of Qualities, Scenarios, and Analysis Results

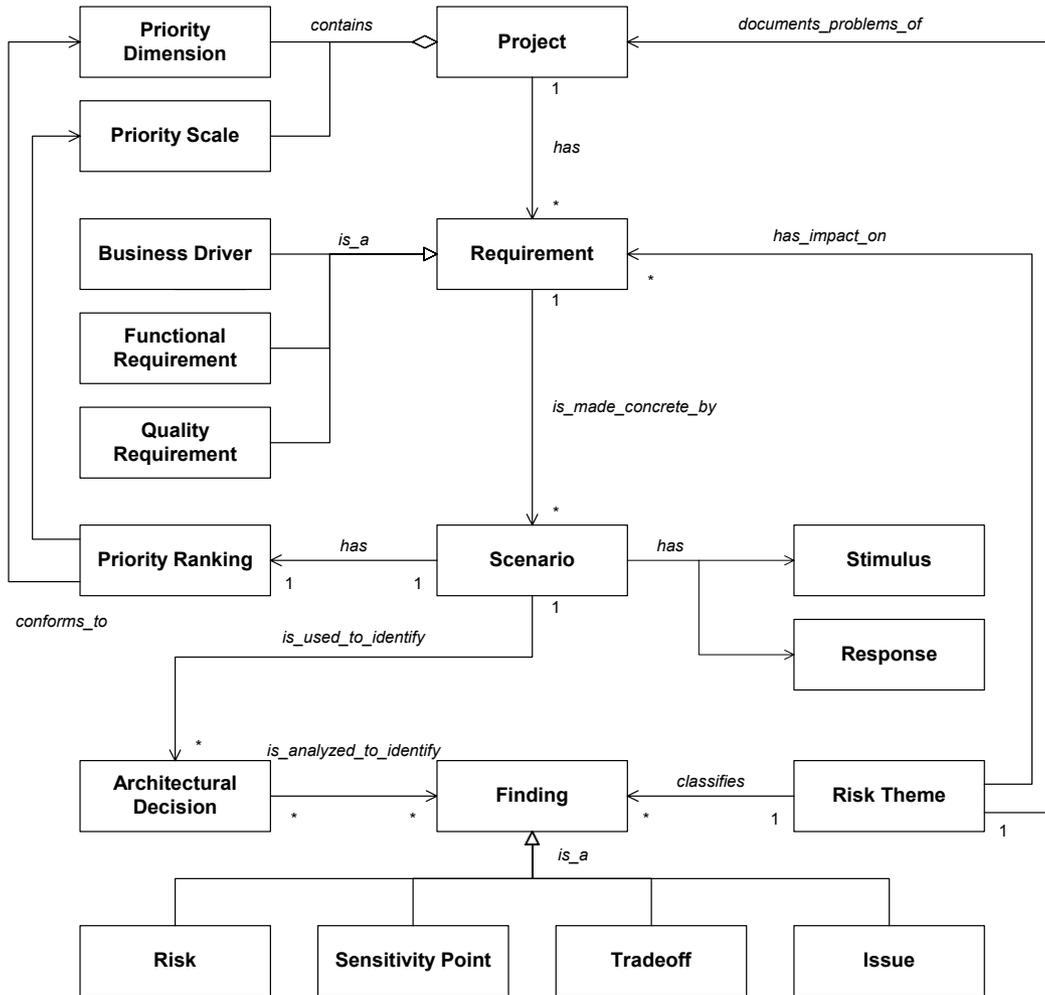


Figure 4. AET Data Model

Scenarios have a stimulus and a response. They are explored in order to identify architectural decisions. These decisions can further be analyzed to identify particular findings. A finding can be a risk, sensitivity point, tradeoff, or issue. Risk themes classify and summarize findings. Each risk theme has an impact on one or more requirements – this means, some of the business drivers or qualities cannot completely be met. The risk themes document the problem areas associated with the system under evaluation. They indicate how close an organization is to fielding a successful system.

5. Summary and Outlook

In this paper we have introduced AET, a tool that supports scenario-based architecture evaluation. We first discussed typical steps of an architecture evaluation. Next, we gave an overview of AET, and, finally, we sketched development issues and the AET data model.

AET is still under development. We plan to improve the export interface for report generation and to include functionality for querying the project database and for performing evaluation statistics.

6. References

- [1] P. Clements, R. Kazman, M. Klein: **Evaluating Software Architectures: Methods and Case Studies**; Addison-Wesley, 2002.
- [2] S. Ferber, P. Heidl, P. Lutz: **Reviewing Product Line Architectures: Experience Report of ATAM in an Automotive Context**; Proceedings of 4th International Workshop on Product Family Engineering (PFE-4), pp. 364-382, Bilbao, Spain, October 3-5, 2001.
- [3] S. Thiel: **On the Definition of a Framework for an Architecting Process Supporting Product Family Development**; Proceedings of 4th International Workshop on Product Family Engineering (PFE-4), pp. 125-142, Bilbao, Spain, October 3-5, 2001.

Experiences from the Architectural Change Process

Josef Nedstam

Department of Communication Systems
Box 118, SE-221 00 Lund, Sweden
josef.nedstam@telecom.lth.se

Even-André Karlsson

Q-Labs
Ideon, SE-223 70 Lund, Sweden
even-andre.karlsson@q-labs.se

Martin Höst

Department of Communication Systems
Box 118, SE-221 00 Lund, Sweden
martin.host@telecom.lth.se

Abstract

A good software architecture is becoming recognized as a major factor for successful products. There has been much research on the technical aspects of software architecture and it is recognized that the driving requirements for architectures are "non-functional", but few have studied how organizations decide on architectural changes. In this paper we study the topic through several case studies. The changes to the architecture are in all cases changes to the "non-functional" requirements on the system. Issues that we want to evaluate are: when and how is the need for an architectural change discovered; what is the underlying non-functional requirement; who drives the change; how is it prepared and evaluated; and finally, who makes the decision and how is it implemented.

Through interviews with people that have experience from architectural changes we compare the decision process for architectural changes to the ordinary functional requirement change process and the organizational change process. We find that architectural changes have aspects of both functional and organizational changes. An architectural change does not only need to be technically sound, it also needs to be anchored firmly in the organization. This report gives both architects and managers guidelines to balance short-term project goals and long-term organizational goals with respect to architecture.

1. Introduction

Software architecture is becoming a well-established field in technical terms, i.e. the different types of architectures have been characterized [1]; different useful views of the architecture have been described [2, 3]; as well as books covering the whole area, e.g. [4, 5, 6]. However, little research has been done on how decisions on architectural changes are made in organizations.

Architectural changes are often different in nature from other functional changes. They can impact larger parts of the product, they can imply new ways of working, they are often not clearly connected to one customer requirement, and they are often expensive to implement. Functional changes often originate from a customer demand and are the responsibility of a defined role in a company, i.e. product management. Architectural changes, on the other hand, often emerge from various sources, and roles are seldom defined to drive such

changes. All these factors imply that they differ from pure functional changes.

The process for taking decisions regarding functional changes and features has received attention in recent years [7, 8]. Software development processes generally support this rather well. When it comes to decisions regarding the software architecture, the architect is often not so well supported, neither for the analysis of the technical impacts nor the organizational aspects of the change.

Since architectural changes have impact on organizations they might be best compared to the organizational change process, as defined by Kotter [9]. Kotter's eight-stage process describes how to prepare an organization for major change, and how to anchor the change in the organization:

1. Establishing a sense of urgency
2. Creating the guiding coalition
3. Developing a vision and strategy
4. Communicating the change vision
5. Empowering employees for broad-based action
6. Generating short-term wins
7. Consolidating gains and producing more change
8. Anchoring new approaches in the culture

These steps will be referred to in the overview of the suggested process for architectural change in Section 3.

This paper examines how several changes to the software architecture have been handled at three software development organizations, and what internal or external forces that drive the need for changes and control which solutions are decided upon. Concretely we have looked at the following questions for each architectural change:

1. What is the architectural change?
2. Why was the architectural change needed?
3. Who initiated it?
4. How was the associated decision made?

Based on the analysis of these questions, the ordinary process for deciding on functional changes, and theories for organizational change, we propose a process for handling architectural changes, which provides guidelines to consider in each step.

The three companies involved in this study are industrial partners of the Center for Applied Software Engineering at Lund University (LUCAS). Part of LUCAS is the LUCAS Architecture Academy that is a one-year part time software architecture education program for the LUCAS partners. This research is done based on issues that came up in the context of the architecture academy.

2. Method

In this study we have studied seven architectural changes initiated at three Swedish software-developing companies. The project has included a number of sessions where the companies present their architectural work for each other, and issues in the area have been raised and elaborated.

The approach taken in this research can be described as *flexible* [7]. This type of research is characterized by less pre-specification than in, for example, controlled experiments. In a flexible design the major research questions can be specified in advance, although they must be allowed to evolve during the course of the research.

Qualitative data has been collected in two sets of interviews. The first set was held with architects and system designers at the three companies to collect information about the companies, their products, and their architecture. Recent architectural changes were identified. Key persons in those changes were interviewed in a second set of interviews. These interviews were guided by the four questions mentioned in the introduction. The data was then analyzed according to the following factors:

- Architectural change
- Phase of change process
- Topics that were considered important in changes

The collected data was categorized and tabulated according to these factors, and analysis was carried out through discussion and pattern searching.

3. Process Overview

This section describes our suggested process for making technical decisions. The process is illustrated in Figure 1 and has been derived from the case studies, Section 4. The relation between the process and the case studies is shown in Section 5. The purpose of this process is to enable organizations to make the right decisions by the right people at the right time. From an employee viewpoint the process shall give guidance in the decision process, both for change initiators and decision-makers. Note that one aspect that differentiates the architectural change from the functional change is that the functional change usually is initiated by a customer request, and there is usually someone in the organization dedicated to handling these, e.g. product management. Architectural changes can be initiated by many roles in the organization.

The general process for functional changes involves requirements elicitation, pre-studies, implementation, and related decision-points. It focuses on how an organization

shall *make decisions*. Kotter's [9] process for large-scale organizational change instead focuses on how to *make changes happen*. In the following process the two features are combined. This has basically been done by mapping Kotter's change process onto the functional change framework, which is considered to be fairly established in software industry. In practice the process therefore has to be adapted to the present functional change framework.

1. **A need emerges:** The process is superceded by a chain of events where need for change emerges or is created, and someone, the change initiator, sees this need and considers it his or her responsibility. This can to some extent be compared to Kotter's *Establishing a sense of urgency*, and to requirements elicitation in a functional change process.
2. **Initial decision preparation:** In this phase the change initiator does preparations with the goal of getting resources to analyze and implement the change.
 - Document background: To increase the chance of having an impact on the resolution of the need, the change initiator should document the background of the need, i.e. what products, components or organizational entities are involved, the history behind the need, how it manifests itself, what effects it might have not to satisfy the need etc.
 - Identify stakeholders/decision makers: While documenting the background, stakeholders are sure to emerge. In order to have optimal impact, the change initiator should pay special attention to these and especially to the decision makers that will be involved in the following process. This is related to Kotter's *Creating the guiding coalition*.
3. **Decision: Go/no-go:** An initial decision must be made whether the issue at hand is adequate and feasible to treat. Probably, there has not been spent very much effort before this decision point, e.g. one person's work for hours or days. Work done in the rest of this process, but before a decision on any particular solution or implementation of change, probably requires resources that must be budgeted, e.g. a handful of persons or more, which work for days or weeks. Therefore a person responsible for resources must make a decision whether to go on with this process or not. The formality of this decision-point is controlled by the organization at hand. If the change

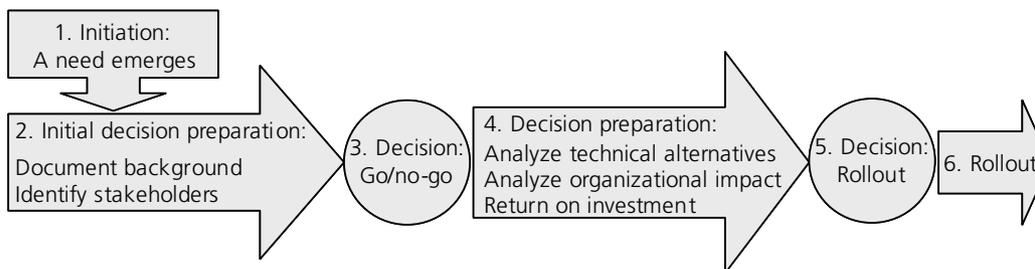


Figure 1. The process of architectural change

can be viewed as a normal product requirement or change proposal, it can be treated as such through the ordinary channels: implementation proposal and related decision points. If the change however is more of a change in the way people work, or a change in an internal quality attribute not leading up to completion of a specific project, the process steps that follow are of a different complexity. The risks of facing opposition are higher and the decision process and preparations must be more thorough.

4. **Decision preparation:** This phase is akin to performing a pre-study or developing an implementation proposal in technical change management. In terms of Kotter's process, it resembles *Developing a vision and strategy*.
 - Analyze technical alternatives: When technical alternatives have been proposed, these can be analyzed from an architectural viewpoint in a number of ways [11], i.e. ATAM [12].
 - Analyze process and organization impact: When making a technical analysis, the organizational implications are often forgotten. This might lead to unexpected resistance to a change. An organizational analysis is therefore made, based on the initial analysis of stakeholders, in order to assess the impact of the change and prepare the organization for the change. The activity therefore contains parts of Kotter's *Communicating the change vision*.
 - Return on investment: The need that the change satisfies has to have a financial side. A return on investment analysis will simplify getting support for the change from top management and management of any project that might implement the change. This activity will support Kotter's *Generating short-term wins*.
5. **Decision: Rollout:** Software projects generally have a tollgate or decision point where it is decided which implementation proposals will be included in the resulting product. The same decision is made in this phase, regarding technical aspects of the architectural change. Organizational changes are however not suitable to implement in a product oriented project, and will therefore need another form of implementation and associated decision.
6. **Rollout:** This activity involves the implementation of the change. The objective of this process is that the rollout of the technical part of the change shall be carried out within an ordinary project, i.e. where generally most organizational resources are allocated. This has to be synchronized with the rollout of the organizational change, which must be managed by, and given resources from, the line organization. This activity is related to the late phases of Kotter's process: *Consolidating gains and producing more changes*, and *Anchoring new approaches in the culture*.

When comparing to Kotter's process it is important to keep the proper context in mind. Kotter presents a process for long-term organizational changes, which means some phases are of a different scale. Kotter's process also focuses on engaging employees and preparing an organization for a change, and not so much on how to perform the actual change. Since this paper focuses on changes to software architectures, we can use the decision framework common in software projects as a basis for a change process with features of both perspectives.

4. Case Descriptions

This section describes architectural changes at three companies, located in southern Sweden. All companies develop products to a mass-market, and their products have long lifetimes. This implies that their architectures need to support several simultaneous versions of their products, with several releases over an extended period of time.

4.1 Company A

Company A develops control system environments for industrial automation, e.g. chemical plants, dairies, oil platforms, etc. The control system environment consists of both a development view, called control builder, and a deployment view, i.e. the controller itself. Within the control builder, controllers can be designed by specifying hardware sensors and actuators, constructing control loops, and connecting variables in those control loops to the hardware devices. A fully specified system can then be compiled and deployed onto a controller in a control system.

Company A typically carries out one large project at a time, involving the entire organization. Each project evolves the same product further by adding features to the control builder, e.g. new editor facilities, and the controller, e.g. new hardware interfaces. Implementation proposals are developed during a feasibility study. Accepted implementation proposals pass a tollgate, where after implementation begins. Development is organized in teams, each working on a number of implementation proposals. Work is feature-focused and the organization has no module-responsible and no architects, but instead relies on senior developers to take responsibility for long-term architectural goals. Two changes were studied at the company:

Protocol Framework: Company A recently acquired companies within their domain in order to increase their market share. The controller developed by Company A was intended to replace those companies' products. To support the same customers, the controller therefore had to support a number of legacy protocols from those products. This was realized as a problem using the present architecture, as the protocols were intertwined with the rest of the code, and could only be developed at one site, the one studied here. This site only had capacity to develop 1-2 new protocols per project. To be able to develop

several protocols a year, Company A decided to develop a generic IO and communication protocol framework. The solution was developed through a pre-study and an implementation proposal, which resulted in a solution that enabled frequent releases of the product with many new or legacy protocols in each release. This would be accomplished by letting other departments of the company develop the protocols they were responsible for, using the protocol framework.

Real-Time Operating System: Company A had for a number of years had discussions about cutting licensing costs on Real-Time Operating Systems (RTOS). A suggestion from local product management at the studied site to develop their own RTOS was rejected by local development management. In parallel, high-level management decided to reduce the number of RTOSs to only one. This would not only lower licensing costs but also provide focus on a common competency regarding RTOSs and tool support, which would standardize and simplify distributed development. Top-level development management initiated a pre-study across all departments of the company. Participants were interviewed regarding their use of, and competencies in RTOSs. The site studied here used one RTOS, but the pre-study led to a recommendation for all departments to switch to another. Eventually the recommendation became a requirement for a project at the studied site. This requirement was postponed by the local organization, while an OS expert prepared a solution with a Virtual Operating System (VOS) layer, which was introduced in a later project.

4.2 Company B

Company B develops platforms for consumer electronic devices. These platforms are sold to external customers who configure the services within the platform to create complete products. The software platform consists of a number of modules, and a middleware layer hides the internal architecture from the customers.

Projects are organized in: a project management group, with product management responsibility; a system-engineering group, with expert groups and function groups responsible for major features within market requirements; and a system realization group, which receives specifications from the system engineering group and develops the platform. The system realization group is divided into a hardware- and a software branch, which are subdivided into development teams responsible for a set of modules. The organization has module responsible that work with function groups during specification and development teams during implementation. The company also has a dedicated architecture group that performs most of its work within projects, especially supporting and influencing the system-engineering group. Three changes were studied at the company:

Data Router: During routine reviews the system-engineering group discovered several modules handling data streams in similar ways. These modules could instead

use a common data router and thereby save memory. The architecture group developed a design proposal that was approved, but no resources were provided from the project. Project management did not consider the memory savings to be large enough. Therefore the solution was implemented by the software architecture group, and integrated with a small-scale system on an isolated branch of the code. After inspection this branch was merged with the main track, and the software architecture group initiated documentation and education on the new architectural mechanism. The solution was still not widely accepted, as most modules already had their own implementations of the same functionality.

Hardware Abstraction Layer (HAL) Split: The bottom layer of the architecture had existed in previous versions of the product, but had not been formally defined, and therefore there had been no clear rules as to how to access the hardware. The hardware was also not encapsulated well enough from the majority of the software, leading to unnecessary impacts in the software when the hardware changed. The developers working in the lower layers of the product realized the need for a clearer definition of these layers. They proposed a solution that meant clearing the HAL interface from hardware dependencies, i.e. creating a logical layer on top of the previous HAL. One driving force for introducing this logical layer is that the cost for a product developed from the platform is very dependent on the hardware components used, and therefore these are often changed to provide cheaper solutions. The purpose of the logical layer is to allow such changes without expending effort in the higher layers of the software.

The solution was presented for the system-engineering group and brought to the software architecture group. When the proposed solution was established within the system-engineering group and the software architecture group, project management decided to assign resources to the change. The software architecture group introduced new coding rules according to the suggestion and made changes to the architecture descriptions. At the same time, the developers in the HAL prepared by planning the change, before doing the actual implementation when resources were assigned and the architecture was updated.

Include-file Reorganization: The software architecture group had created a flexible structure for the source- and include-files. The design rules that enforced this structure required several files for each component, and when the number of modules grew to around 100, unexpected effects on the development tools emerged. Compilation times increased, the configuration management system behaved sluggishly and the globally distributed CM servers started to crash more frequently. The persons responsible for tool support within Company B were in contact with support personnel from the tool supplier, who identified the problem as having too many files in the system. The software architecture group was assigned to create a new structure.

The flexibility provided by the original structure was only needed by a few of the about 100 modules, and these could continue to use the previous structure. The rest of the modules were given a new structure, which basically involved merging three or four source files into one file. This resulted in a three-to-one reduction of source files.

4.3 Company C

Company C develops software engineering tools. One of their main products is a design tool that consists of a front-end with editors for various types of diagrams and source code, and a back-end for compiling the diagrams into code. Other utilities such as a simulation tool are also part of the design tool.

Company C releases a new version of their product every six months, and successive release cycles overlap. Features are implemented by development teams in an assembly-line fashion, described in [13, 14]. The organization has architects per project but no established line organization for architecture, and module responsibility is assigned to senior experts. Two changes were studied at the company

Communication Mechanism: New requirements, especially related to new language standards, have meant that the old architecture could not support further development. Therefore top-level management decided to create a new product generation. Company C had recently acquired other companies, which developed software engineering tools that were to be integrated into the new product. One of the problems with the new requirements was an increase in the number of diagram editors. The old communication mechanism did not support this increase, but one of the acquired companies had recently solved that problem, using a common object model. A technical discussion led to a consensus of using the new solution, although it meant major architectural changes.

Editor Framework: The editor framework used to develop graphical editors was also changed using a more generic solution, a decision also taken by consensus in the development project. The drivers for this change were increased reuse of common editor elements, and outsourcing of development throughout the organization. Several other decisions in this change process had to be enforced by the responsible architect, as consensus could not be reached. Both these changes were introduced in the same project.

5. Analysis of Process versus Cases

This section compares the process suggested in Section 3 to the architectural changes described in Section 4.

5.1 A Need Emerges

Before the suggested process is initiated a need for a change somehow appears. The reasons for changes in this report has included business decisions to increase market share, lower costs and lead time, but also more technical

reasons where the architecture has not been able to support increased complexity and new features.

In Company A the need for the protocol framework was initiated when top management decided to increase the market-share by acquiring other actors in the same domain. Mid-level managers and experts then saw the need for support of legacy protocols found in the newly acquired companies' products. The need for a change of RTOS on local level came from a higher-level need to save licensing costs and focus competencies by reducing the number of RTOSs. The process was initiated by higher-level management and supported by developers at other sites of the organization.

In Company B the introduction of a data router was driven by memory size being an important quality attribute. The opportunity to save memory was discovered by system engineers during routine code-reviews. The need for a HAL split emerged as the company wanted to be able to change hardware components frequently in order to save costs. The hardware-related developers themselves initiated the change in order to simplify the frequent changes. The need for an include-file restructuring became apparent, as the configuration management tool did not support the existing structure. The architecture group initiated this change since they were responsible for the include-file structure.

The product generation shift performed in Company C contained two major changes. A new mechanism, which allowed different editors to work against the same system representation, was introduced in order to increase the number of possible editors. A framework for editor development was introduced in order to increase reuse of common editor components and enable outsourcing of editor development. Local experts initiated these changes and the technology came from the newly acquired companies.

Change initiators have been identified from all levels of the companies, i.e. managers, experts appointed when the issue came up or as part of their ordinary role, where a special case is the architects themselves, and down to the developers. This can be compared to functional changes where needs often emerge from customers and are taken care of by marketing or product management.

5.2 Initial Decision Preparation

A decision process that can be initiated by non-decision makers will eventually have to be brought before a decision maker. In this phase the change initiator documents the background of the issue, and identifies stakeholders and decision makers.

When the need for legacy protocol support had emerged in Company A, local experts and managers analyzed the protocol framework solution in a pre-study. Limited attention was however paid to other departments that were supposed to implement protocols on this framework. Regarding the change of RTOS, the pre-study had been carried out by higher-level management. This re-

sulted in recommendations to change to a single and specified RTOS. The pre-study involved interviews on all company sites.

The introduction of a data router in Company B was initially prepared by the system-engineering group by marking the places where similar functionality had been found. Stakeholders such as current users of such functionality and future clients to the data router were loosely identified but not further analyzed. The only stakeholder that was approached was the architecture group, who would be responsible for developing an implementation proposal. Regarding the HAL split, the developers in that layer prepared a solution themselves, and set up a meeting with the appropriate decision-makers, in this case the system-engineering group. In the case of the include-file restructuring, the initial preparation was made by the tool-vendor's support organization. They concluded that the projects contained too many files. The architecture group was identified as a stakeholder, since they had developed the previous structure. Apart from that, stakeholder identification was not done actively, since the frequent tool failures meant that stakeholders presented themselves.

In Company C the first steps of the product generation shift were taken on many levels, both within the original organization and by developers and managers in newly acquired organizations. Technical discussions were held which led to the realization that the whole architecture had to be changed. Solutions were gathered from all parts of the organization, and the new architecture was adapted to enable distributed development. Stakeholders and decision-makers were therefore covered.

In the case studies we have seen examples of less successful changes, where too little has been known about the impact of the change. Effects of functional changes are often more limited and customer-oriented. As opposed to architectural changes, functional changes often have resources allocated to this phase, such as product management performing requirements elicitation.

5.3 Decision Point: Go/No-Go

In this activity the first decision to commit resources is made. The right decision maker shall have been defined previously, and process and organizational issues must not be forgotten in this decision.

In Company A, local level management decided that an implementation proposal of the protocol framework should be developed, since the solution would allow for more protocols and more frequent releases of the product. The organizational impact was not given much focus in this decision. Regarding the RTOS switch, top management decided to turn the recommendation into a requirement for the following projects. This requirement was later postponed by the local organization.

In Company B, the system-engineering group decided that the architecture group should develop an implementation proposal of a data router. Regarding the HAL split, the solution was so well prepared by developers that nei-

ther the system-engineering group, nor the architecture group had to invest large resources in preparation, and therefore the related decision was of little significance. Regarding the include-file structure the architecture group themselves decided that they should develop a solution. Resources spent by the architecture group were considered insignificant in comparison to the resources wasted during tool problems. Organizational impact related to difficulties in rolling out the new structure was considered at this stage.

In Company C, the decision to apply resources to the change process was at a higher level, since it involved starting a whole new line of product-oriented projects. A decision was therefore made by top management to prepare and plan for a first project, which should result in a prototype for the product.

A forum for architecture issues could be helpful when making this decision. Considering functional changes, organizations sometimes have product management fora, making similar decisions. The problem for the architect is that the decision is one of resources, for which the architect seldom has responsibility. Getting project resources has a benefit since the change can be more easily embraced by that project. It is however not trivial to receive resources from a project manager.

5.4 Decision Preparation

In the decision preparation phase a small group of people will analyze technical alternatives, process and organizational impact, and return on investment. From a company viewpoint this is done to make the right decision, and from an architect or change initiator viewpoint this will help convincing people of the need for change. This phase is similar to developing an implementation proposal when making a functional change, and should therefore be adapted to how implementation proposals are handled within the organization. The analysis of technical alternatives can be done in parallel with the analysis of process and organizational impact.

At Company A, the protocol framework was prepared by developing an implementation proposal, in the same way as a normal requirement. The technical solution was based on expert opinions. The process and organizational impact was considered, and a pilot study was made which involved developing a protocol at another site in the same company. However, there are many developers in the acquired companies that are impacted by this change but have not been involved in the first phase. The change of RTOS was postponed to a later project, and in the meantime an OS expert prepared a solution involving a VOS layer to allow for several operating systems. One organizational impact was overlooked, as the change meant that new RTOS support contacts had to be established. Regarding return on investment, the change of RTOS led to no short-term wins for the local organization.

In Company B, the architecture group developed the data router solution in a pre-study. It was based on already

implemented solutions, but the group failed to realize opposition from project management and developers. A return on investment was calculated late in the process. The developers had already prepared the HAL split so the architecture group only had to prepare changes to architecture documentation and design rules. No quantitative return on investment was made but the ability to change components was considered an obvious benefit. Regarding the include-file restructuring, the architecture group found that the flexibility provided by the original structure was only needed in a few modules, and a simpler structure was created for other modules. Return on investment calculations were made regarding the rollout, since rollout was expensive and did not contribute directly to any product.

In Company C the first project of the new product generation was planned. When making technical decisions many parts of the organization were involved, and consensus in joint forums was the goal. When this could not be reached, the architect responsible for that type of functionality had to make the decision. Organizational impact was not only considered when selecting solutions, but also when distributing development of various modules. This distribution could at least in one case have been better planned, as they ended up with developing a module at one site, which was highly dependent on two other modules at another site, leading to unnecessary problems.

5.5 Decision Point: Rollout

When a feature-oriented implementation proposal is completed, it is generally passed through a tollgate in the project. In this tollgate the project decides which features or implementation proposals shall be included in the upcoming release. The activity described here is similar, but the changes we have studied have had organizational impact. Such changes, and their related decisions, are hard to make in a product-oriented project, i.e. a project that will result in a product aimed at the market.

In Company A, the implementation proposal for the protocol framework involved two different types of protocols, and a set of services for these protocols. The decision to implement was made according to the standard project model. Both protocol types were to be implemented in the upcoming project, but a part of the services were postponed to later projects. Regarding the change of RTOS, the expert's VOS solution was chosen, and local development management decided to roll it out onto a current project. This project had to start implementation before the VOS was ready, and therefore local development management decided that the VOS team would make relevant modifications of the project's code when the VOS was ready.

In Company B the system-engineering group approved the implementation proposal for the data router, but the architecture group did not receive project resources to implement the proposal. They then decided to implement the data router with their own resources. The

HAL split was however granted resources by project management, because it had backing from developers, system engineering, and the architecture group. The include-file restructuring was urgent, but difficult to roll out. First a script was developed that would automate rollout. This script depended on that the design rules had been followed, which was not the case. A second strategy was to halt development over a number of days, and perform the changes manually. This solution was too costly and eventually appropriate line management decided to roll the new structure out onto newly started projects, letting old projects use the old structure.

Company C decided to launch the series of projects for the new generation of products. Top management took this decision, and the content of each project has slowly been decided throughout the first projects by top management, product management and project management.

One conclusion from this activity is that it might be beneficial to restrict functional content of a new product when introducing major architectural changes. This was adequately done when introducing the IO and communication framework in Company A, as the number of services available to the protocols was restricted in the first release. Company C has however had problems deciding on the final content of the first product to be released on the market. Restriction of functional content is a tradeoff since customers will not accept lower functional content, and the new architecture must be able to support future functional content. Another tradeoff regarding how many future features an architecture should enable concerns the debate of programming for the future or, as XP [15] advocates, programming only for the present.

5.6 Rollout

Implementation of technical aspects of changes is made successfully within product-oriented projects. Implementing technical aspects elsewhere is more problematic, since such implementations are not so easily embraced by developers in projects. The problem is that the process and organizational aspects are often forgotten in product-oriented projects, and there seldom exists a standard routine for carrying out such changes, as opposed to carrying out a product-oriented project.

In Company A the protocol framework was implemented as part of a product-oriented project, but many departments that were intended to develop protocols have not yet had opportunity to give feedback on the framework. There is therefore still a risk that some departments will object to the framework. The VOS was developed in parallel with a product-oriented project. When the VOS was ready the two projects were merged, and the VOS team had to make remaining modifications.

In Company B the architecture group developed the data router on an isolated branch, which was later merged with the main branch. The problem was that most of the clients to the new data router already had implemented their own solutions, and usage of the router was only rec-

ommended, not required. It has therefore not provided the anticipated memory savings. The HAL split had been well prepared by both developers and architects before decisions were made, and it was rolled out as part of a project. The new include-file structure was rolled out onto one project at a time across the whole organization. The roll-out coincided with an architectural change, which led to little overhead when the key module responsible checked in the new file structure into the tool at project startup.

Company C has implemented their architectural changes in a prototype project, and a product for the market is under development. The main problems have been to settle on feature content, and as previously mentioned, the distribution of work.

In the case studies we have seen several examples of changes where the technical part has been assigned to a certain project as a requirement, but postponed to later projects. We have also seen examples where the changes have been performed outside of product-oriented projects, further decreasing the chance of embracing the change. One of the cases made a satisfactory tradeoff, where the change of operating system was postponed to a later project, but prepared by an expert ahead of the project start.

6. Conclusions

In the case studies we have seen that need for architectural changes can emerge from various sources, and that various roles, such as managers, architects and developers, may take responsibility for initiating the change. The decisions regarding architectural changes are often carried out in the same way as companies make decisions regarding functional changes, while the implementation of architectural changes may take many forms, such as part of ordinary projects, parallel but separate projects, independent smaller projects or as new full-scale projects.

We have discovered three major differences between functional changes and architectural changes. First of all, architectural changes are often more complex than functional changes and affect large parts of the product without showing a clear connection to a customer need. Secondly, architectural changes do not only have impact across large parts of the product, but often across the whole organization, and changes of processes and organization are often overlooked and hard to implement in product-oriented projects. Finally, while companies often have mechanisms and resources in place to treat functional changes, such mechanisms are seldom established for architectural changes, and it is also hard to commit resources to activities without clear customer value.

We believe the process presented here helps putting focus on organizational issues in an architectural change, while taking advantage of the decision support found in the ordinary functional change process. This will lead to that the technical part of the architectural change is implemented according to company standard, hopefully within a product-oriented project.

In further studies, the process presented here could be optimized by running it in pilot studies. A goal of such studies could be to find a framework for implementing the organizational part of the change.

Acknowledgement

We would like to express our gratitude to the companies and individuals participating in the LUCAS architecture academy. Without their help this work could not have been completed.

This work was partly funded by The Swedish Agency for Innovation Systems (VINNOVA), under a grant for the Center for Applied Software Research at Lund University (LUCAS).

References

- [1] Garlan, D. and Shaw, M., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996
- [2] Soni, D., Nord, R., and Hofmeister, C., "Software Architecture in Industrial Applications", Proc ICSE, pp. 196-210, Seattle, 1995
- [3] Kruchten, P., "The 4+1 View of Architecture", IEEE Software 12 (6), pp 42-50, IEEE, 1995
- [4] Bass, L., Clements, P. and Kazman, R., *Software Architecture in Practice*, Addison-Wesley, 1998
- [5] Hofmeister, C., Nord, R. and Soni, D., *Applied Software Architecture*, Addison-Wesley, 1999
- [6] Bosch, J., *Design and Use of Software Architectures*, Addison Wesley, 2000
- [7] Evans, R., Park, S. and Alberts, H., "Decisions not Requirements: Decision-Centered Engineering of Computer-Based Systems", Proc IEEE Int. Conference and Workshop on Engineering of Computer-Based Systems, pp. 435-442, 1997
- [8] Regnell, B., Paech, B., Aurum, A., Wohlin, C., Dutoit, A. and Natt och Dag, J., "Requirements Mean Decisions! - Research issues for understanding and supporting decision-making in Requirements Engineering", Proc 1:st Swedish Conference on Software Engineering Research and Practice, pp 49-52, Blekinge Institute of Technology, 2001
- [9] Kotter, J. P., *Leading Change*, Harvard Business School Press, 1996
- [10] Robson, C., *Real World Research, 2nd Ed.*, Blackwell Publishers Inc., 2002
- [11] Dobrica, L. and Niemelä, E., "A Survey on Software Architecture Analysis Methods", IEEE Transactions on Software Engineering 28, pp 638-653, IEEE, 2002
- [12] Clements, P., Kazman, R. and Klein, M., *Evaluating Software Architectures*, Addison-Wesley, 2002
- [13] Regnell, B., Berenmark, P. and Eklund, O., "A Market-Driven Requirements Engineering Process", Journal of Requirements Engineering 3, pp 121-129, Springer-Verlag, 1998
- [14] Höst, M., Regnell, B., Natt och Dag, J., Nedstam, J., and Nyberg, C., "Exploring bottlenecks in market-driven requirements management processes with discrete event simulation", Journal of Systems and Software 59, pp 323-332, Elsevier Science Inc., 2001
- [15] Beck, K., "Embracing Change with Extreme Programming", IEEE Computer, October 1999, pp 70-77

Pattern Oriented Software Development: Moving Seamlessly from Requirements to Architecture

M S Rajasree , P Jithendra Kumar Reddy, D Janakiram
Distributed & Object Systems Lab
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Chennai, India
{rajasree, jithendra}@cs.iitm.ernet.in, djram@lotus.iitm.ernet.in

Abstract

Requirements Engineering (RE) deals with the early phases of software engineering namely requirement elicitation, modeling, specification and validation. Architecture of a software system emphasizes the structural constraints imposed on the application. Potential reuse in the form of software patterns are available for software designers to structure their applications. This paper proposes a pattern oriented methodology for software development. Using this approach, the skeleton of the application can be perceived up-front by using knowledge of previously identified patterns. Functional requirements of the application can subsequently be made evolving around this basic structure. The methodology thus bridges the gap between requirements specification and the architecture of the application. This approach not only leads to highly flexible and reusable design solutions, but also provides traceability of requirements in design solutions making maintenance less tedious.

Keywords: *Requirements Engineering (RE), Software Architecture, Design Patterns, Architectural Patterns*

1. Introduction

Architecture gained importance in software development process as a powerful means of software abstraction. To a great extent, architecture is distanced away from the details of the system. Potential reuse in the form of interaction modeling is captured in patterns at varying levels of granularity. Patterns enable designers to capture these interactions as reusable artifacts in software design process. These interactions in turn provide a structure for the entire application. In other words, patterns deal with the architectural aspects of a software system. Commonly occurring

patterns in software systems have been categorized. Architectural pattern expresses a fundamental structural organization for the software system by providing a set of predefined subsystems and their responsibilities. It also includes rules and guidelines for organizing the relationships between these subsystems [8]. Object orientation facilitates reuse of classes within and across applications. Generalization and aggregation hierarchies enable this. Design patterns [7] are based on these principles. The fundamental structure of the entire software is not governed by design patterns. But they do influence the architecture of a subsystem.

Software development methodologies practiced today, fail to address the synergy between the requirement engineering process and architectural design. Traditional system development methodologies like waterfall model follow a sequential step. The requirements are captured first and only upon completion of this step, design and subsequent stages in the development process are addressed. Requirement elicitation mainly concentrates on the functional aspects of the system. Unless the collaborations among the entities directly contribute to the functional aspects, they are not adequately captured during this phase. We propose a development methodology wherein the systems structure in terms of the collaborations, is captured at the requirements phase itself by intuitively understanding the interactions among the participants and relating them with the previously known patterns. This gives a skeleton for the application's solution at a higher level, which can further be refined to lower level patterns.

Patterns are available at varying levels of granularity for the above mentioned approach [5, 7, 8]. Architectural patterns guide us in giving a structure for the software. Gang-of-Four (GoF) patterns address issues close to code. The same principles which form the basis of these patterns could as well be applied at an abstract level in the design process.

Choosing an appropriate structure for the application upfront, constrains and bounds the design space. Also, choice of a pattern conveys the semantics of the application. The characterization of application in terms of patterns do not stick to any formal definition of that pattern, but they do convey much more about the structure as well as computing model.

Creating an exhaustive set of patterns for the entire software domain is a never ending process. Also, it is not possible to have a complete pattern language to design a software system. In such cases, the design should be based on the relationship between the entities identified during the requirements phase. Depending on the problem domain involved, a hierarchy of patterns and a relationship between them could be figured out. As this process attains maturity, it could lead towards valuable design guidance in the form of a design handbook for the organization for specific domains and specific concerns [4] similar to the ones available in mature engineering disciplines.

The paper is organized as follows. Section 2 details the important activities in requirements engineering. Section 3 introduces the pattern oriented software development life cycle model. Section 4 explains software design as a pattern composition problem. The approach proposed in Section 3 is explained in Section 5 using a small case study. Section 6 provides a comparative account of related work. Section 7 concludes the paper with a discussion on a few ideas which includes outstanding issues for further work.

2. Requirements engineering

Requirements engineering deals with the early phases of software engineering namely requirements elicitation, modeling, specification and validation [6]. We could employ a variety of techniques for this such as interviewing, use-case modeling, essential prototyping, Class Responsibility Collaborator (CRC) modeling etc. Irrespective of the modeling techniques used, the basis of the activity remains same. Requirements analysis results in domain classes. Domain classes along with framework classes lead to class models.

Extracting information out of problem space itself may not be easy in some cases. Concepts in problem space may not necessarily be translated to concrete objects. This may be due to the fact that realization of requirements may require multiple classes. Ambiguity in problem space needs to be resolved before moving on to solutions.

Requirement specifications should not only aim at solution end from implementation point of view, but should also focus on the long life of designs. Such designs will be resilient to evolving requirements. RE is concerned with the services provided by and the constraints on a large and complex software system [9]. Apart from this, RE is also concerned with the relationship of these factors to precise

specifications of systems behavior and their evolution over time and across system families. Thus RE becomes a challenging activity which has effect on the forthcoming phases and the quality of the design. For an application which is intended to be used once, the traceability of requirements is important only during the maintenance phase. However, for the derivation of architectures like productlines, this activity is more crucial. Here RE encompasses activities like planning the baseline architecture, analyzing commonality-variability etc. A pattern oriented approach for the design of frameworks for software productlines is explained in [17].

3. Pattern oriented software development life cycle model

We propose a pattern oriented life cycle model for software development. Figure 1 gives an outline of this approach. The key idea here is to have a global structure for the application based on its overall computation and communication model, guided by the knowledge available in the form of pattern catalogs and pattern languages. An intuitive understanding of the application in terms of the global data flows would suffice for this step. This is an elegant approach since the global concerns of the application are addressed here and it is possible to apply this approach to the system at varying levels of granularity. Architectural patterns [8] can be used as fundamental design decisions for the software system, imposing a structural framework for the application during this step. For example, a system where information flows in a sequential fashion can be perceived as a pipe and filter architectural pattern. Database applications and network protocols could be structured as a layers pattern [8]. The structural framework thus perceived, in turn forms a context for subsequent analysis and realization of requirements.

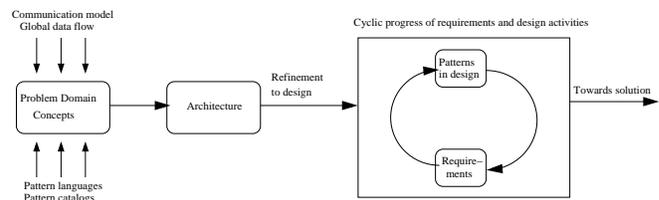


Figure 1. Pattern Oriented Life Cycle Model

Next step in the life cycle is the refinement of this architecture to design. During this phase, requirements could further be analyzed in detail, to identify lower level patterns in the systems and subsystems. By lower level patterns, we mean design patterns which could be product specific like J2EE patterns or general solutions like GoF patterns. Choice of product specific patterns again could be a

requirement driven factor. This is a cyclic activity during which, requirements as well as structure of the application get evolved simultaneously, each activity forming the context for the other.

3.1. Process improvement by patterns

The application of a well-managed, iterative and incremental development life-cycle has been pointed out as one of five characteristics of successful object-oriented projects [10]. Usually in system development process, the requirement models developed early in the development cycle undergo several working compromises during the development cycle. So it is natural that the initially perceived and documented models are not available when the development is complete. Pattern based requirement models solve this problem considerably because the basic design trade-offs encountered by software designers are well captured in the patterns chosen to fit in the design. Considerable variation from this structure is unlikely when the design elements are filled up in this structure. These models act as powerful communication mechanisms during design and redesign process.

Software design is primarily dictated by the context in which the design activity takes place, and is influenced by enabling techniques like modularization, encapsulation information hiding, separation of interface and implementation etc. Software patterns are solutions, which are based on these enabling techniques. Patterns address the issues in design to a great extent. Requirement models can rightly be transformed to design models by means of these patterns. The domain functionality could then be provided in the design. Since the induction of a pattern is for addressing a specific concern in the system, traceability of requirements in solutions becomes easy.

3.2. Requirements engineering from a new perspective

Requirements engineering should adequately address functional and non-functional requirements of the software. In fact, if functional requirements affect only that part of the software that determines them, they typically have localized effects. On the other hand, requirements which cut across various parts of the system, can be captured from the interactions among these parts. These interactions govern the structure of the system.

While analyzing the requirements in a system it is a good idea to classify the requirements. Certain requirements could be currently existing in the system. The analysis process could stretch itself to foresee certain requirements which the system is likely to accommodate in the future at the same time making provision for incorporating the

requirements. Certain requirements may necessitate potential changes in systems design. There may be some, which the system will never be able to handle. This categorization helps the systems designer to come up with an optimum architecture for the system. The designer could also make judgement about the capability of the system that has been designed based on this classification.

Architecture concerns with the structure and is like "load-bearing walls" [13] of the software. This means that within a particular architectural framework, it is possible for the application to undergo changes, without affecting this structure. The system functionality should be evolvable within this architecture. Pattern oriented approach that we suggest becomes meaningful in this context. Since there are infinite ways of realizing these design solutions in code, it will be possible to add or remove requirements which have localized effects in the future unless they are precluded in advance by the choice of a specific pattern.

3.3. Novel approach for requirements capturing

To ensure long life for designs, they should be adaptable. Software in general and OO systems in particular should be realized as an implementation of an abstraction. At the same time, these abstractions should have the ability to accommodate requirement changes. The modular decomposition of a system should be both open and closed [1]. The designs thus have a stable core on which the resulting applications can rely on, at the same time have open portions which can accommodate context dependent variations or requirement changes. Most of the design patterns address this issue.

Portions of an application that should be kept resilient to changes and extension are often referred to as hot spots [21]. Organization of an application around such hot spots determines how well it is closed for modifications at the same time open for adaptation. Knowledge about the hot spots in a design and how they are accessed by the client software is important for all phases of software development and maintenance, whether it is construction, comprehension or evolution. The "open-closed" principle and hot spot driven design should be conceived very early in the development life cycle; precisely at requirement capture stages itself. Pattern based models that we suggest essentially do this.

3.4. Patterns in requirements engineering

Any software development methodology has an underlying model supporting the development process. Models and abstractions constitute the basic framework for the development process in a domain. From the requirements point of view, architectural abstractions make trade-off analysis simpler, and provides a model that is easily refinable to code.

The model gets itself evolved as the development proceeds. For example, the domain processes at a coarse level could be expressed by using subsystem or components and their interactions. Further analysis could be aided by use cases for requirement modeling. Use cases lead to a conceptual model where concepts are realized using objects. Subsequently, the collaboration between objects are addressed. Subsystem interactions and relation between various use cases could be seen as requirement patterns, which can be documented. Once these requirement patterns are mapped to corresponding architectural or design patterns, the mapping could as well be used as a reusable artifact.

Interaction diagrams are one of the most important artifacts created during RE. Skillful assignment of responsibilities for the participants in the interaction diagrams is also important irrespective of the granularity of the participants, whether they be subsystems or objects. Proactive and prescriptive use of patterns assist the designer to a great extent in this step. Patterns can aid the RE process in two ways. They can result in single solutions. Secondly, they can aid in the development of reusable frameworks which are customizable design solutions. Patterns play an important role in customizing and designing application frameworks. This has been emphasized in [15].

It is generally observed that successful projects spend considerable amount of time and resources in the RE phase. It would be useful if, this phase can as well come up with requirements patterns and their corresponding mapping for related problems. Requirements patterns could be documented in the form of use cases, combination of use cases or sequence of occurrence of events.

3.5. Patterns as solution to non-functional requirements of software

In software design process, choice of an architecture is more of an activity of giving a structure to the whole application. The realization of the rest of the functionality of the software succeeds this step and ideally the software functionality should be evolvable within this architecture, without compromising its constraints.

Patterns mostly address nonfunctional requirements of software. By structuring a database application using a multiple layered pattern [8], we make the changes isolated. Problems such as lack of flexibility in most OO systems can be solved by reorganizing the design by making use of a strategy pattern [7]. The implication of this is that the design is addressing a non functional requirement of the system called flexibility. Another interesting point is that this reorganization will result in the degradation of system performance because the instance of one class needs to invoke an instance of a strategy class. Thus, pattern based requirement models since they abstract out details, serve the ideal

solutions for requirement models. Also, these models enable the point at which certain quality attributes are inhibited. As a result, selecting desired solutions from a set of alternate solutions becomes easier.

In order to make sure that a system is well structured and organized, in addition to exposing global structure of the system, design should obey certain basic design principles which are to be well documented. Well structured requirements and design decisions at several layers of abstraction are crucial for understanding a detailed specification document [20]. The pattern oriented software development method proposed, assists in systematic unfolding of requirements at varying levels of abstraction and provides sound design documentation.

4. Software design as pattern composition problem

Application of patterns in software development is to be seen as a pattern composition problem. Here we provide a design solution, rather than a programming solution that is tunable only at the implementation level. Understanding of how the abstractions in software are to be adapted, extended composed and maintained is equally important as providing knowledge about the locating of the key abstractions in it. Advantages of using patterns as building blocks of architecture and the related issues are explained in [12].

The combination of collective behavior of components need to be explored at the design and architectural levels. This issue is addressed by design patterns. If we use patterns as building blocks of architecture, not only that we address a specific functional aspect, but also the interaction among the various requirements. The objectives met by design elements could be addressed using the roles played by different objects in that pattern. Requirements modeling addresses dynamic nature of the requirements in this case.

When patterns combine to generate solution architectures, the structural and behavioral composition need to be addressed. Behavior composition addresses concerns like the roles played by various objects as elements in patterns. This kind of design is referred to as responsibility-driven design or interaction oriented design in OO literature [16]. Assignment of responsibilities to objects and design of object collaborations is very important. Neglecting the importance of the creation of interaction diagrams and responsibility assignment has been pointed out as a common problem in object technology projects [2]. We believe that requirements analysis emphasizing these steps and patterns as design solutions, will alleviate this problem to a large extent. When patterns are used as compositional units of an architecture, an elegant mechanism for addressing the collaborations among the pattern participants is discussed in [3].

5. Case study

This Section illustrates the methodology we have proposed using an example of component interactions in a feedback control system. The system uses feedback from the output to control a process like any feedback control system available in control literature. Feedback unit takes the output data from the process being controlled and then makes necessary adjustment to be fed to the feedforward unit after comparing the feedback value derived from the output, with a reference value. Then, feedforward unit sends the modified output to the controlled process. Considering the global data flow in the application, the adjustor reads feedback data and reference data, the controlled process reads the modified output and the feedback unit reads the output data from the controlled process.

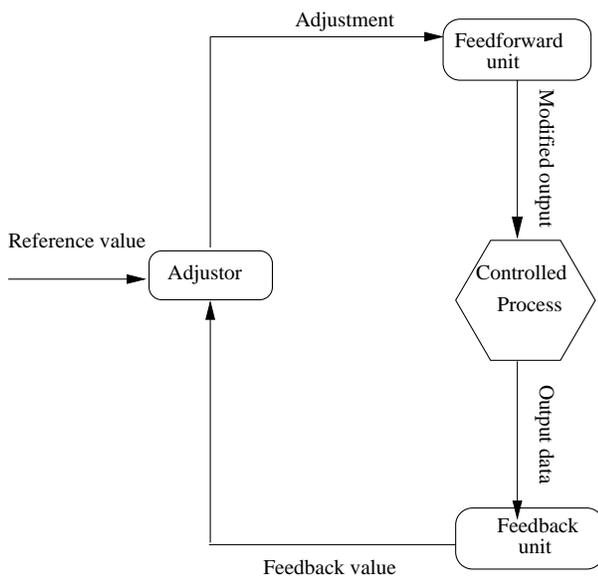


Figure 2. Component Interactions in a Feedback Control System

Even though the global data flows in the application, look like a pipe-and-filter architectural pattern [8], this architecture does not fit in here since pipe-and-filter does not allow any feedback loops. On the other hand, it fits into the blackboard pattern [8], in which several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

Figure 3 gives one possible design for this problem. Analyzing the requirements further, it can be seen that controlled process acts as a mediator between feedforward and feedback units thus the interactions among them resemble that of a mediator pattern [7]. Controlled process can also

be realized as a singleton pattern. Now, having identified these patterns, requirements could further be analyzed in detail to assign responsibilities to the classes in the patterns.

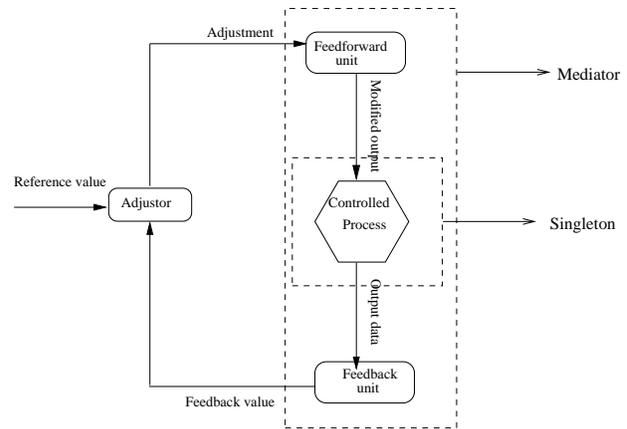


Figure 3. Design 1 for the Feedback Control system

It is natural that more than one design is possible for the same problem. In this context we give an alternative design for the same problem. This design is given in Figure 4. The functionality provided by the feedforward unit to the rest of the units should be the same, irrespective of the different control strategies used by it. A strategy pattern [7] could be used for this. The same interpretation holds with feedback unit also. To reduce the dependency between controlled process and the feedback unit, observer pattern can be used.

When alternate designs exist for the same problem, based on some trade-off analysis, the designer may have to choose the best design. In such situations, a methodology proposed in [4] aids the designer to compare the alternate designs in terms of some metrics like static adaptability, dynamic adaptability, extendibility etc. Details regarding this methodology is available in [4].

From the case study, it is evident that pattern oriented life cycle model allows the developer to systematically arrive at the design, by concentrating on the interactions existing in the domain. It is to be emphasized that trade-off analysis for alternate designs is also possible. The requirements are mapped to corresponding patterns as the design evolves. This makes traceability of requirements in solutions easy. Design solutions thus obtained are reusable since they are composed of patterns which are implementation independent, abstract entities.

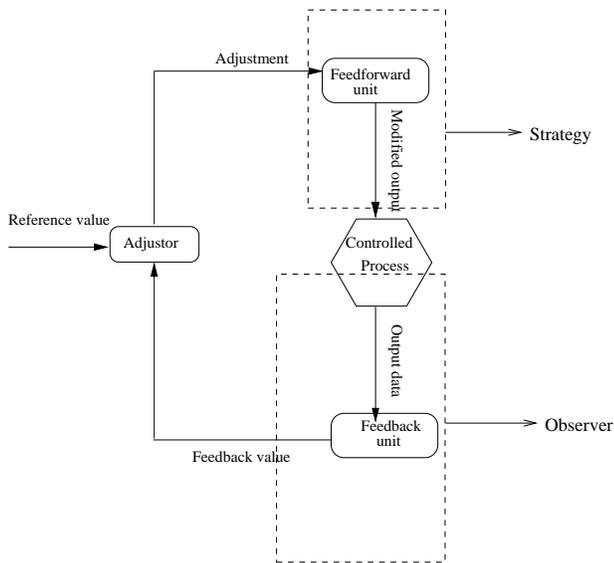


Figure 4. Design 2 for the feedback control System

6. Related work

Software engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software [11]. Steps available in software development life cycle models explained in [14] do not seem to be addressing all these aspects. Since patterns are identified as distilled experience of expert designers, pattern oriented development life cycle model turns out to be a systematic and disciplined approach.

[19] proposes a mechanism that utilizes UML modeling capabilities to compose design patterns at various levels of abstractions. The approach gives emphasis to the traceability of patterns in designs. A systematic approach which takes into account the global structure of the application and subsequently refining this structure to lower level patterns does not seem to be addressed by research community yet. Our approach also opens up issues that arise when patterns are used as fundamental building blocks of architecture, most important issue being the interactions among patterns.

Pattern composition is addressed by using role diagrams in [18]. The focus here is on deriving a composite pattern, which is a combination of individual patterns. This composite pattern solves a bigger problem in the sense that the synergy of participating patterns makes the composition more than its parts. However, a generalized application development using patterns is not addressed here.

Patterns solving independent problems are documented in [8, 5, 7]. These serve only as independent pattern documentations, explaining the context, forces and solution. Our approach is towards refining and combining these solutions to build reusable application solutions.

7. Conclusions and future work

We have proposed a life cycle model using pattern oriented approach for the development of software. The approach relies on the application of previously known solutions to design problems in the form of patterns. The structure of the application is perceived in the beginning and detailed requirements elicitation follows this step. As the patterns are refined to lower level patterns, requirements also get refined. Through this approach, RE, aids architectural design by mapping the constraints imposed by the requirements to known solutions and facilitates fast trade-off analysis. Architectural modeling is supported by not only the functional and nonfunctional requirements, but also the rationale behind the formation of the pattern. The methodology proposed enables requirements capture in the context of formal architectures. We believe that when complex systems are composed from pre-existing components, the contractual obligations of the participating components also need to be captured as requirements. These contracts may lead to composition patterns, as necessitated by composition context and semantics. As part of our future work, we plan to address these issues. We foresee this as an important problem worth addressing in the context of design reuse in the form of patterns and code reuse in the form of components.

References

- [1] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.
- [2] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design*. Prentice Hall, PTR, Upper Saddle River, New Jersey 07458, 1998.
- [3] D. Janaki Ram, Jithendra Kumar Reddy, M. S. Rajasree. An Approach to form a Pattern Oriented Design Graph Using PPC Model. In *Proceedings of the SoDA'02, International Workshop on Software Design and Architecture*. Bangalore, India, December 2002.
- [4] D. Janaki Ram, K. N. Anantharaman, K. N. Guruprasad, M. Sreekanth, S.V.G.K. Raju and A. Ananda Rao. An Approach for Pattern Oriented Software Development Based on a Design Handbook . *Annals of Software Engineering*, 10:329–358, October 2000.
- [5] D. Schdmit, M. Stal, H. Rohnert, F. Buschmann. *Pattern Oriented Software Architecture: A System of Patterns - Vol II* . John Wiley and Sons, 1999.
- [6] A. M. Davis. *Software Requirements: Analysis and Specification*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

- [7] E. Gamma, R.Helm, R.Johnson, J.Vlissides. *Design Patterns, Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.
- [8] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlab, M. Stal. *Pattern Oriented Software Architecture: A System of Patterns - Vol - I*. John Wiley and Sons, 1996.
- [9] A. Finkelstein. The future of Software Engineering: 2000. In *Proceedings of 22 nd International Conference on Software Engineering*, 2000, ACM Press.
- [10] Grady Booch. *Object Solution, Managing the Object Oriented Projects*. Addison Wesley, 1996.
- [11] IEEE. *IEEE Standards Collection: Software Engineering*. IEEE Standard 610.12-1990, IEEE 1993.
- [12] M. S. Rajasree, D. Janaki Ram, P. Jithendra Kumar Reddy. Composing Architectures from patterns. In *Proceedings of the SoDA'02, International Workshop on Software Design and Architecture*. Bangalore, India, December 2002.
- [13] Perry, D. E., A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT*, 1992. 17(4).
- [14] R. S. Pressman. *Software Engineering A Practitioner's Approach*. Tata McGraw-Hill Companies, Inc., 1997.
- [15] R. Johnson, B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.
- [16] R. Wirfs-Brock B. Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'89.*, pages 71–75, 1989.
- [17] Rajasree M S, D Janaki Ram, Jithendra Kumar Reddy. Systematic Approach for Design of Framework for Software Productlines. In *Proceedings of the PLEES'02, International Workshop on Product Line Engineering: The Early Steps Planning Modeling and Managing*, In Association with OOPSLA 2002. Fraunhofer IESE, October 28, 2002.
- [18] Riehle, D. Composite Design Patterns. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'97, Atlanta, Georgia, USA*, pages 218–228, October 1997.
- [19] Sherif M. Yacoub, Hany H. Ammar. UML Support for Designing Software as a Composition of Design Patterns. In *UML 2001 - The Unified Modeling Language - Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 2001.
- [20] M. Weber and J. Weisbrod. Requirements Engineering in Automotive Development: Experiences and Challenges. *IEEE Software*, pages 16–24, January 2003.
- [21] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.

Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery

Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA
nenom@usc.edu

Alexander Egyed

Teknowledge Corporation
4640 Admiralty Way
Marina Del Rey, CA 90292, USA
aegyed@acm.org

Paul Gruenbacher

Sys. Eng. & Automation
Johannes Kepler University
4040 Linz, Austria
gruenbacher@acm.org

ABSTRACT

Ideally, a software project commences with *requirements gathering and specification*, reaches its major milestone with system *implementation and delivery*, and then continues, possibly indefinitely, into an *operation and maintenance* phase. The software system's *architecture* is in many ways the linchpin of this process: it is supposed to be an effective reification of the system's requirements and to be faithfully reflected in the system's implementation. Furthermore, the architecture is meant to guide system evolution, while also being updated in the process. However, in reality developers frequently deviate from the architecture, causing *architectural erosion*, a phenomenon in which the initial architecture of an application is (arbitrarily) modified to the point where its key properties no longer hold. In this paper, we present an approach intended to address the problem of architectural erosion by combining three complementary activities. Our approach assumes that a given system's *requirements* and *implementation* are available, while the architecturally-relevant information either does not exist, is incomplete, or is unreliable. We combine techniques for *architectural discovery* from system requirements and *architectural recovery* from system implementations; we then leverage architectural styles to identify and *reconcile* any mismatches between the discovered and recovered architectural models. While promising, the approach presented in the paper is a work in progress and we discuss a number of remaining research challenges.

1 INTRODUCTION

Ideally, software systems are developed via a progression starting from *requirements* through *architecture* to *implementation*, regardless of the lifecycle model employed. Any changes to those systems during their, possibly indefinite, lifespans should then follow the same progression: a change in the requirements is reified in the architecture and, subsequently, the implementation. However, frequently neither the initial development process nor the system's evolution and maintenance follow such a path for reasons that include developer sloppiness; requirements that are immediately implemented due to (the perception of) short deadlines; architectural decisions that are violated to achieve non-functional qualities (e.g., improve performance, satisfy real-time constraints, reduce application memory footprint); off-the-shelf (OTS) functionality that is directly incorporated into the system's implementation; and the existence of legacy code that is perceived to prevent careful system architecting.

For these reasons, architectural artifacts are often out of sync with the system's requirements and its implementation, and we say that the architecture is *eroded* [27]. There are many

potential problems associated with architectural erosion: difficulties in assessing how well the current implementation satisfies the current requirements; inability to trace a specific requirement to implementation artifacts; lack of understanding the complex effects of changing a requirement; and inadequate system maintainability and evolvability. The incorrect perception of the architecture may lead to incorrect architecture-level and, subsequently, implementation-level decisions in response to new or changing requirements.

To deal with the problem of architectural erosion, researchers and practitioners have typically engaged in *architectural recovery* [2,10,14,15,18,22,28,31,32], where the system's architecture is extracted from its source code. However, existing architectural recovery approaches fail to account for several pertinent issues. They rely primarily on implementation information, leveraging requirements in a limited fashion, if at all. Since the implementation may have violated certain system requirements, they will, in effect, recover incorrect architectures in such cases. In addition, architecturally-relevant decisions are frequently obscured by the implementation. This may be the result of justified implementation-level decisions, such as eliminating processing bottlenecks, removing duplicate modules for efficiency, OTS reuse, and so on. Architectural decisions might also be ignored without justification, due to a missing system-wide view, developer sloppiness, misguided "creativity" in implementing the desired functionality, and so on. Another problem with existing approaches to architectural recovery is their relative heavy weight, a by-product of the lack of reliance on information already present in the system's requirements. Perhaps most importantly, the existing architectural recovery approaches exhibit no understanding of the importance and role of *architectural styles* in developing large-scale, complex software systems. An architectural style is a key design idiom that implicitly captures a large number of design decisions, the rationale behind them, effective compositions of architectural elements, and system qualities that will likely result from the style's use [8,22,29]. Without this knowledge, a system's architecture will present only a partial picture regardless of how faithfully its structural, compositional, behavioral, and/or interaction details are recovered.

Our research goal is to combine software requirements, implementations, and architectural styles in a light-weight and scalable manner to stem architectural erosion. Requirements serve as the basis for discovering a software system's architecture. Implementations serve as the basis for recovering the system's architecture. Because of their different inputs, discovery and recovery are likely to reveal different and possibly incomplete architectural models. Architectural styles can be used to

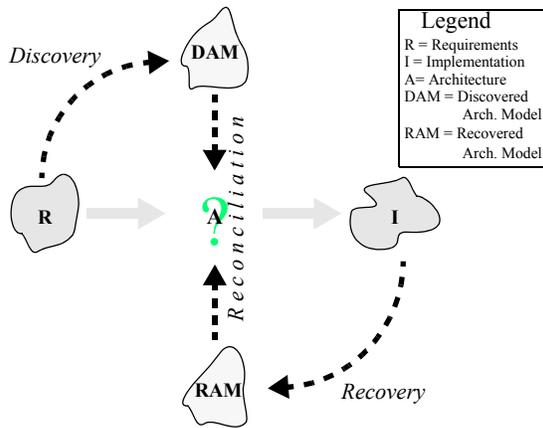


Figure 1. Conceptual view of the approach.

reconcile the two models and combine them into a coherent and more complete model of the software system’s architecture. Our approach therefore consists of three interrelated activities as depicted in Figure 1:

1. a technique supporting the *discovery* of an architecture from system requirements;
2. a technique for *recovering* an architecture from system implementations; and
3. an architectural style characterization technique to identify and *reconcile* any mismatches between the discovered and recovered architectural models.

We assume that the existing information about an architecture either does not exist or is unreliable. We also assume that the system’s requirements are known and that an inspectable implementation exists. We acknowledge that many modern software systems depend heavily on off-the-shelf libraries (e.g., GUI libraries) or middleware platforms (e.g., CORBA, DCOM). However, deriving architectural properties from such technologies is a challenging task and is thus outside the current scope of our work.

2 BACKGROUND

This work builds on three related areas: software architectures and architectural styles; software requirements, and specifically approaches for mapping requirements to architectural decisions; and architectural recovery.

2.1 Software Architectures and Styles

Software architecture is a level of design that “involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns” [29]. A goal of software architectures is to facilitate development of large-scale systems, preferably by integrating pre-existing building blocks of varying granularity, typically specified by different designers, implemented by different developers (possibly in different programming languages), with varying operating system requirements, and supporting different interaction protocols.

An *architectural style* [8,16,29] is a set of design rules

that identify the kinds of building blocks that may be used to compose a system, together with the local or global constraints on the way the composition is done [29]. Styles codify the best design practices and successful system organizations [1,20]. Several architectural styles have been in use for a number of years, including client-server, pipe and filter, blackboard [29], C2 [30], and REST [9].

2.2 Architectural Discovery

Software requirements describe aspects of the problem to be solved and constraints on the solution. Requirements deal with stakeholder goals, options, agreements, issues, and conditions to capture the desired system features and properties. Requirements may be simple or complex, precise or ambiguous, stated concisely or elaborated carefully. Although informal requirements described in natural language often lead to ambiguities and inconsistency, they are frequently used in practice and are thus of special interest in our research.

The relationship between the requirements and architecture for a desired system is not readily obvious. Several existing techniques provide suggestions for addressing the problem. For example, the QUASAR approach [4] relates desired system features (e.g., “The system must be secure.”) to solution fragments that effect those features (e.g., “Employ an encryption scheme.”). The objective of QUASAR is to allow reuse and compose solution fragments across systems with similar desired features. However, this work has only recently begun addressing the relationship of desired features and software architectures. ATAM [17], a technique that supports the evaluation of architectural decision alternatives in light of non-functional requirements, has a similar limitation. Twin Peaks [25] attempts to overcome the separation of requirements specification and design activities by intertwining them. However, unlike our approach, Twin Peaks does not take into account the implementation. Brandozzi and Perry [3] have recently coined the term “architecture prescription language” for their extension of the KAOS goal specification language [19] to include architectural dimensions. Their approach has the same limitations as our architectural discovery technique: they are unable to suggest a complete architectural configuration based on the information extracted from the requirements, and they currently make no use of non-functional requirements in modeling the discovered architecture. This is why we have decided to couple architectural discovery, recovery, and styles.

Finally, a key issue in transforming requirements into architecture and other software models is traceability. Researchers have recognized the difficulties in capturing development decisions across software models [11]. In response to this, Gotel and Finkelstein [12] suggest a formal approach for ensuring the traceability of requirements during development.

2.3 Architectural Recovery

A number of existing approaches focus on recovering a software architecture from source code. ARM [14] is an

approach to architectural reconstruction distinguishing between the conceptual architecture and the actual architecture derived from source code. ARM applies design patterns and pattern recognition to compare the two architectures. Unlike our architectural recovery approach, ARM assumes the availability of system designers to formulate the conceptual architecture. Similarly to our recovery approach, software reflexion models [24] treat a system's architecture from two perspectives: the idealized, high-level view and the low-level view derived from source code. Reflexion models support incremental architectural recovery to analyze whether varying sets of relationships hold between the idealized and actual architectures. However, reflexion models do not make direct use of architectural concepts such as styles and connectors.

MORALE [28] is an approach for evolving legacy software systems developed with procedural languages. COREM [10] is an approach that converts procedural into object-oriented systems via four steps: design recovery, application modeling, object mapping, and source code adaptation. Neither of these approaches provides a means for determining whether the implemented systems completely and correctly satisfy their original requirements, or whether the requirements themselves are complete and consistent.

Recently, a series of studies has been undertaken to recover the architectures of several open-source applications [2,15]. The approach taken in these studies has been to come up with a conceptual architecture from a system's documentation and use it as the basis for understanding the system's implementation. The system documentation is assumed to be correct when, in fact, both the documentation (e.g., requirements) and implementation may be partially incorrect, incomplete, or internally inconsistent. As with all of the above approaches, architectural style information is not leveraged during recovery.

3 EXAMPLE APPLICATION

To illustrate the discussed concepts we use ShareDraw, an application implemented in Visual C++. ShareDraw is an extension to the DrawCli application, which is provided as part of the Microsoft Foundation Classes (MFC) release. DrawCli allows users to manipulate 2-D graphical objects (lines, ovals, polygons). ShareDraw extends DrawCli into a

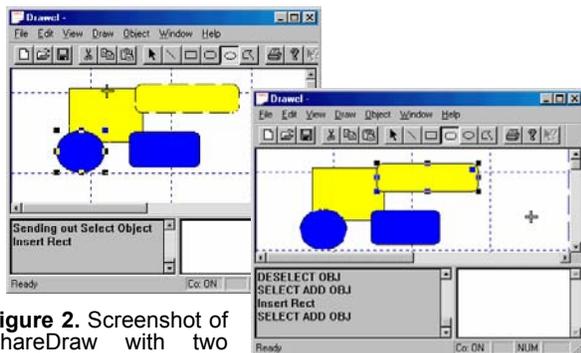


Figure 2. Screenshot of ShareDraw with two clients shown.

distributed application that adds collaborative drawing and chatting facilities, as depicted in Figure 2.

The architecture of ShareDraw was not available to us. Similarly, DrawCli's requirements were not available. However, given the highly interactive nature of the application, we can easily extract many of the functional requirements from the application's observed behavior. The requirements for the extension of DrawCli into ShareDraw were available. Several informally stated requirements describing some commonly performed operations are as follows:

Req₁: ShareDraw should allow the user to save drawings for later retrieval.

Req₂: ShareDraw should make object manipulation operations easily accessible to the user.

Req₃: ShareDraw should allow the user to group and simultaneously manipulate multiple drawing objects.

Req₄: ShareDraw should allow the user to instantly view the actions of all other users.

4 THE APPROACH

The goal of our research is to develop a generally applicable, style-centered approach for integrating architectural discovery and recovery techniques, and reconciling the identified differences. Our approach will comprise three separate, but complementary techniques, as depicted in Figure 1:

1. an architectural style-based technique for architectural discovery from software requirements,
2. an architectural style-based technique for architectural recovery from software implementations, and
3. a technique that leverages styles to reconcile the results of discovery and recovery.

4.1 Architectural Discovery

Elaborating system requirements into a viable software architecture satisfying those requirements is often based on intuition [25]. Software engineers face some critical challenges in performing this task [13]:

- Requirements are frequently captured informally in a natural language, while software architectures are usually specified formally [21].
- Non-functional system requirements are hard to capture in an architectural model [21].
- Mapping requirements into architectures and maintaining their inter-consistency is complicated since a single requirement may address multiple architectural concerns and vice versa.
- Large-scale systems have to satisfy hundreds, possibly thousands of requirements, making it difficult to identify and refine the architecturally relevant information contained in the requirements.

To address these challenges we developed CBSP [13], a light-weight technique to distill from the system requirements the key architectural elements and the dependencies among them. The result of the technique is an intermediate model between the requirements and architecture that con-

tains the essence of architectural information embedded in the requirements. This model is referred to as the discovered architectural model, or DAM. The CBSP approach creates DAM in a structured process using conflict resolution to address ambiguities in the requirements. The process consists of three main activities, detailed in [13]:

1. classify architecturally relevant requirements,
2. identify and resolve classification inconsistencies, and
3. refine/restate architecturally relevant requirements.

In this section we detail the DAM model itself.

4.1.1 Discovered Architectural Model

The basic idea behind our approach to architectural discovery is that any software requirement may explicitly or implicitly contain information relevant to the software system's architecture. It is frequently very hard to surface this information, as different stakeholders will perceive the same requirement in very different ways. CBSP captures this information in the intermediate DAM model. DAM is structured around a simple set of general architectural concerns derived from existing software architecture research [21,27,29]:

- *Components* provide application-specific functionality. They may be *data* or *processing* components [27].
- *Connectors* facilitate and govern all interactions among the components.
- *Configuration* of a system or a particular subsystem describes the relationships and organization among multiple (possibly all) components in the system.
- *Properties* describe the non-functional characteristics of individual components and connectors, or the entire configuration.

Thus, each derived DAM element explicates an architectural concern and represents an early architectural decision for the system. For example, a requirement such as

Req_t: The system should provide an interface to a Web browser.

can be recast into a DAM processing component element and a DAM connector element

Comp_p: A Web browser should be used as a component in the system.

Conn: A connector should be provided to ensure interoperability with third-party components.

Because of the complexity of the relationship between requirements and architecture, DAM gives a software architect leeway in selecting the most appropriate *refinement* or, at times, *generalization* of one or more requirements. Examples of both refinement and generalization are given below.

There are seven possible DAM dimensions discussed below and illustrated with simple examples from the Share-Draw application. The seven dimensions involve the basic architectural constructs and, at the same time, reflect the simplicity of our approach.

(1-2) *Comp_p* and *Comp_d* are model elements that describe or involve an individual processing or data component in an architecture, respectively. For example

Req_t: The system should allow the user to directly manipulate graphical objects.

may be refined into DAM elements describing both processing components and data components

Comp_p: Graphical object manipulation component.

Comp_d: Data for abstract depiction of graphical object.

(3) *Conn* are model elements that describe or imply a connector. For example

Req_t: Manipulated graphical objects must be stored on the file system.

may be refined into

Conn: Connector enabling interaction between UI and file system components.

(4) *Conf* are model elements that describe system-wide features or features pertinent to a large subset of the system's components and connectors. For example

Req_t: Allow independent customization of application look-and-feel and graphical object manipulation tools.

may be refined into

Conf: Strict separation of graphical object manipulation, visualization, and storage components.

(5) *Prop_{Comp}* are model elements that describe or imply data or processing component properties, such as reliability, portability, incrementality, scalability, adaptability, and evolvability. For example

Req_t: The user should be able to view the effects of his actions with minimal perceived latency.

may be refined into

Prop_{Comp}: Graphical object manipulation component should be efficient, supporting incremental updates.

(6) *Prop_{Conn}* are model elements that describe or imply connector properties. For example

Req_t: The system should support loading of graphical manipulation tools at runtime.

may be refined into

Prop_{Conn}: Robust connectors should be provided to facilitate runtime component addition and removal.

(7) *Prop_{Conf}* are model elements that describe or imply system (or subsystem) properties. For example

Req_t: The system must support collaborative editing of graphical objects.

may be transformed into

Prop_{Conf}: The system should be distributable.

Note that, e.g., the *Prop_{Conn}* example (5) involved refining a general requirement into a more specific DAM element. On the other hand, the *Prop_{Conf}* example (6) involved the generalization of a specific requirement into a more general DAM artifact. In fact, in both cases multiple DAM artifacts may be produced as part of a single requirement. We are currently studying this issue with the goal of providing practical guidelines to architects engaging in this task.

4.1.2 Summary and Open Issues

At this point, we have an intermediate model, DAM. DAM classifies the key architectural concerns into seven categories: data components, processing components, con-

nectors, configurations, component properties, connector properties, and (sub)system properties. DAM is still stated in a requirements-like notation, such that it can be verified against the intentions of the system’s non-architect stakeholders (e.g., customers). DAM reinterprets architecturally relevant requirements; no requirements are actually changed aside from clarifications that arise during the discovery process. Finally, DAM classifies and describes the system’s architecturally relevant information in a way that makes it much easier to derive an architecture, and, subsequently, implementation from it would be from “raw” requirements.

However, a remaining problem is that the DAM elements provide a very low-level view of the architecturally relevant system requirements (recall the above examples). It may not be straightforward to map some aspects of DAM (e.g., configuration information, properties) into an effective architecture that will realize them. For example, in our experience architectural discovery is often unable to infer all interdependencies between architectural elements. This directly motivates the need to introduce additional information into the picture, as further discussed below.

4.2 Architectural Recovery

Architectural recovery complements architectural discovery by highlighting the major structural characteristics of the implemented system: data and processing components, connectors, and configuration. The result of architectural recovery is a recovered architectural model, or RAM. In this section we discuss the process of generating RAM. Later we will show how this information can be coupled with DAM to arrive at a more complete architectural model. We use UML to represent the recovered architecture.

4.2.1 Recovered Architectural Model

Our proposed architectural recovery technique will consist of the following four simple activities.

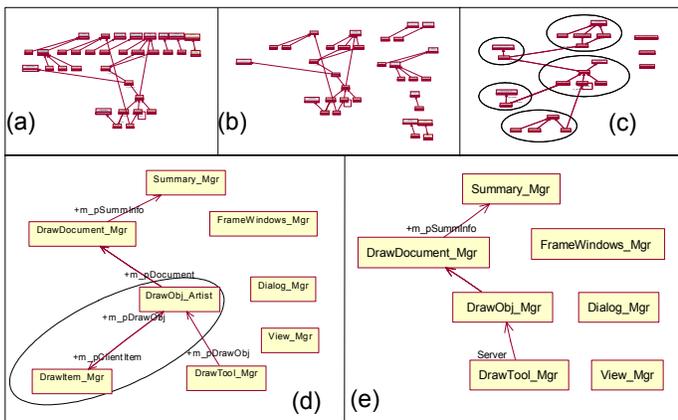


Figure 3. Identifying components from a UML class diagram. At this magnification, the top three diagrams are shown only for illustration, to convey the scope of the task.

Generate class diagrams. Numerous tools are available to infer class diagrams from source code automatically; the engineer need not even look at the system’s source code to accomplish this step. Figure 3a shows the class diagram of ShareDraw’s client subsystem, automatically generated by Rational Rose®.

Group related classes. Typically, a large number of implementation classes are required to implement individual architectural components and connectors. Classes can be grouped based on different criteria and/or architectural concerns. Multiple architects may participate in this process and, consequently, disagreements and mismatches may arise. The diagrams in Figure 3b-e show one possible such grouping of ShareDraw’s classes, obtained by applying the three simple rules adopted from our Focus technique [6]:

- Classes isolated from the rest of the diagram comprise one grouping (Figure 3b).
- Classes that are related by generalization (i.e., inheritance) comprise additional groupings, as do classes related by aggregation and composition (Figure 3c).
- Finally, classes with two-way associations are grouped together since they denote tight coupling (Figure 3d).

Package groups of classes into architectural elements.

Clusters of classes identified in the previous stage are packaged together into processing components, connectors, or their relationships (links). These elements can be further aggregated into even larger elements. Using this process, ShareDraw’s client implementation is abstracted into seven components and three inter-component links (Figure 3e), as well as two remote procedure call (RPC) connectors. The connectors are not shown in Figure 3 since UML and Rational Rose® provide no mechanisms for distinguishing connectors from components. Figure 3 also does not show data components as introduced by Perry and Wolf [27] and discussed in Section 4.1. Data components may be extracted from the processing components’ states and interfaces based on varying desired criteria (e.g., all class variables, all public method parameters, or both).

Determine partial system configuration. The relationships among the components identified in the preceding steps reflect the system’s configuration. The configuration information may be incomplete in cases where the components do not interact in easily detectable ways (e.g., access to shared implementation substrate classes, implicit invocation, distributed interaction, and so on). Figure 3e shows only a partial configuration of ShareDraw’s client: the topological relationship of the *FrameWindows_Mgr*, *Dialog_Mgr*, and *View_Mgr* components with the remaining components has not been identified in this process; in addition, the diagram does not identify the connectors for reasons discussed above.

4.2.2 Summary and Open Issues

The described architectural recovery technique is very simple and scalable, relying only on structural manipula-

tion of the system’s implementation. The outcome of the technique is the RAM model, i.e., the collection of existing system’s major processing and data components, its connectors, as well as a partial architectural configuration. RAM is intended to map to the structural aspects of DAM proposed in Section 4.1.

The result of the recovery step is not a complete “as is” architecture of the system. Several pieces of information are still missing. As discussed above, the architectural configuration information will likely be incomplete. In addition, similarly to a great majority of the existing recovery techniques (e.g., [2,10,14,15,18,28,31,32]), our proposed approach does not take into account non-functional properties. This shortcoming suggests the next step of our approach: by coupling the information represented in RAM and DAM with architectural style information, we can mitigate this problem and present a more complete picture of the architecture, as discussed below.

4.3 Reconciling Discovery and Recovery

The above two techniques provide related, though disconnected models of the system’s architecture, as depicted in Figure 1. The requirements are refined and rephrased into DAM elements along the seven dimensions representing the key architectural concerns: data and processing components, connectors, configurations, component properties, connector properties, and (sub)system properties. The implementation is abstracted into four types of RAM elements: data and processing components, connectors, and (partial) configurations. This section discusses how the two

models can be “matched up” to derive a more complete architecture based on their combined information.

4.3.1 Determining Appropriate Architectural Styles

As discussed earlier, architectural styles [8,16,29] provide rules that exploit recurring structural and interaction patterns across a class of applications. Styles constrain architectural models syntactically and semantically. In order to select the appropriate style(s) for the given application, we propose to classify existing architectural styles across a set of commonly recurring dimensions. Our goal is to provide the foundation of a classification that is rich enough to allow us to effectively represent and select styles based on the given DAM and RAM models.

Our preliminary study of architectural styles [22] has identified the following seven dimensions as a good candidate set for effectively describing styles.

1. the types of *data* exchanged between style elements;
2. the *structure* of the elements allowed in a style;
3. the allowed *topologies* of architectural elements;
4. the allowed *behavior* of a style element;
5. the types of supported *interactions* between style elements and their allowed specializations;
6. the key *non-functional properties* especially enabled by the style; and.
7. the style’s *domain* of applicability.

Table 1 depicts the result of an exercise in which we mapped four commonly occurring styles using this framework. This experience has indicated several challenges that

Table 1: Characterization of Four Architectural Styles

	Data	Structure	Topology	Behavior	Interaction	Properties	Domain
C2	Discrete events	Separable components	Limited component dependencies	Exposed via named services only	Asynchronous coordination	Distributability	GUI Systems
	Data tuples	Explicit connectors	Partially ordered connectivity-based “top” and “bottom” relations	Data queueing and buffering by connectors	Implicit invocation	Heterogeneity	
				Multi-tasking mechanisms such as threads	Event-based interaction	Composability	
			Dynamic creation of connections	Direction-oriented events propagated to topology-based recipients	Dynamicity		
Client-server	Parameterized request	Independent servers	Many-to-many connections among clients and servers	Listening server	Server location	Distributability	Distributed Systems
		Specialized clients		Connections setup and teardown	Remote connection and communication protocol		
	Typed response	Distributed protocol stacks	Dynamic creation of connections	Buffering and queueing of requests	Implicit server invocation	Security	
				Multi-tasking mechanisms such as threads	Data marshalling and unmarshalling	Evolvability	
			Exposed via named services only	Client call synchronization	Heterogeneity		
Pipe-and-filter	Streams of typed records	Explicit pipes and filters	Stream between a pipe and a filter	Stream transformation state machine	Synchronization between filter reads and writes	Heterogeneity	Dataflow systems
		Input and output ports on filters	No two sources or sinks connected to the same port instance		Data buffering by pipes	Propagation of stream contents to sinks	
		Sources and sinks on pipes					
Push-based	Channel notification	Independent producers	Producers connected only to distributors	Content filtering in distributors	Distributor location	Distributability	Distributed systems
		Explicit distributors		Buffering and queueing by distributors	Remote connection and communication protocol	Scalability	
	Subscription request	Channel access/subscribers	Many-to-many channels among receivers and distributors	Subscription setup	Data marshalling and unmarshalling	Robustness	
		Receiver user interface		Content storage/expiration	Distribution policy	Security	
				Implicit invocation			

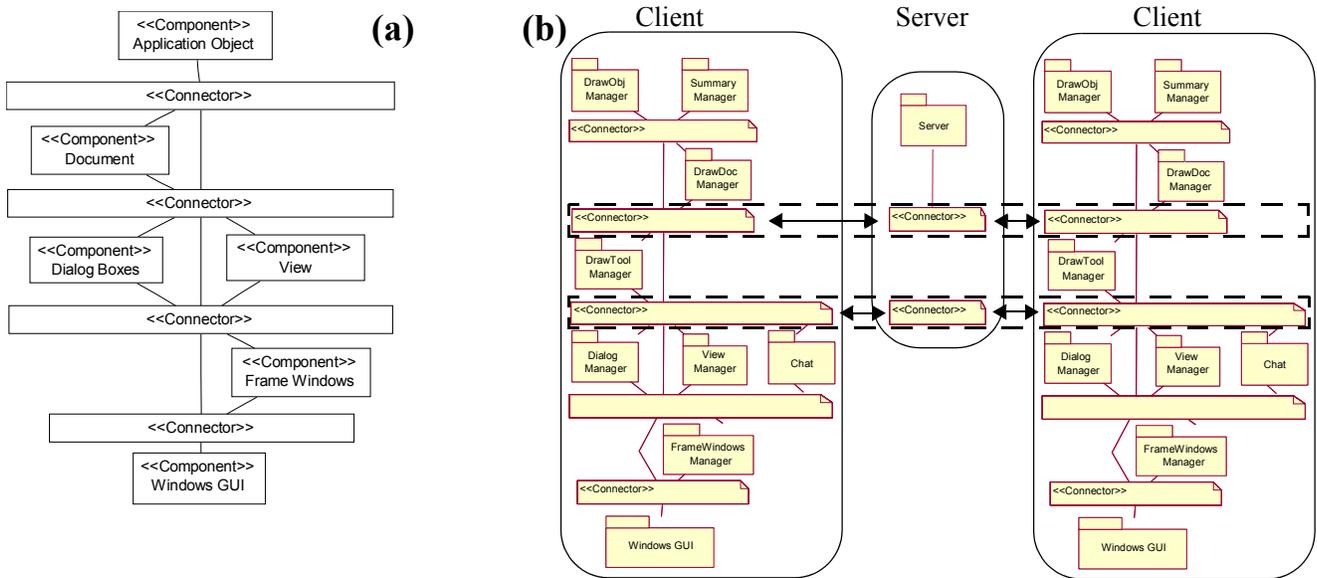


Figure 4. (a) “As intended” architecture of the ShareDraw client subsystem. A number of details have been elided for brevity. (b) Final architecture of the ShareDraw application, shown with two client subsystems (each roughly equivalent to the Draw-Cli application). The two highlighted connectors use RPC to communicate across processes.

we will need to address. First, we will need to carefully specify a large, representative if not complete, set of existing architectural styles. This process will help us test our hypothesis that the seven dimensions are sufficient to uniquely and richly describe a style. Second, we will need to characterize each style in a manner that will simplify the task of relating the information contained in DAM and RAM models to the information contained in Table 1. Third, we will need to address situations in which multiple styles are highlighted in this process as plausible candidates. A related issue is dealing with situations in which multiple styles are most appropriate to use *in tandem* for a given problem.

In fact, we indeed selected two styles in our ShareDraw example application: client-server to handle the distributed, coarse-grained aspects of the application, and C2 for its ability to compose the GUI-intensive application components within each client and server. This choice was aided by several factors, including our familiarity with these two styles, the fact that we had used them together in the past, the relatively small number of styles we had considered (e.g., Table 1 only includes four styles), and some domain properties (distribution and GUI aspects) that clearly mapped to these two styles. We envision this to be a much greater challenge in a more general setting. Our future work will include identifying conditions and situations under which specific combinations of styles are (dis)allowed. This is a non-trivial problem that deserves particular attention.

4.3.2 Integrating DAM and RAM

Once we have determined the suitable architectural style(s), we can integrate the, still separate, DAM and RAM models into a single integrated model. There are three possible approaches to accomplishing this step:

1. Apply the style information to DAM to derive an “as intended” architecture, and then “map” the information from RAM onto this architecture.
2. Apply the style information to RAM to derive an “as implemented” architecture, and then “map” the information from DAM onto this architecture.
3. Integrate DAM and RAM into an “as extracted” architecture, and then apply the style information to the integrated model.

We are currently investigating the respective benefits and drawbacks of the three approaches.

The “as intended” architecture of the ShareDraw application, obtained by integrating DAM and architectural style information as discussed above, is given in Figure 4a. The complete architecture, obtained by mapping the information contained in RAM to the “as intended” architecture, is shown in Figure 4b. As mentioned above, the final ShareDraw architecture combines the client-server and C2 styles.

Irrespective of the chosen integration approach, integrating DAM and RAM requires knowledge about how their elements interrelate. Although both DAM and RAM present architectural perspectives, they may be inconsistent, e.g., in the element names or level of architectural detail. The two models will thus need to be reconciled. Various interesting reconciliation scenarios can be envisioned. For example, a single RAM element may map to multiple DAM elements, and vice versa. It is also possible that no obvious relationship can be established between an element in one of the models and the other model’s elements. We will carefully study these scenarios.

5 CONCLUSION

This work described in this paper is motivated by the observation that architecturally-relevant information is

readily available in a system's requirements and its implementation, although not always in an obvious form. This information can then be uncovered and used to help stem architectural erosion. The information captured in a system's requirements is high-level, possibly imprecise, but rich in human stakeholders' insights and rationale; this information often *suggests* the suitable architectural style(s) for the system. On the other hand, the information contained in a system's implementation is low-level, precise, and rich in detail; this information *reflects* the style(s) applied in the system's construction. We postulate that neither of the two sources of information should be considered complete or correct by itself. Instead, we propose that they be combined using the three presented techniques: architectural discovery, recovery, and reconciliation.

The work described in this paper is on-going. We have already identified several open issues that will frame our future research. In addition, we envision that the combination of the three techniques will likely result in a self-adjusting process in which the architecture is already known to be incorrect and/or incomplete, but, in addition, neither the requirements nor the implementation need be assumed correct or complete. Furthermore, the proposed approach will result in clearly specified and maintainable traceability links across the requirements, architecture, and implementation. We plan to adopt existing techniques (e.g., [7,26]) to capture and manage the traceability links.

6 REFERENCES

- [1] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, October 1992.
- [2] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. *ICSE'99*, Los Angeles, CA, May 1999.
- [3] M. Brandozzi and D. E. Perry. Transforming Goal-Oriented Requirement Specifications into Architecture Prescriptions. *ICSE 2001 STRAW Workshop*, Toronto, May 2001.
- [4] H. de Bruin, H. van Vliet, and Z. Baida. Documenting and Analyzing a Context-Sensitive Design Space. *WICSA-3*, August 2002, Montreal.
- [5] A. Dardenne, S. Fickas, A. van Lamsweerde. Goal-Directed Concept Acquisition in Requirement Elicitation. *6th Int. Workshop on Software Specification and Design*, Oct. 1993.
- [6] L. Ding and N. Medvidovic. Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution. *WICSA-2*, Amsterdam, August 2001.
- [7] A. Egyed. A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering*. Accepted, to appear.
- [8] R. Fielding. Architectural Styles and the Design of Network-Based Software Architectures. *Ph.D Thesis*, UC Irvine, 2000.
- [9] R. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *ICSE 2000*, Limerick, June 2000.
- [10] H. Gall, R. Klosch, and R. Mittermeir. Object-Oriented Re-Architecting. *ESEC-5*, Berlin, Sep. 1995.
- [11] L. R. Gieszl. Traceability for Integration. *2nd International Conference on Systems Integration*, pp. 220-228, 1992.
- [12] O. C. Z. Gotel and A. C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. *1st International Conference on Rqts. Eng.*, pp. 94-101, 1994.
- [13] P. Gruenbacher, A. Egyed, and N. Medvidovic. Reconciling Software Requirements and Architectures: The CBSP Approach. *5th IEEE International Symposium on Requirements Engineering (RE'01)*, Toronto, August 2001.
- [14] G. Y. Guo, J. M. Atlee, and R. Kazman. A Software Architecture Reconstruction Method. *WICSA-1*, San Antonio, Feb. 1999.
- [15] A. E. Hassan and R. C. Holt. A Reference Architecture for Web Servers. In *Working Conference on Reverse Engineering*, Brisbane, Australia, Nov. 2000.
- [16] M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. *ESEC/FSE '99*, Toulouse, Sep. 1999.
- [17] R. Kazman, et al. Experience with Performing Architecture Tradeoff Analysis. *ICSE '99*.
- [18] R. Kazman and J. Carriere. View Extraction and View Fusion in Architectural Understanding. In *5th International Conference on Software Reuse*, Canada, June 1998.
- [19] A. van Lamsweerde, R. Darimont, E. Letier. Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering*, vol. 24, No. 11, Nov 1998.
- [20] A. MacDonald and D. Carrington. Guiding Object-Oriented Design. *TOOLS'98*, Melbourne, Nov. 1998.
- [21] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70-93 (January 2000).
- [22] M. Mikic-Rakic, N. R. Mehta, and N. Medvidovic. Architectural Style Requirements for Self-Healing Systems. *1st Workshop on Self-Healing Systems*, Charleston, Nov. 2002.
- [23] G. Mullery. CORE: A Method for Controlled Requirements Specification. *ICSE4*, Munich, Germany, Sept. 1979.
- [24] G. Murphy, D. Notkin and K. J. Sullivan, Software Reflection Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, vol. 27, no. 4, April 2001, pp. 364-380.
- [25] B. Nuseibeh, Weaving Together Requirements and Architectures. *IEEE Computer*, 34(3):115-117, March 2001.
- [26] B. Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, Oct. 1994.
- [27] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.
- [28] S. Rugaber. A Tool Suite for Evolving Legacy Software. In *ICSM'99*, Oxford, England, Aug/Sep 1999.
- [29] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.
- [30] R.N. Taylor, N. Medvidovic, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6), June 1996.
- [31] V. Tzerpos and R. C. Holt. A Hybrid Process for Recovering Software Architecture. In *CASCON'96*, Toronto, Nov. 1996.
- [32] K. Wong, S. Tilley, H. A. Muller, and M. D. Storey. Structural Redocumentation: A Case Study. *IEEE Software*, Jan. 1995.

Mapping requirements to software architecture by feature-orientation

Dongyun Liu, Hong Mei

Institute of Software, School of Electronics Engineering and Computer Science

Peking University, Beijing 100871, P.R.China

liudy@cs.pku.edu.cn, meih@pku.edu.cn

Abstract

Requirements engineering and software architecting are two key activities in software life cycle. Researchers have paid much attention to mapping and transformation from requirements to software architecture, but there's still lack of effective solutions. In this paper, the inadequacy of traditional mapping approaches (such as approaches in structured method and OO method) for this challenge is analyzed, and further a feature-oriented mapping approach is introduced. The rationale, process and guidelines for this approach are specified, and the approach is illustrated by an example of bank account and transaction (BAT) system.

1. Introduction

Requirements engineering and software architecting are two important activities in software life cycle. Requirements engineering is concerned with purposes and responsibilities of a system. It aims for a correct, consistent and unambiguous requirements specification, which will become the baseline for subsequent development, validation and system evolution. In contrast, software architecting is concerned with the shape of the solution space. It aims at making the architecture of a system explicit and provides a blueprint for the succeeding development activities. It is obvious that there exist quite

different perspectives in user (or customer) requirements and software architecture (SA). Mapping from requirements to SA is by no means trivial work. In traditional software development methods, the mapping relationship between requirements and SA is indirect and not straightforward, and existing mapping solutions are inadequate for mapping user (or customer) requirements to SA. In order to adapt to iterative, incremental and evolutionary development paradigm, it is necessary to make the mapping relationship between user (or customer) requirements and SA direct and straightforward, so as to support the traceability between requirements and SA more effectively.

As we have noticed, today more and more researchers pay their attentions to the research of feature-oriented software development methods. There have been efforts to apply feature to software development. In 1982, Davis [1] identified features as an important organization mechanism for requirements specification. In 1990 Kyo C. Kang [2] etc. proposed feature-oriented domain analysis (FODA) method. In this method, the concept of using feature model for requirements engineering was introduced. As a main activity in domain modeling, feature analysis is intended to capture the end-user's (and customer's) understanding of the general capabilities of applications in a domain. Later, FORM method [3] extends FODA to the software design and implementation phases and prescribes how the feature model is used to develop domain architectures and components for reuse. FORM method is quite fit for

software development in mature domain where standard terminology, domain experts and up-to-date documents are available. C. Reid Turner [4] puts forward a conceptual framework for feature engineering in 1999. Turner prefers to look feature as an important organizing concept within the problem domain and proposes carrying a feature orientation from the problem domain into the solution domain. Turner's framework comes from software development experience in telecommunication domain, and is still conceptual and incomplete. It does not provide particular solution for mapping requirements to SA from software engineering perspective. But above researches and practice show that it is feasible and effective to make features explicit in software development and to take feature orientation as a paradigm during the software life cycle.

In this paper, we will explore how to apply feature orientation as a solution for the mapping problem between requirements and SA from general software engineering perspectives, focusing on the mapping and transformation process. Our solution is to organize requirements in problem domain into a feature model, and then base our architectural modeling on the feature model, with the goal maintaining direct and natural mapping between requirements model and architecture models. Further, we will address functional features and nonfunctional features separately in different architectural models. Our approach is not a replacement of but an improvement on traditional methods. Our approach can integrate closely with OO method. The modeling concepts and notation adopted in this paper are based on UML, but have appropriate extension.

The rest of this paper is organized as follows. Section 2 analyzes the relationship between requirements engineering and software architecting, and specifies the necessity for supporting traceability between requirements and SA. Section 3 analyzes the inadequacy of mapping approaches in traditional methods. Section 4 proposes a feature-oriented mapping solution, and specifies the rationale, process and guidelines for this approach. Section 5 concludes the paper and further research effort is

envisioned. Application of our mapping approach to the bank accounts and transactions system (BAT) has been used in this paper as an illustrative example.

2. Requirements Engineering and Software Architecting

The IEEE standard [5] defines "requirement" as

"(1) A condition or capability needed by a user to solve a problem or achieve an objective.

(2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed document.

(3) A documented representation of a condition or capability as in (1) or (2)."

This definition is not so clear. In practice, requirements for a software system may exist in multiple different abstract levels, varying from organization's business requirements, through user's task requirements, to eventual software requirements specification (SRS).

Requirements engineering aims at reaching a good understanding of the problem domain and user's (or customer's) needs through effective problem analysis techniques, and producing a correct, unambiguous, complete and consistent requirements specification which serves as a baseline for developers to implement the software system. Requirements engineering only focuses on problem domain and system responsibilities, but not design and implementation details.

SA has become an important research field for software engineering community. There exists a consensus that for any large software system, its gross structure-that is, its high-level organization of computational elements and interactions between those elements-is a critical aspect of design [6][7]. It is widely accepted that SA is a very important product and software architecting is a necessary phase in software life cycle. As an important design concept, SA "can serve as the key milestone in the entire software life cycle process". SA's "support of the needs of system engineers, customers, developers, users, and

maintainers, also implies that is involved in all phases of the software and system life cycle”[8].

Until now software engineering researchers don't reach an agreement about the relationship between requirements engineering and software architecting. Following waterfall development model, software architects should not begin software architecting until a complete, correct and consistent requirements specification is reached. But some researchers[10] have pointed out that this model is discredited. In “multilevel life cycle chart” model, proposed by Merlin Dorfman [10], requirements engineering is involved throughout the software architecting process, that is, the steps of requirements analysis and design alternate. Rational Software Corporation [9] proposes the Unified Process, which is a use case driven, architecture-centric, and iterative and incremental process framework. In spite of existing different perspectives, now iterative, incremental, evolutionary and concurrent development paradigms are gaining more and more wide-spread acceptance. In development following such paradigms, it is more important to maintain direct and natural mapping and traceability between requirements specification and SA.

3. Traditional mapping approaches

Looking back on the development of software development methodology, it is not difficult to find that keeping the traceability and the consistency in concepts between requirements and designs always are the goals that we pursue. Two main software development methods, structured method and object-oriented method, both provide solutions for mapping analysis model to design model.

In structured method, software requirements are captured in Data Flow Diagram (DFD), and design is described in Module Structure Chart (MSC). Because there exists evident difference between the basic concepts and principles of DFD and MSC, mapping DFD to MSC is difficult and just by heuristic rules. Object-oriented approach cured the symptom that the structured paradigm

did not. Because Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) use the uniform basic concepts and principle, the mapping between OOA model and OOD model is natural and direct. Also, keeping traceability is easy and transformation could be done mechanically.

But both structured method and OO method don't provide complete solution for mapping requirements to SA indeed. On one hand, in both methods, SA and components are not paid enough attention to; On the other hand, both DFD and OOA model describe internal structure of the system from developer's view and not external behavior from end users' view. They contain some information that is not of interest to end-users (or customers). So there is still a gap between analysis model (DFD or OOA model) and user requirements description. Based on above analysis, we can conclude that, the mapping approaches in traditional methods are inadequate for mapping from requirements to SA.

4. Feature-oriented mapping

In this section we will explore how to apply feature orientation as a solution for mapping and transformation from requirements to SA aiming at improving traditional methods.

Feature has been defined in various ways, some important definitions are as follows: A feature is “a coherent and identifiable bundle of system functionality that helps characterize the system from the user perspective”[4]; A feature is “a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems”[2]; A feature is “an externally desired service by the system that may require a sequence of inputs to effect the desired result” [11]; A feature is “a service that the system provides to fulfill one or more stakeholder needs”[12]. We think that a feature is a higher-level abstraction of a set of relevant detailed software requirements, and is perceivable by users (or customers). So it is reasonable to identify features as “first-class entities” in requirements modeling, and

combine feature modeling and traditional requirements modeling together to describe requirements in different levels. Further, in architectural modeling we will take feature orientation as a goal, and try to maintain direct and natural mapping between feature model and architectural models at higher levels. By feature-orientation, we aim at making the mapping relationship between requirements and SA straightforward, which is impossible by traditional approaches.

We also have recognized the different effect of functional features and nonfunctional features on software architecture and address them separately. First, we get an initial partition of the proposed system into components based on functional features. Then, further optimization and transformation can be imposed on the partition, iteratively and incrementally, considering nonfunctional features.

The feature-oriented mapping process consists of two stages: feature-oriented requirements modeling and feature-oriented architectural modeling.

4.1 Feature-oriented requirements modeling

Feature-oriented requirements modeling is intended to capture users' (or customers') high-level expressions of desired system behavior in terms of application features, analyze the relationship between features, and then organize and refine the features into a feature-oriented requirements specification. Feature-oriented requirements modeling can be divided into three activities: feature elicitation, feature organization and analysis and feature refinement.

Feature elicitation

Features elicitation focuses on eliciting user requirements in terms of features. Keeping elicitation at abstract feature levels, we can avoid plunging into detailed functional requirements too early. Also, as the user often has expertise in the domain and knows the value of the features, problem analysis effort can be concentrated on user-desired features and unnecessary work can be reduced.

Users' (or customers') knowledge about problem domain is main source of features. Books, user manuals, etc. are also sources of features. Main feature elicitation techniques include interview, questionnaire, requirements workshop, and so on. In mature domains, analyzing the terminology of the domain language is also an effective and efficient way to identify features.

Feature analysis and organization

As potential features are identified and elicited, they are analyzed and organized into a feature hierarchy in a tree form. The features collected can be divided into functional features and nonfunctional features. All features reflect stakeholders' need to solve their problems. According Karl E. Weigers' view [13], stakeholders' requirements may exist in multiple levels, including business requirements, user requirements and functional requirements. As abstraction of functionality, features may exist at either business level or user level. A feature at business level describes the high-level desire of an organization or a customer for future system. Features at user level describe services which future system should provide for user tasks and constraints on the services. We first partition the features into the two levels, and we then further organize the features based on following criteria:

- The features to support a specific business process can be grouped and abstracted as a higher-level feature
- The features to support a specific user class can be grouped and abstracted as a higher-level feature
- A nonfunctional feature that is a constraint on a functional feature becomes a sub-feature of the functional feature.
- If a feature at user level is used to realize a feature at business level, then the former becomes a sub-feature of the latter. For instance, in the bank account and transaction system (BAT) example (see Figure 1), "identify client" feature is a realization of the nonfunctional feature "security", so "identify client" feature becomes a sub-feature of "security" feature.

There exist various relationships among the features. We

have identified several kinds of relationships: “composed-of”, “generalization/specialization”, “derived-from”, “constrained-by” and “dependent-on”, etc. As shown in Figure 1, “identify client” is derived from “security”, “withdraw money” is constrained by “response time $\leq 1\text{min}$ ”, all customer services is dependent on “identify client”.

Features themselves may be “mandatory” or “optional”. A mandatory feature is necessary for general users, and an optional feature is necessary for partial users. For example,

“withdraw money” is a “mandatory” feature, but “transfer money” is an “optional” feature.

Feature refinement

Now we have a feature hierarchy, but it is not specific enough for implementation. The next task is to refine the features into detailed functional requirements. Here use case technique can be used to elaborate a feature into a set of functionality.

Figure1 presents the resulting requirements model through feature-oriented modeling.

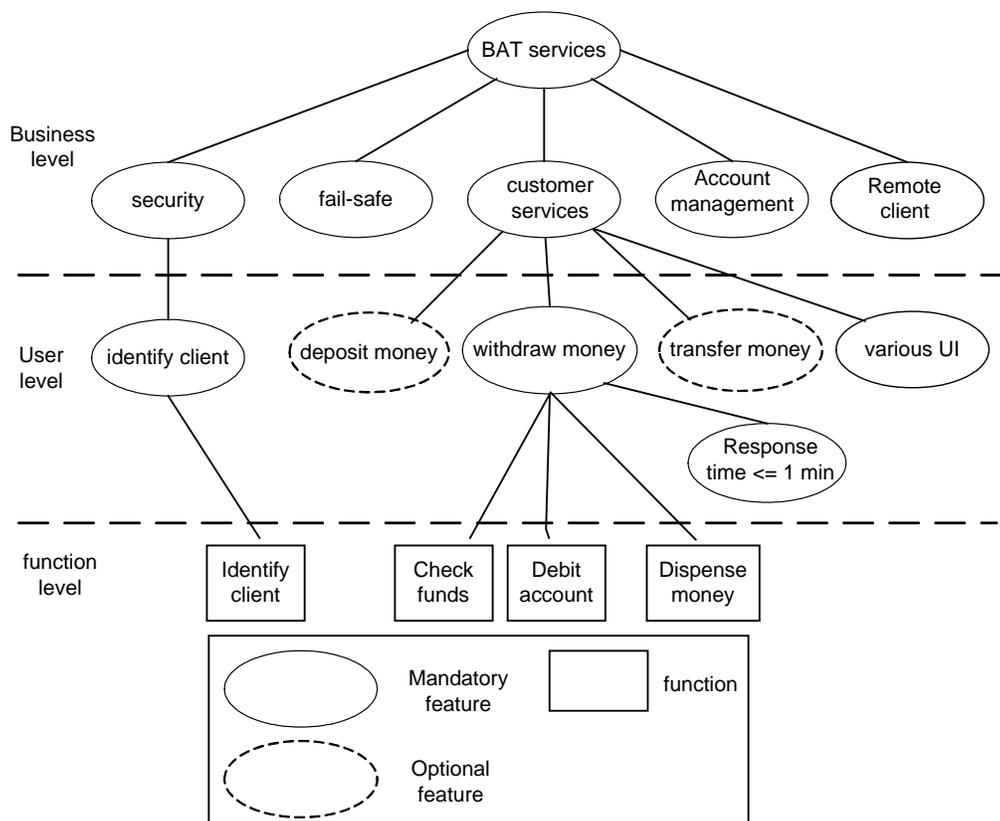


Figure 1. The feature model of BAT system

4.2 Feature-oriented architecture modeling

After we have got requirements specification organized by features, we can take it as an input to architecture modeling and derive SA from it. We will take feature prominence as a goal and try to maintain direct and natural mapping between feature model and architecture models.

Also, we have recognized that functional features and nonfunctional features have each different contribution to SA. A functional feature can be mapped to a subsystem or a component. Nonfunctional features can generally not be pinpointed to a particular component, but have impacts on the system structure and behavior as a whole. So we can address them separately in different models. We define SA

from three different viewpoints: conceptual architecture, logical architecture, and deployment architecture. As shown in figure 2, conceptual architecture focuses on the system's partition based on functional features, not considering nonfunctional features. Logical architecture focuses on logic design for addressing nonfunctional features, considering the implementation environment. Deployment architecture focuses on physical distribution of the system, addressing related nonfunctional features.

Conceptual architecture

The conceptual architecture identifies the system

components, the responsibilities of each component, and relationships between components in terms of application domain concepts. It tries to preserve the structure of problem domain by partitioning system based on functional features and problem domain structure. Each functional feature is mapped to a conceptual subsystem in the conceptual architecture, and each function at the function level can be mapped to an operation of a class in the class diagram. Figure3, Figure4 and Figure5 illustrate the three views of conceptual architecture in different levels of details, among which the lower-level view is a refinement of the higher-level view.

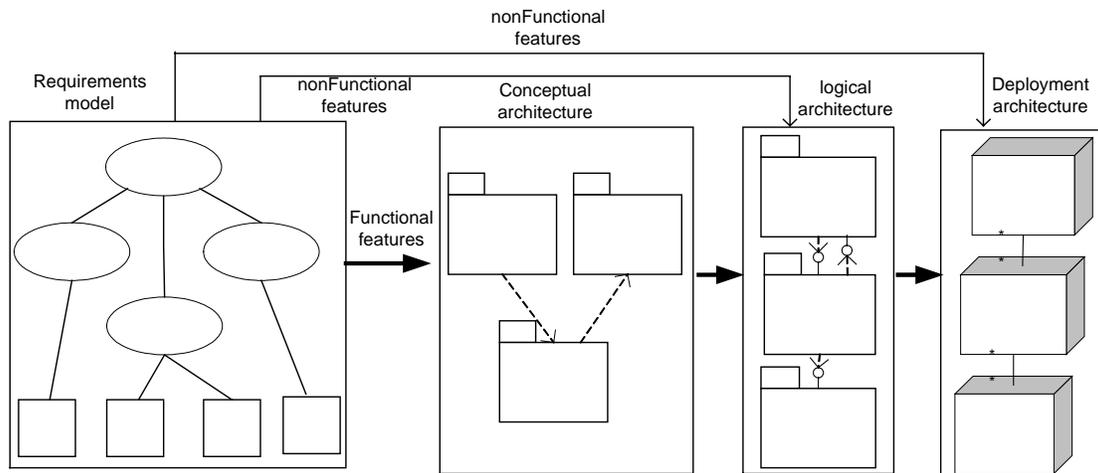


Figure 2. Mapping feature model to architecture models

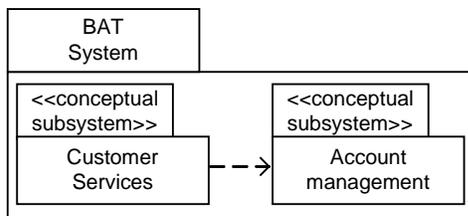


Figure 3. Business view of BAT conceptual architecture

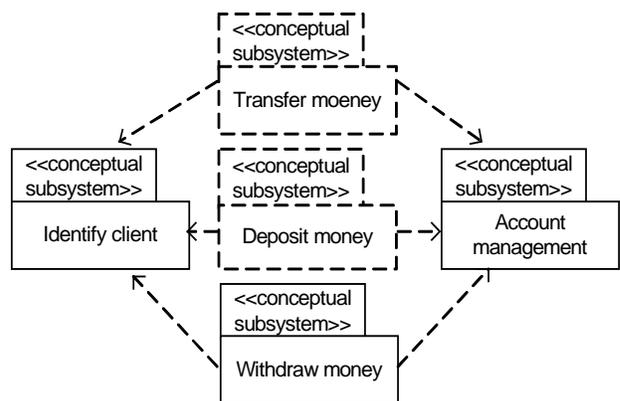


Figure 4. User view of BAT conceptual architecture

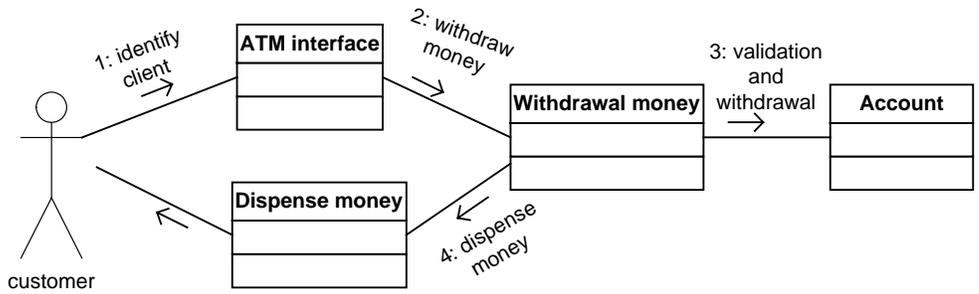


Figure 5. "Withdraw money" conceptual subsystem specification

Logical architecture

Logical architecture is the detailed architecture specification that defines the system components and interactions between components. Comparing with conceptual architecture, the logical architecture introduces more logical structures or mechanisms considering the implementation context and nonfunctional features. The form of the conceptual architecture may be adapted or

even transformed. As shown in figure 6, considering nonfunctional feature "various UI", we apply "separation of concerns" strategy to the conceptual architecture. We separate responsibility for user interface from responsibility for transaction management. So we got a new system partition: "ATM interface" subsystem, "Transaction management" subsystem and "Account management" subsystem.

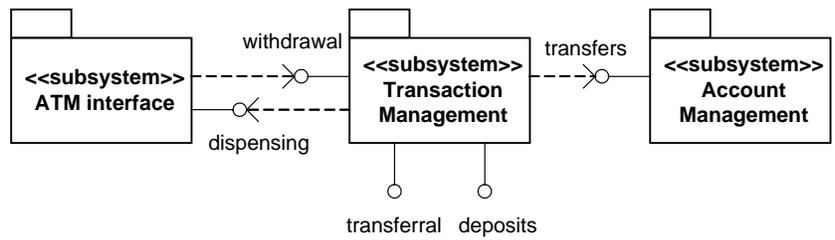


Figure 6. Logical architecture of BAT system

Deployment architecture

The deployment architecture focuses on how functionality is distributed among computational nodes and how computational nodes interact, to meet related nonfunctional features. As shown in figure 7, considering "remote client" and "fail-safe" features, a Three-Tier

architecture style is selected, and the CHAP acknowledgement protocol is adopted to ensure connection safety. Some components identified in conceptual architecture and logical architecture, such as "withdraw money" subsystem, is distributed to the three nodes in this view.

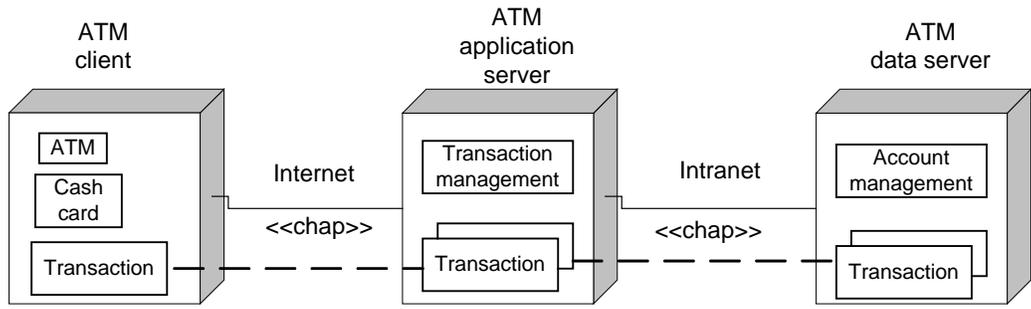


Figure 7. Deployment architecture of BAT system

5. Conclusion

In this paper, we analyze the gap between requirements and SA and the inadequacy of mapping approaches in traditional structured method and OO method. Based on this analysis, we propose a feature-oriented mapping and transformation approach. Our solution is to take feature-oriented as a paradigm both in requirements engineering and software architecting, so as to maintain direct and natural mapping between requirements specification and SA. Further, considering the different effect of functional features and nonfunctional features on SA, we address them separately in different models, iteratively and incrementally. So our approach can fit into iterative, incremental or evolutionary development paradigm.

We believe that feature-oriented development paradigm will gain more and more wide-spread acceptance. Further work will be to integrate our approach with existing CBSD development paradigm in order to support components reuse at different stages in software life cycle.

Acknowledgements

This effort gets support from China Natural Science Funds project under the contract 60125206. We would like to thank all the teachers and students who have discussed with us about this topic and give us good advice.

Reference

- [1] Davis, A.M. The design of a family of application-oriented requirements languages. *Computer* 15 (5) (1982) 21-28.
- [2] Kang, Kyo C. etc. Feature-Oriented Domain Analysis Feasibility Study (CMU/SEI-90-TR-21, ADA235785), CMU-SEI, 1990.
- [3] Kyo C. Kang , Sajoong Kim etc. FORM: A feature-oriented reuse method with domain-specific reference architectures, *Annals of Software Engineering* 5 (1998) 143-168
- [4] C. Reid Turner e.tc, A conceptual basis for feature engineering, *The Journal of Systems and Software* 49 (1999)

3-15

- [5] Institute of Electrical and Electronics Engineers. IEEE Standard Glossary of Software Engineering Terminology (IEEE Standard 610.12-1990). New York, N.Y.: Institute of Electrical and Electronics Engineers, 1990
- [6] David Garlan and Mary Shaw, An Introduction to Software Architecture, In *Advances in Software Engineering and Knowledge Engineering*, Volume 1, World Scientific Publishing Company, 1993.
- [7] Dewayne E. Perry and Alexander L. Wolf, Foundations for the Study of Software Architecture, *ACM SIGSOFT Software Engineering Notes*, 17(4), 1992.
- [8] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, and Barry Boehm, On the Definition of Software System Architecture, *ICSE 17 Software Architecture Workshop*, 1995.
- [9] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*, Addison Wesley Longmon, 1999
- [10] Merlin Dorfman. *Requirements Engineering*, SEI Interactive, March,1999
- [11] Institute of Electrical and Electronics Engineers. IEEE Recommended Practice for Software Requiements Specifications (IEEE Std 830-1998), New York, N.Y.: Institute of Electrical and Electronics Engineers, 1998
- [12] Dean Leffingwell, *Managing Software Requirements: A Unified Approach*, AT&T, 2000
- [13] Karl E. Wiegers, *Software Requirements*, Microsoft Corporation, 2000

Architecture as an Emergent Property of Requirements Integration

R.G. Dromey,
Software Quality Institute, Griffith University,
Nathan, Brisbane, Qld., 4111, AUSTRALIA
rgd@cit.gu.edu.au

Abstract

Despite the advances in software engineering since 1968, how to go from a set of functional requirements to an architecture that accommodates those requirements remains a challenging problem. Progress with this fundamental problem is possible once we recognize (1) that individual functional requirements represent fragments of behaviour, (2) a design that *satisfies* a set of functional requirements represents integrated behaviour, and (3) an architecture must accommodate the integrated behaviour expressed in a set of functional requirements. This perspective admits the prospect of constructing a design *out of* its requirements. A formal representation for individual functional requirements, called *behavior trees* makes this possible. Behaviour trees of individual functional requirements may be composed, one at a time, to create an integrated design behaviour tree. From this *problem domain* representation it is then possible to transition directly and systematically to a *solution domain* representation of the component architecture of the system and the behaviour designs of the individual components that make up the system – both are emergent properties.

“Finding deep simplicities in a complex logical task leads to work reduction”- Harlan Mills.

1. Introduction

A great challenge that continues to confront software engineering is how to go in a systematic way from a set of functional requirements to a design that will satisfy those requirements and an architecture that will support the implied integrated behavior. In practice, these two tasks are further complicated by defects in the original requirements and, subsequent changes to the requirements.

A first step towards taking up this challenge is to ask – what are functional requirements? Study of diverse sets of functional requirements suggests it is safe to conclude that individual requirements express *constrained behaviour*. By comparison, a system that satisfies a set of functional requirements exhibits *integrated constrained behaviour*. The latter behaviour of systems is not inherently different.

Functional requirements contain, and systems exhibit, the behavior summarized below.

- Components realise states
- Components change states
- Components have sets of attributes that are assigned values
- Components, by changing states, can cause other components to change their states
- Supplementing these component-state primitives are conditions/decisions, and events involving component-states.
- Interactions between components also play a key role in describing behaviour. They involve control-flow and/or data-flow between components.

Notations like sequence diagrams, class and activity diagrams from UML[1], data-flow diagrams, Petri Nets[2], state-charts and Message Sequence Charts (MSCs) [3,4], accommodate the behaviour we find expressed in functional requirements and designs. Individually however, none of these notations provide the level of constructive support we need. This forces us to contemplate another representation for functional requirements and designs. Such ventures are generally not enthusiastically received – a consensus is that new notations just muddy the waters. Our justification for ignoring this advice is that the *Behavior Tree Notation* solves a fundamental problem – it provides a clear, simple, constructive path for going first from a set of functional requirements to an integrated behaviour representation that will satisfy those requirements and then to an architecture and the set of accompanying component behaviour designs [5].

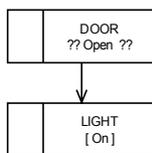
2. Behavior Trees

The Behavior Tree Notation captures in a simple tree-like form of composed component-states what usually needs to be expressed in a mix of other notations. Behavior is expressed in terms of components realizing states, augmented by the logic and graphic forms of conventions found in programming languages to support composition, events, control-flow data-flow, threads, and

constraints. Behavior trees are equally suited to capture behavior expressed in natural language functional requirements as to provide an abstract graphical representation of behavior expressed in a program. We may therefore ask can the same formal representation of behaviour be used for requirements and for a design? If it could it may clarify the requirements-design-architecture relationship.

Definition: A Behavior Tree is a formal, composable, tree-like graphical form that represents behaviour of individual or networks of entities which realize or change states, make decisions, respond-to/cause events, and interact by exchanging information and/or passing control.

Behavior trees provide a direct and clearly traceable relationship between what is expressed in the natural language representation and its formal specification. Translation is carried out on a sentence-by-sentence basis, e.g., the sentence “when the door is opened the light should go on” is translated to the behaviour tree below:



with a component a [State], ??Event??. ?Decision?, [Sub-cpt[State] or relation, or [Attribute := expression | State]. Exactly what can be an event, a decision, a state, etc are built on the formal foundations of expressions, Boolean expressions and quantifier-free formulae (qff). To assist with traceability to original requirements a simple convention is followed. Tags (e.g. R1 and R2, etc, see below) are used to refer to the original requirement in the document that is being translated. System states, are used to model high-level (abstract) behaviour and some preconditions/postconditions. Key elements of the notation are given in Figure 1, above (see EBNF, semantics, web-site <http://www.sqi.gu.edu.au/gse/papers>).

In practice, when translating functional requirements into behavior trees we often find that there is a lot of behavior that is either *missing* or is only *implied* by a requirement. We mark implied behavior with a “+” in the tag (and/or the colour yellow if colour can be shown). Behavior that is missing is marked with a “-“ in the tag (and/or the colour red). Explicit behavior in the original requirement that is translated and captured in the behavior tree has no “+/-“ marking, and the colour green is used - see Fig. 4 below. These conventions maximize traceability to original requirements.

3. Genetic Software Engineering Method

Conventional software engineering applies the underlying design strategy of constructing a design that will *satisfy* its set of functional requirements. In contrast to this, a clear advantage of the behavior tree notation is that it allows us to construct a design *out of* its set of functional requirements, by integrating the behavior trees for individual functional requirements (RBTs), one-at-a-time, into an evolving design behavior tree (DBT). This very significantly reduces the complexity of the design process and any subsequent change process [5].

What we are suggesting is that a set of functional requirements, represented as behavior trees, in principal at least (when they form a complete and consistent set), contains enough information to allow their composition. This property is the exact same property that a set of pieces for a jigsaw puzzle possess. And, interestingly, it is the same property which a set of genes that create a living entity possess. Witness the remark by geneticist Adrian Woolfson: in his recent book ([6], p.12), *Living Without Genes*, “we may thus imagine a gene kit as a cardboard box filled with genes. On the front and sides of the box is a brightly coloured picture of the creature that might in principle be constructed if the information in the kit is used to instruct a biological manufacturing process”

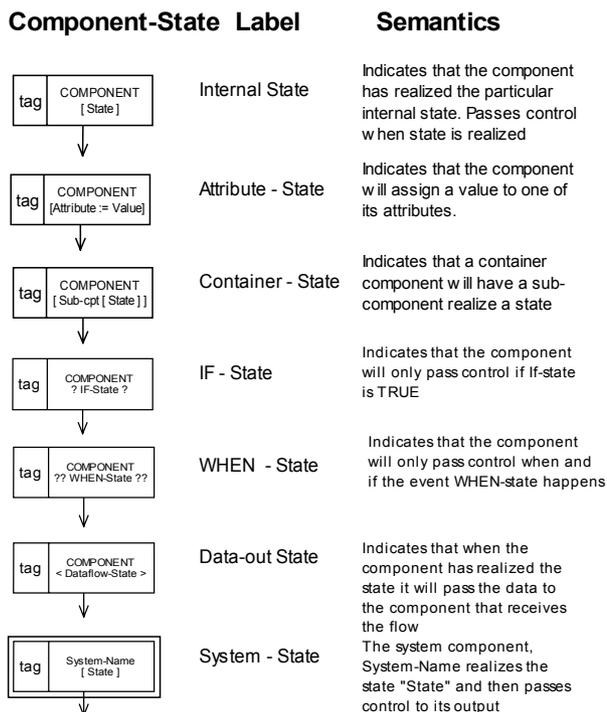


Figure 1. Behavior Tree Notation, key elements

The principal conventions of the notation for component-states are the graphical forms for associating

The obvious question that follows is: “what information is possessed by a set of functional requirements that might allow their composition or integration?” The answer follows from the observation that the behaviour expressed in functional requirements does not “just happen”. There is always a *precondition* that must be satisfied in order for the behaviour encapsulated in a functional requirement to be accessible or applicable or executable. In practice, this precondition may be embodied in the behaviour tree representation of a functional requirement (as a component-state or as a composed set of component states) or it may be missing - the latter situation represents a defect that needs rectification. The point to be made here is that this precondition is needed, in each case, in order to integrate the requirement with at least one other member of the set of functional requirements for a system. (In practice, the root node of a behaviour tree *often* embodies the precondition we are seeking). We call this foundational requirement of the genetic software engineering method, the *precondition axiom*.

Precondition Axiom

Every constructive, implementable individual functional requirement of a system, expressed as a behavior tree, has associated with it a precondition that needs to be satisfied in order for the behavior encapsulated in the functional requirement to be applicable.

A second building block is needed to facilitate the composition of functional requirements expressed as behavior trees. Jigsaw puzzles, together with the precondition axiom, give us the clues as to what additional information is needed to achieve integration. With a jigsaw puzzle, what is key, is not the order in which we put the pieces together, but rather the *position* where we put each piece. If we are to integrate behavior trees in any order, one at a time, an analogous requirement is needed. We have already said that a functional requirement’s precondition needs to be satisfied in order for its behaviour to be applicable. It follows that some *other* requirement, as part of its behavior tree, must establish the precondition. This requirement for composing/integrating functional requirements expressed as behaviour trees is more formally expressed by the following axiom.

Interaction Axiom

For each individual functional requirement of a system, expressed as a behavior tree, the precondition it needs to have satisfied in order to exhibit its encapsulated behavior, must be established by the behavior tree of at least one other functional requirement that belongs to the set of functional requirements of the system. (The functional requirement that forms the root of the design behavior tree, is excluded from this requirement. The external environment makes its precondition applicable).

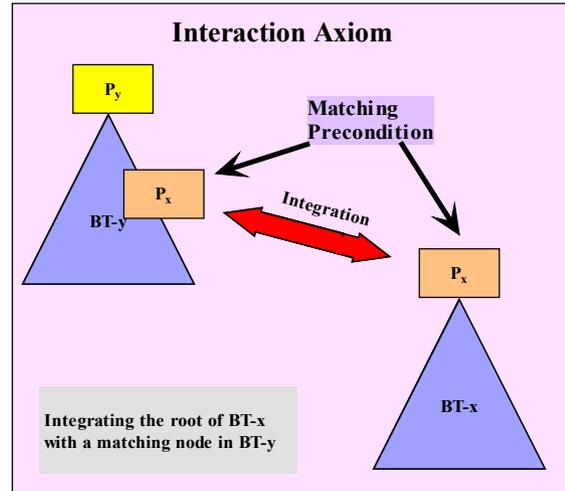


Figure 2. Interaction Axiom - graphic form

The precondition axiom and the interaction axiom play a central role in defining the relationship between a set of functional requirements for a system and the corresponding design. What they tell us is that the first stage of the design process, in the problem domain, can proceed by first translating each individual natural language representation of a functional requirement into one or more behavior trees. We may then proceed to integrate those behavior trees just as we would with a set of jigsaw puzzle pieces. What we find when we pursue this whole approach to software design is that the process can be reduced to the following four overarching steps:

- Requirements translation – (problem domain)
- Requirements integration – (problem domain)
- Component architecture transformation
- Component behaviour projection

Each overarching step, needs to be augmented with a validation and refinement step designed specifically to isolate and correct the class of defects that show up in the different work products generated by the process.

Comprehensive description, formalization, and justification of a software development method and notation, like the one here, requires significantly more than a conference paper length treatment. To maximize communication we will only introduce the main ideas of the method informally and show how the architecture and component designs are obtained. The process is best understood in the first instance by observing its application to a simple example. For our purposes, and for the purposes of comparison, we will use a design example for a Microwave Oven that has already been published in the literature [7]. The seven stated functional requirements for the Microwave Oven problem [7, p.36] are given in Table I below. Shlaer, and Mellor have applied their state-based Object-Oriented Analysis method to this set of functional requirements.

Table 1. Functional Requirements for Microwave Oven

<p>R1. There is a single control button available for the user of the oven. If the oven is idle with the door is closed and you push the button, the oven will start cooking (that is, energize the power-tube for one minute).</p> <p>R2. If the button is pushed while the oven is cooking it will cause the oven to cook for an extra minute.</p> <p>R3. Pushing the button when the door is open has no effect (because it is disabled).</p> <p>R4. Whenever the oven is cooking or the door is open the light in the oven will be on.</p> <p>R5. Opening the door stops the cooking.</p> <p>R6. Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.</p> <p>R7. If the oven times-out the light and the power-tube are turned off and then a beeper emits a sound to indicate that the cooking is finished.</p>

3.1 Requirements Translation

After preliminary glossary/vocabulary processing and removal of aliases, etc, requirements translation is the first major step in the Genetic Software Engineering (GSE) design process. Its purpose is to translate each natural language functional requirement, one at a time, into one or more behavior trees. Translation identifies the *components* (including actors and users), the *states* they realise (including attribute assignments), the *events* and *decisions/constraints* that they are associated with, the *data* components exchange, and the *causal, logical* and *temporal* dependencies associated with component interactions.

Example Translation

The translations for the first six functional requirements for the Microwave Oven given in Table 1 are shown in figure 4. Translation of R7 from Table 1 will now be considered in slightly more detail. For this requirement we have underlined the states/actions and made the components **bold**, i.e., “If the **oven times out** the **light** and the **power-tube** are **turned off** and a **beeper emits a sound** to indicate that **cooking has finished**”. Figure 3. (see below) gives a translation of this requirement R7, to a corresponding requirements behavior tree (RBT). In this translation we have followed the convention of trying wherever possible to associate higher level system states (here OVEN states) with each functional requirement, to act as preconditions/postconditions.

What we see from this translation process is that even for a very simple example, it can identify problems that, on the surface, may not otherwise be apparent (e.g. the original requirement, as stated, leaves out the precondition that the oven needs to be cooking in order to subsequently time-out). In addition, the behavior tree representation tags (here R7) are able to provide very direct traceability back to the original statement of requirements. Our claim is that the translation process has good repeatability if

translators forego the temptation to interpret, design, and introduce new things as they do an initial translation.

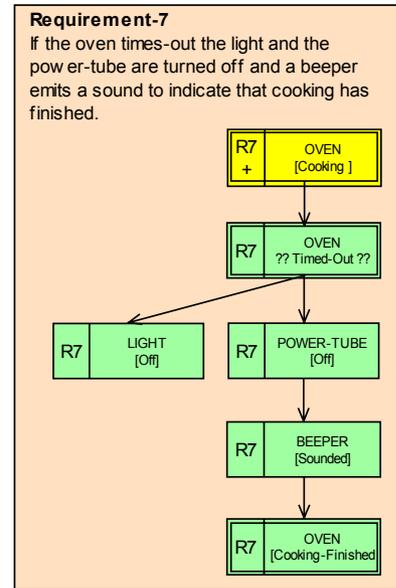


Figure 3. Behavior Tree for Requirement R7

3.2 Requirements Integration

When requirements translation is completed each individual functional requirement has been translated to one or more corresponding requirements behavior tree(s) (RBT). We can then systematically and incrementally construct a design behavior tree (DBT) that will satisfy all its requirements *by integrating the requirements' behavior trees* (RBT). Integrating two behavior trees turns out to be a relatively simple process that is guided by the precondition and interaction axioms referred to above. In practice, it most often involves locating where, (if at all) the component/state root node of one behavior tree occurs in the other tree and grafting the two trees together at that point. This process generalises when we need to integrate N behavior trees. We only ever attempt to integrate two behavior trees at a time – either two RBTs, an RBT with a DBT or two partial DBTs. In some cases, because the precondition for executing the behavior in an RBT has not been included, or important behaviour has been left out of a requirement, it is not clear where a requirement integrates into the design. This immediately points to a problem with the requirements. In other cases, there may be requirements/behavior missing from the set which prevents integration of a requirement. Attempts at integration uncover such problems with requirements at the earliest possible time.

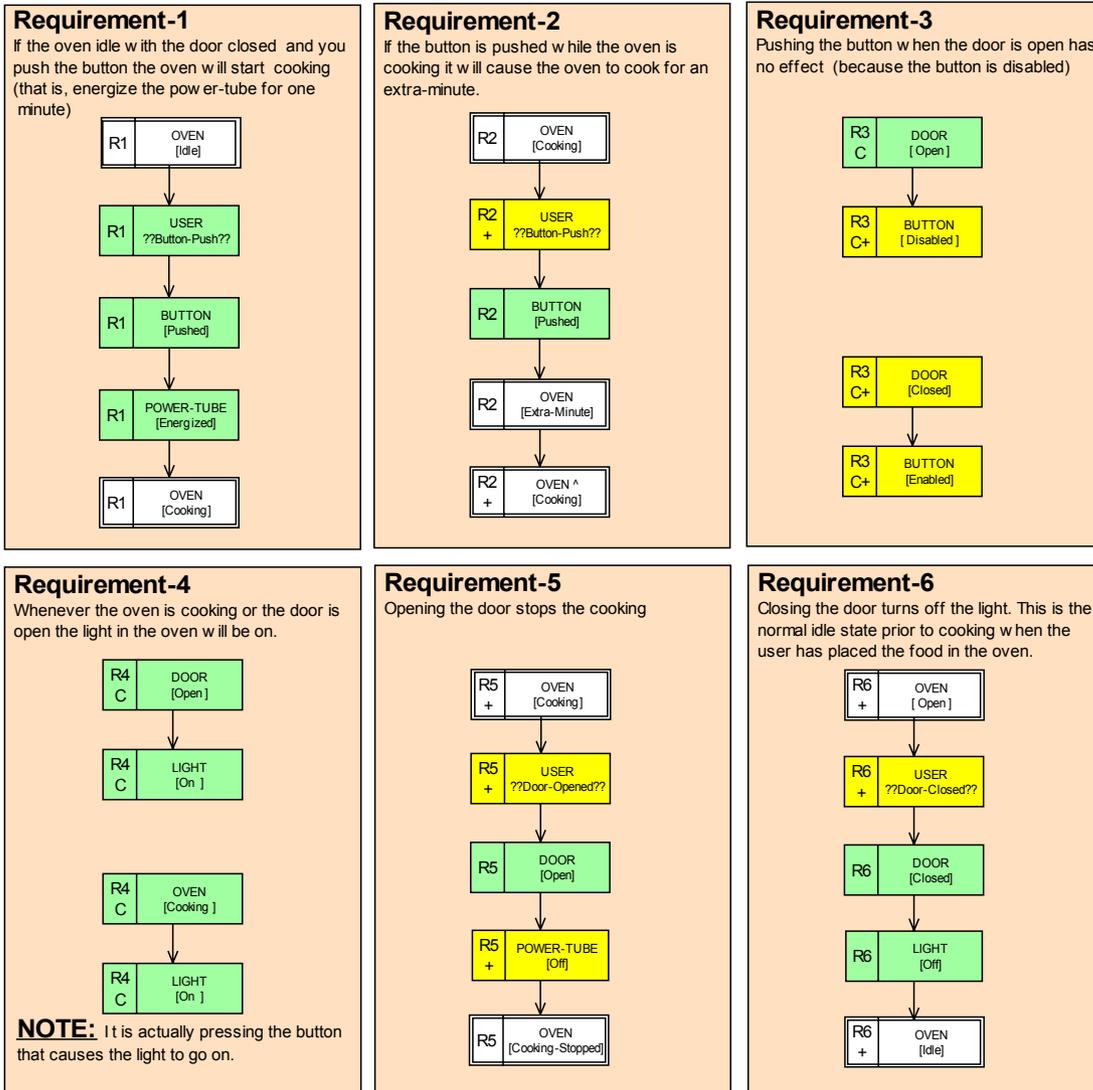


Figure 4. Behavior trees for Microwave Oven

Example Integration

To illustrate the process of requirements integration we will integrate requirement R6, with part of the constraint Requirement R3C to form a partial design behaviour tree (DBT). This is straightforward because the root node (and precondition) of R3C, DOOR[Closed] occurs in R6. We integrate R3C into R6 at this node. Because R3C is a constraint it should be integrated into every requirement that has a door closed state (in this case there is only one such node). The result of the integration is shown below.

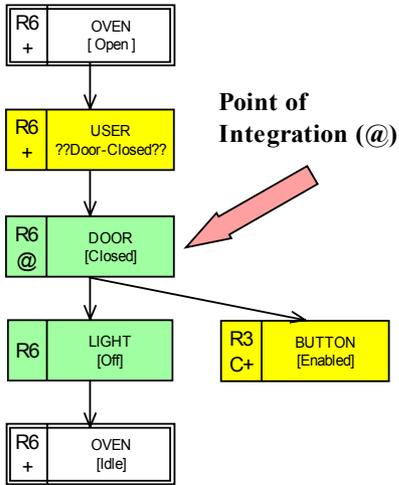


Figure 5. Result of Integrating R6 and R3C

When R6 and R3C have been integrated we have a “partial design” (the evolving design behavior tree) whose behavior will satisfy R6, and the R3C constraint. In this DBT it is clear and traceable where and how each of the original functional requirements contribute to the design.

Using this same behavior-tree grafting process, a complete design is constructed (it evolves) incrementally by integrating RBTs and/or DBTs pairwise until we are left with a single final DBT (see Figure 6 below). This is the ideal for design construction that is realizable when all requirements are consistent, complete, composable and do not contain redundancies. When it is not possible to integrate an RBT or DBT with any other it points to an integration problem with the specified requirements that needs to be resolved. Being able to construct a design incrementally, significantly reduces the complexity of this critical phase of the design process. And importantly, it provides direct traceability to the original natural language statement of the functional requirements. From a careful inspection of the integrated DBT (Fig. 6) we see that there is a missing requirement associated with opening the oven when it is idle. This has been added as requirement R8. Note with constraint R4 we have used the causal relationship for the light turning on rather than the literal translation of the requirement.

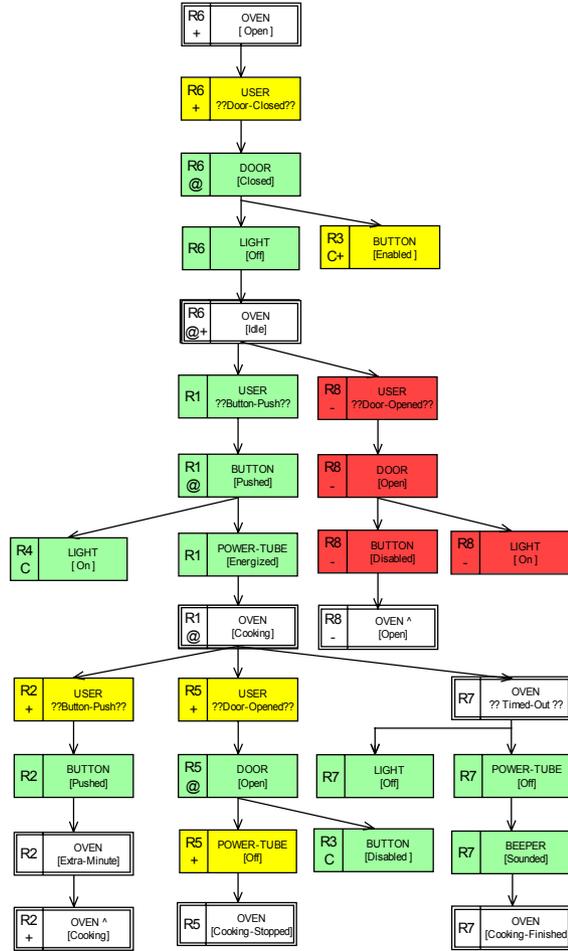


Figure 6. Integration of all functional requirements

Once the design behavior tree (DBT) has been constructed the next jobs are to transform it into its corresponding software or component architecture (or *component interaction network* - CIN) and then project from the design behavior tree the component behavior trees (CBTs) for each of the components mentioned in the original functional requirements.

3.4 Architecture Transformation

A design behavior-tree is the *problem domain* view of the “shell of a design” that shows all the states and all the flows of control (and data), modelled as component-state interactions without any of the functionality needed to realize the various states that individual components may assume. *It has the genetic property of embodying within its form two key emergent properties of a design: (1) the component-architecture of a system and, (2) the behaviors of each of the components in the system.* In the DBT representation, a given component may appear in different

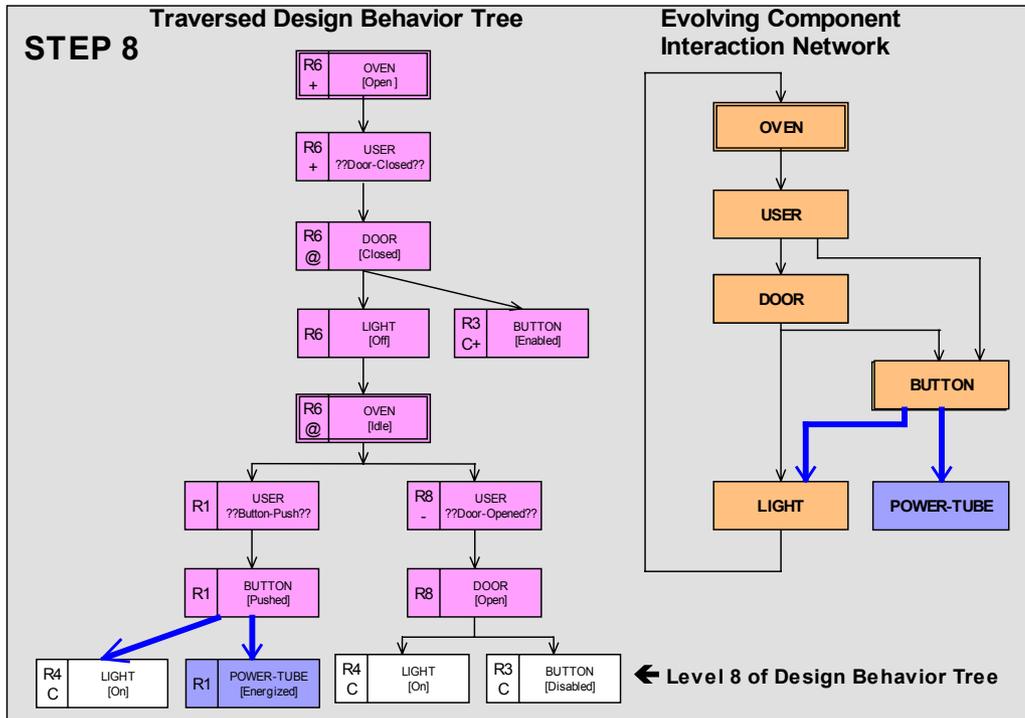


Figure 7. A step in the Tree-to-Network Transformation

parts of the tree in different states (e.g., the OVEN component may appear in the Open-state in one part of the tree and in the Cooking-state in another part of the tree). Interpreting what we said earlier in a different way, we need to convert a design behavior-tree to a component-based design in which each distinct component is represented only once. This amounts to shifting from a representation where functional requirements are integrated to a representation, which is part of the *solution domain*, where the components mentioned in the functional requirements are themselves integrated. A simple algorithmic process may be employed to accomplish this transformation from a tree into a network. *Informally, the process starts at the root of the design behavior tree and moves systematically down the tree towards the leaf nodes including each component and each component interaction (e.g. arrow) that is not already present.* When this is done systematically the tree is transformed into a component-based design in which each distinct component is represented only once. We call this a Component Interaction Network (CIN) representation. Above, we show the eighth step of this transformation, involving the components on the eighth level of the DBT. Here the POWER-TUBE gets included into the CIN network and the link between the BUTTON and the LIGHT is added to the network.

The complete Component Interaction Network derived from the Microwave Oven design behavior tree is shown below in Figure 8. It defines the component-component

interactions and therefore the interfaces for each component. It also captures the “business model” or “conceptual design” for the system and represents the first cut at the software architecture for a system. The next important task is to isolate the behaviors of the individual components present in the architecture from the DBT using projection.

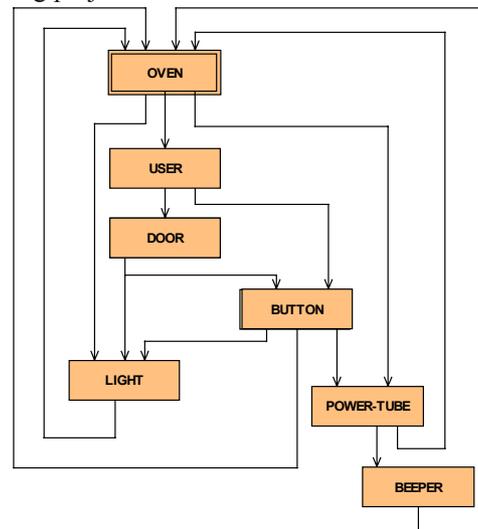


Figure 8. Component Interaction Network - (CIN)

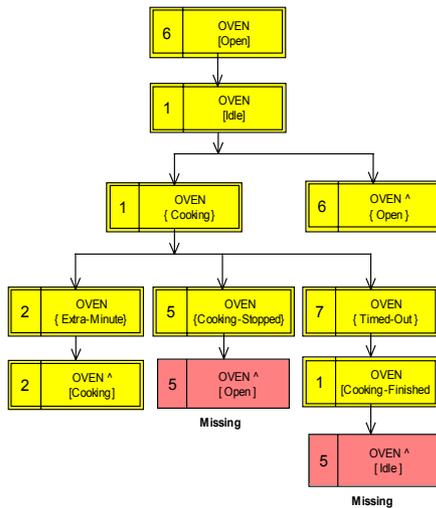
3.4 Component Behavior Projection

In the design behavior tree, the behavior of individual components tends to be dispersed throughout the tree (for example, see the OVEN component-states in the Microwave Oven System DBT). To implement components that can be embedded in, and operate within, the derived component interaction network, it is necessary to “concentrate” each component’s behavior. We can achieve this by systematically *projecting* each component’s behavior tree (CBT) from the design behavior tree. We do this by essentially ignoring the component-states of all components other than the one we are currently projecting. The resulting connected “skeleton” behavior tree for a particular component defines the behavior of the component that we will need to implement and encapsulate in the final component-based implementation.

Example – Component Behavior Projection

To illustrate the effect and significance of component behavior projection we show the projection of the OVEN SYSTEM component from the DBT for the Microwave Oven.

OVEN COMPONENT - Projected Behavior



Component behavior projection is a key design step in the solution domain that needs to be done for each component in the design behavior tree. When this process has been carried out for ALL the components in the DBT, that is, USER, BUTTON, etc, all the behavior in the DBT has been projected into the components that are intended to implement the system. ***That is, the complete set of component behavior projections conserve the behavior that was originally present in the DBT.*** What this set of component projections allows us to achieve is a metamorphosis from an integrated set of functional requirements to an integrated component based design. To complete the component-based design, we embed the behaviors of each component into the architectural design provided by the component interaction network (CIN) –

see, for example figure 8 above. The tasks that then remain are to rationalize the component interfaces and to implement the component interaction network which supports the component interactions that, in turn, implement the system behaviors. And finally, we must provide implementations to support the behaviors exhibited by each of the components. Component integration can be done using either the facilities of a component framework [1] or by using a standard code implementation that maps the graphic network into code.

In a number of reports and presentations at <http://www.sqi.gu.edu.au/gse/papers> we provide a more detailed account of the GSE method, the notation and its application to a diverse set of problems including contract automation and much larger applications. We also provide examples that show how to translate the designs that the method produces into object-oriented and component-based implementations in Java.

Conclusion

What we have presented is an intuitive, stepwise process for going from a set of functional requirements to a design and a supporting architecture. The method is attractive for its simplicity, its traceability, its ability to detect defects, its control of complexity, and its accommodation of change. Derivation of the software component architecture from the design behavior tree and projection of the set of component behavior trees from a design behavior tree are both repeatable, algorithmic processes, that can be automated if we choose to do so. The greatest chance for variation with work products comes in the translation of natural language descriptions of functional requirements to requirements behavior trees (RBTs)

References

[1] G.Booch, J. Rumbaugh, I Jacobson, The Unified Modelling Language User Guide, Addison-Wesley, Reading, Mass. (1999).

[2] A.M.Davis, A Comparison of Techniques for the Specification of External System Behavior, Comm. ACM, vol. 31 (9), 1098-1115, (1988).

[3] D. Harel., W. Damm, LSCs: Breathing Life into Message Sequence Charts, 3rs IFIP Conf. On Formal Methods for Open Objected-based Distributed Systems, New York, 1999, Kluwer

[4] S.Uchitel, J.Kramer, A Workbench for Synthesizing Behavior Models from Scenarios, 23rd International Conference on Software Engineering (ICSE’01), Toronto, Canada, 2001.

[5] R.G.Dromey, Genetic Software Engineering - Simplifying Design Using Requirements Integration, IEEE Working Conference on Complex and Dynamic Systems Architecture, S4, pp. 1-16, Brisbane, Dec 2001.

[6] A. Woolfson, Living Without Genes, Flamingo, (2000).

[7] S. Shlaer, S.J. Mellor, Object Lifecycles, Yourdon Press, New Jersey, 1992.

Detailing Architectural Design in the Tropos Methodology

Carla T. L. L. Silva¹, Jaelson F. B. Castro¹, John Mylopoulos²

¹ Centro de Informática, Universidade Federal de Pernambuco, Av. Prof. Luiz Freire S/N, Recife PE, Brazil 50732-970, +1 5581 {ctlls,jbc}@cin.ufpe.br

² Dept. of Computer Science University of Toronto, 10 King's College Road Toronto M5S3G4, Canada, +1 416 978 5180 jm@cs.toronto.edu

Abstract

Software systems development happens within a context which organizational processes are well-established. Hence, software needs to be built with flexible architectures based in social and intentional concepts to enable software to evolve consistently with its operational environment. In this sense, the Tropos requirements oriented development methodology, has defined a number of organizational architectural styles which are suitable to agent, cooperative, dynamic and distributed applications. In this paper, we use an extended version of UML to describe these novel architectural styles in order to provide a detailed representation of both the structure and behaviour of the architectural design using these styles. This proposal has been applied to an e-commerce software system.

1. Introduction

Companies are continually changing and turning their attention to improve their business strategies. Stakeholders are demanding more flexible and complex systems. Hence, software has to be based on architectures that can evolve and change continually to accommodate new components and meet new requirements. A flexible architecture with loosely coupled components is much more likely to accommodate new feature requirements than one that has been highly optimized for just its initial set of requirements. Tropos [1], a requirements-driven development methodology, has defined organizational architectural styles [6],[7],[8] based on concepts and design alternatives coming from research in organization management, used to model coordination of business stakeholders – individuals, physical or social systems. Tropos relies on the i* notation [4] to describe both requirements and organizational architectural styles. Unfortunately, this notation is not widely accepted by software practitioners nor able to represent some detailed

information which sometimes is required in architectural design such as set of signals that are exchanged between architectural components, as well as the valid sequence of these signals (protocol). On the other hand, the Unified Modeling Language – UML [3] has been extended and used to represent the architecture of simple and complex systems. Such an architecture description language is based on UML for Real-Time systems (UML-RT), an UML extension tuned for real time software systems.

In an effort to provide detailed representation in architectural phase of Tropos methodology, as well as to represent the organizational architectural styles into a mainstream industrial notation, in this work we propose to accommodate within UML-RT the concepts and features used for representing organizational architectures into Tropos. In order to validate this proposal, we applied it to an e-commerce software system extracted from [1]. This work is an improvement of another attempt for representing the Tropos concepts in UML [2].

The rest of this paper is organized as follows: Section 2 presents the Tropos methodology. Section 3 describes how software architecture can be modeled using UML. In Section 4, we define how organizational architectures can be modeled using UML-RT. Section 5 depicts the application of the proposal to a case study. Section 6 points to some future work and discusses the contribution of this proposal.

2. The Tropos Methodology

Tropos proposes a software development methodology and a development framework which are founded on concepts used to model early requirements and complements proposals for agent-oriented programming platforms. This methodology is based on the premise that in order to build software that operates within a dynamic environment, one needs to analyze and model explicitly that environment in terms of “actors”, their goals

and dependencies on other actors. Tropos supports five 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.

In this work, our focus is on architectural design phase. Software architecture is more than just structure, it includes rules on how system functionality is achieved across the structure. Unfortunately, traditional architectural styles for e-business applications [12],[13] focus on web concepts, protocols and underlying technologies but not on business processes nor non functional requirements of the application. As a result, the organizational architecture styles are not described nor the conceptual high-level perspective of the e-business application.

Tropos has defined organizational architectural styles [6],[7],[8] for agent, cooperative, dynamic and distributed applications to guide the design of the system architecture. These architectural styles (*pyramid, joint venture* (Fig. 1), *structure in 5, takeover, arm's length, vertical integration, co-optation, bidding, ...*) are based on concepts and design alternatives coming from research on organization management. From this perspective, software system is like a social organization of coordinated autonomous components that interact in order to achieve specific and possibly common goals. The purpose is to reduce as much as possible the impedance mismatch between the system and its environment.

For example, the joint venture architectural style (Figure 1) allows a decentralized architecture. The main feature of this style is that it involves an agreement between two or more principal partners/components in order to obtain the benefits derived from operating at a large scale, such as partial investment and lower maintenance costs, as well as reusing the experience and knowledge of the partners/components, since they pursue joint objectives.

To support modeling and analysis during the initial phases, Tropos adopts the concepts offered by *i** [4], a modeling framework offering concepts such as *actor* (actors can be *agents, positions or roles*), as well as social dependencies among actors, including *goal, softgoal, task*

and *resource* dependencies. This means that both the system's environment and the system itself are seen as organizations of actors, each having goals to be fulfilled and each relying on other actors to help them with goal fulfillment.

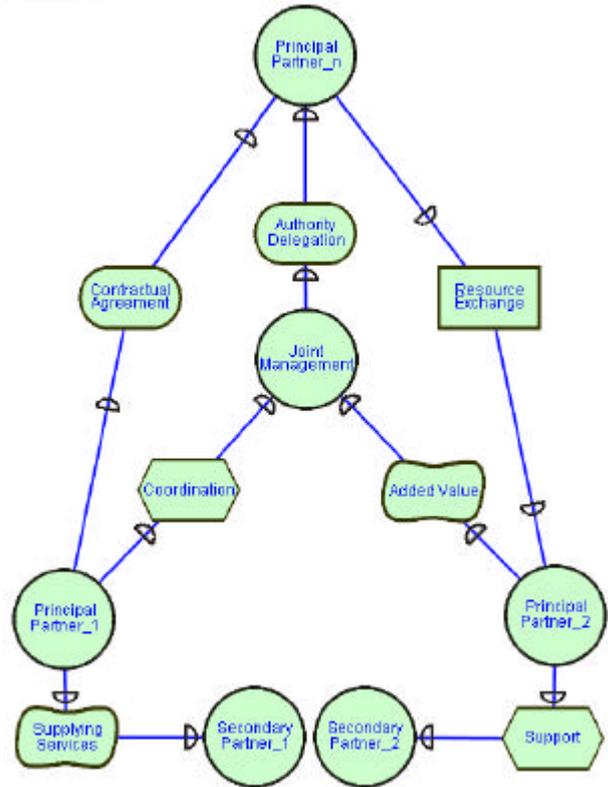


Figure 1. Joint Venture

As shown in Figure 1, actors are represented as circles; dependums -- goals, softgoals, tasks and resources -- are respectively represented as ovals, clouds, hexagons and rectangles; and dependencies have the form *dependor*⇒*dependum*⇒*dependee*. Hence, in Tropos we have the following concepts:

- Actor: An actor is an active entity that carries out actions to achieve goals by exercising its know-how.
- Dependency: A dependency describes an intentional relationship between two actors, i.e., an “agreement” (called *dependum*) between two actors: the *dependor* and the *dependee*, where one actor (*dependor*) depends on another actor (*dependee*) on something (*dependum*).
- Dependur: The dependur is the depending actor.
- Dependee: The dependee is the actor who is depended upon.
- Dependum: The dependum is the type of the dependency and describes the nature of the agreement.

- Goal: A goal is a condition or state of affairs in the world that the stakeholders would like to achieve. How the goal is to be achieved is not specified, allowing alternatives to be considered.

- Softgoal: 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 judgment and interpretation of the developer to judge whether a particular state of affairs in fact achieves sufficiently the stated softgoal.

- Resource: A resource is an (physical or informational) entity, with which the main concern is whether it is available.

- Task: A task specifies a particular way of doing something. Tasks can also be seen as the solutions in the target system, which will satisfy the softgoals (operationalizations). 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.

The first task during architectural design is to select among alternative architectural styles using as criteria the desired qualities identified in the previous phase (Late Requirements). To this end, the NFR framework [5] can be used to conduct the selection of the most suitable organizational architectural style. More details about the selection and non-functional requirements decomposition process can be found in [6],[7].

In the next section, we show how architectural design can be represented by using an extension of UML. We expose our proposal for representing architectural design in the Tropos methodology using this extension of UML.

3. Architectural Representation in UML

The UMLRT [9],[10] is using UML as an architectural modeling language. Some specific architectural modeling concepts are defined as specializations of generic UML concepts. These specializations, usually expressed as stereotypes, conform to the generic semantics of the corresponding UML concepts, but provide additional semantics specified by constraints [9]:

- Capsules: A capsule is a stereotype of the UML class concept with some specific features. A capsule uses its ports for all interactions with its environment. The communication with others capsule is done by one or more ports. The interconnection with other capsules is via connectors using signals. A capsule is a specialized active class and is used for modeling a self contained component

of a system. For instance, a capsule may be used to capture an entire subsystem, or even a complete system.

- Ports: A port represents an interaction point between a capsule and its environment. They convey signals between the environment and the capsule. The type of signals and the order in which they may appear is defined by the protocol associated with the port. The port notation is shown as a small hollow square symbol. If the port symbol is placed overlapping the boundary of the rectangle symbol denotes a public visibility. If the port is shown inside the rectangle symbol, then the port is hidden and its visibility is private. When viewed from within the capsule, ports can be of two kinds: relay ports and end ports. Relay ports are ports that simply pass all signals through and end ports are the ultimate sources and sinks of all signals sent by capsules. These signals are generated by the state machines of capsules (Figure 8).

- Protocols: A protocol specifies a set of valid behaviors (signal exchanges) between two or more collaborating capsules. However, to make such a dynamic pattern reusable, protocols are decoupled from a particular context of collaborating capsules and are defined instead in terms of abstract entities called protocol roles (stereotype of Classifier Role in UML) (Figure 9).

- Connectors: A connector is an abstraction of a message-passing channel that connects two or more ports. Each connector is typed by a protocol that defines the possible interactions that can take place across that connector (Figure 8).

4. Organizational Architectural Styles In UML

The organizational styles are generic structures defined at a metalevel that can be instantiated to design a specific application architecture. They support non-functional requirements, represented in Tropos methodology such as softgoals, during architectural design phase. Unlike functional requirements which define what a software is expected to do, non-functional requirements specify global constraints on how the software operates or how the functionality is exhibited. NFRs are as important as the functional ones. They are not simply desired quality properties, but critical aspects of dynamic systems without which the applications cannot work and evolve properly. The need to treat non-functional properties explicitly is a critical issue when software architecture is built. Organizational architectures integrate NFR with architectural project, since NFRs are composing part of these styles.

Tropos relies on the i^* notation [4] to describe both requirements and represent organizational architectural styles. Unfortunately, this notation is not widely accepted

by software practitioners, since it is just beginning to be recognized as a suitable notation for representing requirements and its tool support is also limited. On the other hand, the Unified Modeling Language [3] has been used to represent the architecture of simple and complex systems. Using UML as an Architecture Design Language in the Tropos methodology allow us for representing detailed information which sometimes is required in architectural design, such as set of signals that are exchanged between architectural components, which are not supported by the *i** notation. In the sequel we explain how the concepts of Tropos can be accommodated within UML-RT, in order to represent organizational architectures in UML.

As explained in section 2.1, in Tropos actors are active entities that carries out actions to achieve goals by exercising their know-how. In section 3.1, we explained that in UML-RT, capsules are specialized active classes used for modeling self contained components of a system. Hence, an actor in Tropos is mapped to a capsule in UML-RT (Figure 2). Note that ports are physical parts of the implementation of a capsule that mediate the interaction of the capsule with the outside world.

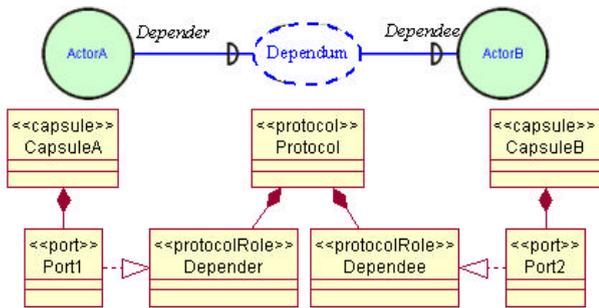


Figure 2. Mapping a dependency between actors to UML

In Tropos a dependency describes an “agreement” (called *dependum*) between two actors playing the roles of *dependor* and *dependee*, respectively. The *dependor* is the depending actor, and the *dependee*, the actor who is depended upon. Dependencies have the form *dependor*⇒*dependum*⇒*dependee*. In UML-RT, a protocol is an explicit specification of the contractual agreement between its participants, which plays specific roles in the protocol. In other words, a protocol captures the contractual obligations that exist between capsules. Hence, a *dependum* is mapped to a protocol and the roles of *dependor* and *dependee* are mapped to protocol roles that are comprised by the protocol (Figure 2).

The type of the dependency between two actors (called *dependum*) describes the nature of the agreement. Tropos defines four types of *dependums*: goals, softgoals,

tasks and resources. Each type of *dependum* will define different features in the protocol and therefore in ports that realizes its protocol roles. As noted earlier, protocols are defined in terms of entities called *protocol roles*. Since *protocol roles* are abstract classes and ports play a specific role in some protocol, a *protocol role* defines the *type* of a port, which simply means that the port implements the behavior specified by that *protocol role*. As defined earlier, capsules are complex, physical, possibly distributed architectural objects that interact with their surroundings through ports. Note that a port is both a composite part of the structure of the capsule and a constraint on its behavior.

Goal type will be mapped to an attribute with boolean type present into the port that realizes the protocolRole *dependee* (Figure 3). It represents a goal that a capsule is responsible for fulfill by exchanging the signals defined in the protocolRole *dependee*.

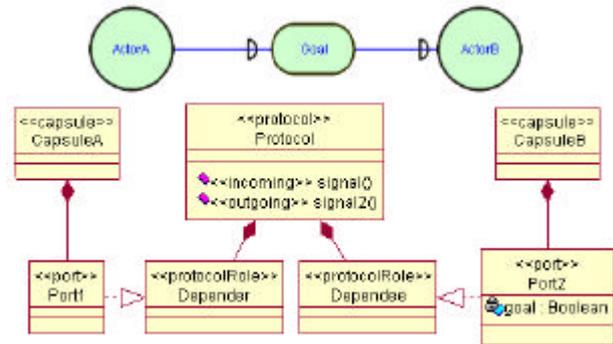


Figure 3. Mapping a goal dependency to UML

Softgoal type is mapped to an attribute with enumerated type present into the port that realizes the protocolRole *dependee* (Figure 4). It represents a quality goal that a capsule is responsible for fulfill to a given extent by exchanging the signals defined in the protocolRole *dependee*.

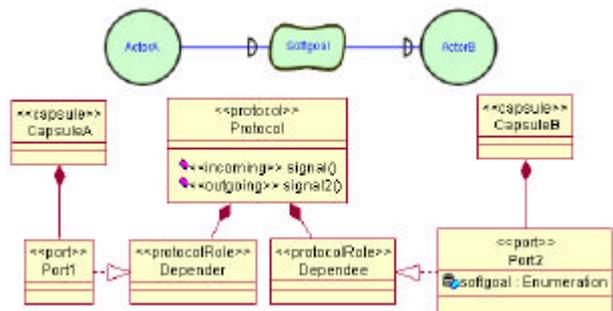


Figure 4. Mapping a softgoal dependency to UML

Resource type is mapped to the return type of an abstract method placed on protocolRole *dependee* that will be realized by a port of a capsule (Figure 5). This return type represents a resource that a capsule is required to provide by exchanging signals defined in the protocolRole *dependee*.

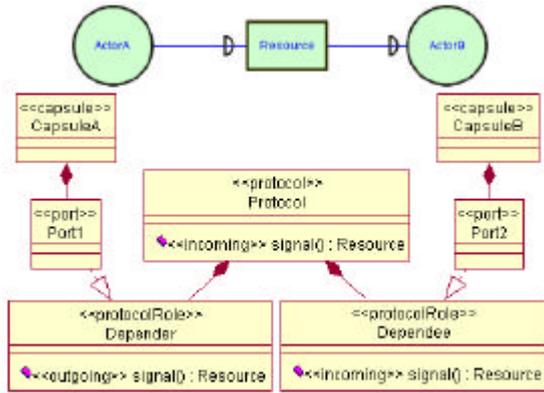


Figure 5. Mapping a resource dependency to UML

Task type is mapped to an abstract method placed on protocolRole *dependee* that will be realized by a port of a capsule (Figure 6). It represents an activity that a capsule is required to perform by exchanging signals defined in the protocolRole *dependee*.

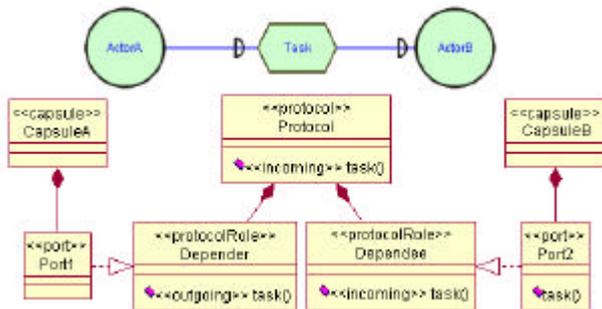


Figure 6. Mapping a task dependency to UML

A more compact form for describing capsules is illustrated in Figure 7, where the ports of a capsule are listed in a special labeled list. The protocol role (type) of a port is normally identified by a pathname since protocol role names are unique only within the scope of a given protocol. However, ports are also depicted in the collaboration diagrams (Figure 8) that describe the internal decomposition of a capsule. In these diagrams, ports are represented by the appropriate classifier roles, i.e., the *port roles*. To reduce visual clutter, port roles are generally shown in iconified form. For the case of binary protocols, an additional stereotype icon can be used: the port playing the conjugate role (*dependee* role) is

indicated by a white-filled (versus black-filled) square. In that case, the protocol name and the tilde suffix are sufficient to identify the protocol role as the conjugate role; the protocol role name is redundant and should be omitted. Similarly, the use of the protocol name alone on a black square indicates the base role (*dependee* role) of the protocol. In Figure 8, we can see the details of (inside) the capsule and the end port/relay port distinction is indicated graphically.



Figure 7. A capsule class diagram

In UML-RT, each connector is typed by a protocol that specifies the *desired* behavior that can take place over that connector. A key feature of connectors is that they can only interconnect ports that play complementary roles in the protocol associated with the connector. In a class diagram, a connector is modeled by an association while in a capsule collaboration diagram it is declared through an association role. Hence, a dependency (*dependee* ⇒ *dependum* ⇒ *dependee*) in Tropos is mapped to a connector in UML-RT (Figure 7 and Figure 8). In the sequel we show how the Joint Venture organizational architectural style (Figure 1) is modeled using UML-RT.

4.1. Joint Venture In UML

The UML-RT notation of capsules, ports and connectors is used to model the architectural actors and their dependencies. In Figure 8, each capsule is representing an actor of the joint venture architecture. When an actor is a *dependee* of some dependency, its corresponding capsule has an implementation port (end port) for each dependency (ex. Port1), which is used to provide services for others capsules. When an actor is a *dependee* of some dependency, its corresponding capsule has an implementation port (relay port) to exchange messages (ex. Port3).

The Joint Venture architectural style presents six capsules disposed according to Figure 8. The capsule Joint Management is responsible for ensuring the strategic operation and coordination of such a system and its partner capsules on a global dimension. Through the delegation of authority it coordinates tasks and manages sharing of knowledge and resources. The two secondary partners are capsules responsible for supplying services or for supporting tasks for the organization core. The three principal partners are capsules responsible for managing and controlling themselves on a local

dimension. They can interact directly with other principal partners to exchange, provide and receive services, data and knowledge.

From Figure 1 you can recall the goal dependency *Authority Delegation* between *Principal Partner_n* and *Joint Management* actors. Each actor present in Figure 1 is mapped to a capsule in Figure 8. Each *dependum*, i.e., the “agreement” between these two actors is mapped to the protocol (see Figure 9). A protocol is an explicit specification of the contractual agreement between the participants in the protocol. In our study these participants are the two actors previously mapped to capsules. Each dependency is mapped to a connector in Figure 8. Each connector is typed by the protocol that represents the *dependum* of its corresponding dependency. The type of the dependency describes the nature of the agreement, i.e., the connector type describes the nature of the protocol. The four types of *dependums* (Goal, Softgoal, Task and Resource) are mapped to four types of protocols (Figures 9, 10, 11 and 12).

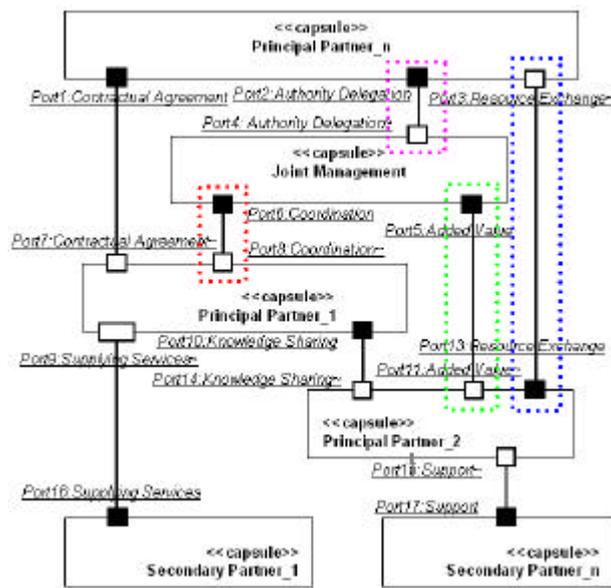


Figure 8. Joint Venture Style in UML-RT’s capsule collaboration diagram

For example, in the Goal type, the protocol *Authority Delegation* (Figure 9) assures that this goal will be fulfilled by using the signals described in the protocolRole *dependee*. The goal will be mapped to a boolean attribute present in the port that implements the protocolRole *dependee*. This attribute will be true if the goal has been fulfilled and false otherwise. Hence, in the dependency between *Principal Partner_n* and *Joint Management* capsules depicted in the second dotted area of Figure 8, the goal dependency will be mapped to a boolean attribute

located in the port which composes the capsule *Principal Partner_n* and implements the protocolRole *dependee* of the protocol that assures the fulfillment of this goal (Figure 9).

Now examine the softgoal dependency *Added Value* between *Principal Partner₂* and *Joint Management* actors depicted in Figure 1. In this case, the protocol *Added Value* (Figure 10) assures that this softgoal will be satisfied in some extent by using the signals described in the protocolRole *dependee*. The softgoal will be mapped to an enumerated attribute present in the port that implements the protocolRole *dependee*. This attribute will represent different degrees of softgoal fulfillment.

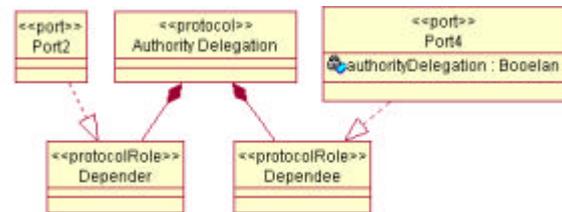


Figure 9. Protocols and Ports representing the Joint Venture’s goal dependency Authority Delegation

Hence, in the dependency between *Principal Partner₂* and *Joint Management* capsules depicted in the third dotted area of Figure 8, the softgoal dependency will be mapped to an enumerated attribute located in the port which composes the *Joint Management* capsule and implements the protocolRole *dependee* of the protocol that assures some degree of fulfillment of this softgoal (Figure 10).

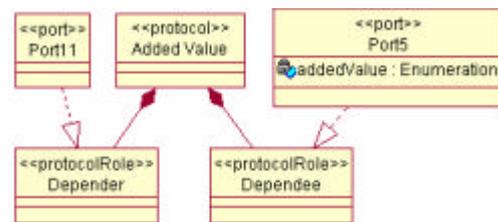


Figure 10. Protocols and Ports representing the Joint Venture’s softgoal dependency Added Value

In the sequence, look at the task dependency *Coordination* between *Principal Partner₁* and *Joint Management* actors depicted in the Figure 1. Here, the protocol *Coordination* (Figure 11) assures that this task will be performed by using the signals described in the protocolRole *dependee*. The task itself will be mapped to a <<incoming>> signal in the protocolRole *dependee* and the port that implements that protocolRole will be

committed to realize their signals. Hence, in the dependency between *Principal Partner_1* and *Joint Management* capsules depicted in the first dotted area of Figure 8, the task dependency will be mapped to a <<incoming>> signal placed in the protocolRole *dependee* of the protocol that assures the performing of this task. The *Joint Management* capsule is composed by a port which implements this protocolRole *dependee* (Figure 11).

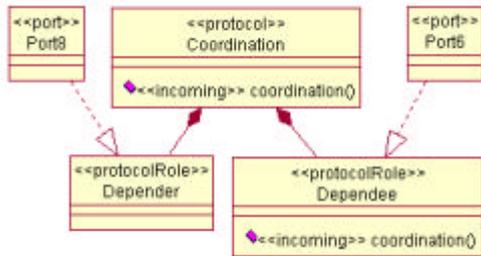


Figure 11. Protocols and Ports representing the Joint Venture’s task dependency Coordination

Finally we have the resource dependency *Resource Exchange* between *Principal Partner_2* and *Principal Partner_n* depicted in the Figure 1. Again, the protocol *Resource Exchange* (Figure 12) assures that this resource will be provided by using the signals described as <<incoming>> signals in the protocolRole *dependee*. The resource will be mapped to a <<incoming>> signal that returns an information of type resource in the protocolRole *dependee* and the port that implements that protocolRole will be committed to realize their signals.

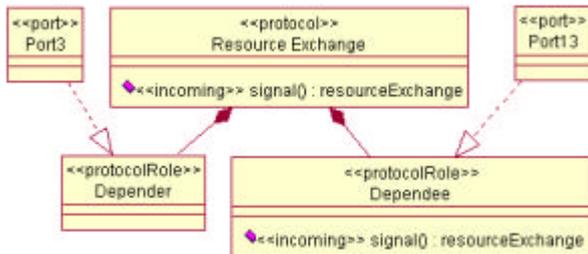


Figure 12. Protocols and Ports representing the Joint Venture’s resource dependency Resource Exchange

Hence, in the dependency between *Principal Partner_2* and *Principal Partner_n* capsules depicted in the fourth dotted area of Figure 8, the resource dependency will be mapped to an <<incoming>> signal that returns an information of type resource and is placed in the protocolRole *dependee* of the protocol that assures the providing of this resource. The *Principal Partner_2*

capsule is composed by a port which implements this protocolRole *dependee* (Figure 12).

Although we have only detailed the mapping of four dependencies in the Joint Venture Style to their respective representation in UML-RT, the remaining ones are mapped analogously, according to their types.

6. Case Study

We extracted a case study from [1] that describes a business organization selling media items (books, newspapers, CDs, etc.) that has decided to open up a B2C retail sales front on the internet named Medi@.

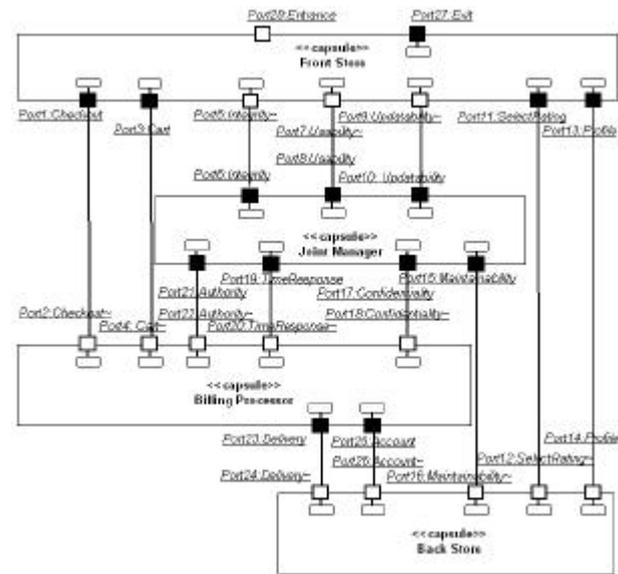


Figure 13– Media@ system architecture

Based on the joint venture architectural style, Figure 13 suggests a possible assignment of system responsibilities. *Front Store* primarily interacts with *Customer* and provides her with a usable front-end web application. Moreover, it is responsible for catalogue browsing, items search in database and supplying on-line customers with information about media items. *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 the controlling security gaps, availability bottlenecks and adaptability issues, in order to ensure the software non-functional requirements. All four capsules need communicate and collaborate each other in the running system.

Observe that the message exchange between capsules happens in the context defined by protocol implemented by ports that compose each capsule involved in the interaction. For example, the communication protocol in Figure 15 shows a request from Back Store to Front Store for producing the Customer Profile.

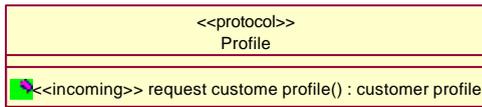


Figure 15. Profile Communication protocol between Front Store and Back Store capsules

Moreover, we can use sequence diagrams to depict the interaction between the capsules which compose the system when realizing a particular scenario: the request for ordering a media item.

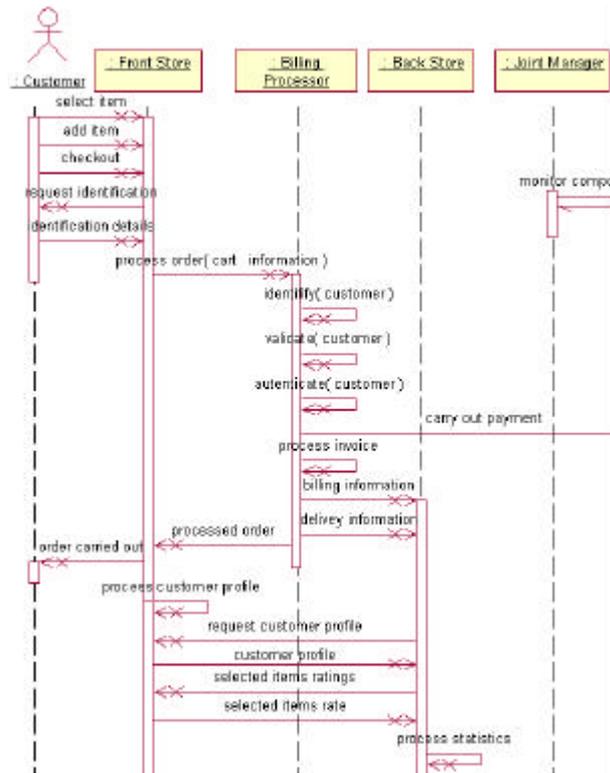


Figure 14. Sequence diagram for Ordering Media Item context

Using UML-RT capsules enable us to refine the system architecture to lower-level components (sub-capsules) which depend on each other to realize the whole system responsibilities. Sequence diagrams insert details in architectural behaviour, since it shows the exchanged

signals in the interactions, as well as the valid sequence of these signals (communication protocol between capsules).

7. Conclusions and Future Work

In this work, we have been proposed using UML Real-Time to accommodate the concepts and features used for representing organizational architectures in Tropos, nowadays. This proposal has been applied to multi-agent software system development for an e-commerce application. In this paper, we outline an organizational architecture in UML. Our approach is appropriate for:

- Obtaining an architectural model closer to organizational environment where the system will eventually operate, mitigating the existent semantic gap between the software system and its running environment.
- Modeling more detailed architectures both in structural and behavioural aspects.
- Building a flexible architecture with loosely coupled components, which can evolve and change continually to accommodate new components and meet new requirements, as well as support non-functional requirements. Hence, it enables to realize stakeholders' demand for more flexible and complex systems.
- Being able to use UML elements to represent non-UML artifacts enables us to use existing UML toolsets to create those views.
- Making organizational architectures styles widely used in industry, namely by other agent-oriented methodologies or those tuned to open, cooperative, dynamic and distributed systems.

In Tropos, UML is used only in detailed design phase. However using UML-RT for modeling architecture can help Tropos in the following issues:

- Common Representation Model: Modeling information of different types of views (UML and non-UML) can be physically stored in the same repository.
- Unified Way of Cross-Referencing Model Information: Having modeling information stored at one physical location further enables us to cross-reference that information. Cross-referencing is useful for maintaining the traceability among artifacts from architectural design and detailed design phases in Tropos.

To improve this proposal, future work is required to provide systematic guidelines. Currently this processes happens in a ad hoc way based on software engineer experience. Proper guidance will enable us to create instances from architectural metamodels, defined by

Tropos, from requirement models represented in i* notation. Also we intend to model internal behaviour of capsules with state diagram. Moreover, we aim at proposing UML extensions for representing social patterns involving agents, as well as both the structural and behavioural aspects and features defining such a software agents, in the context of Tropos Methodology.

7. References

- [1] Castro, J., Kolp, M., Mylopoulos, J.: Towards Requirements-Driven Information Systems Engineering: The Tropos Project. In *Information Systems*, Vol. 27. Elsevier, Amsterdam, The Netherlands (2002) 365–389
- [2] Mylopoulos, J., Kolp, M., Castro, J.: UML for Agent-Oriented Software Development: the Tropos Proposal. In *Proceedings of the Fourth International Conference on the Unified Modeling Language (<<UML>> 2001)*. Toronto, Canada (2001), LNCS 2185, p.p. 422-423
- [3] Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language – Reference Manual*. Addison Wesley (1999)
- [4] Yu., E.: *Modelling Strategic Relationships for Process Reengineering*. Ph.D. thesis, Department of Computer Science, University of Toronto, Canada (1995)
- [5] Chung, L., Nixon, B. A., Yu, E., Mylopoulos, J.: *Non-Functional Requirements in Software Engineering*. Kluwer Publishing (2000)
- [6] Kolp, M., Castro, J., Mylopoulos, J.: A social organization perspective on software architectures. In *Proc. of the 1st Int. Workshop From Software Requirements to Architectures. STRAW'01*, Toronto, Canada (2001) 5–12
- [7] Kolp, M., Giorgini, P., Mylopoulos, J.: A goal-based organizational perspective on multi-agents architectures. In *Proc. of the 8th Int. Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages. ATAL'01*, Seattle, USA (2001)
- [8] Kolp, M., Mylopoulos, J.: Software architectures as organizational structures. In *Proc. ASERC Workshop on "The Role of Software Architectures in the Construction, Evolution, and Reuse of Software Systems"*, Edmonton, Canada (2001)
- [9] Selic, B., Rumbaugh, J.: *Using UML for Modeling Complex Real-Time Systems*. Rational Whitepaper (www.rational.com) (1998)
- [10] OMG: *Unified Modeling Language 2.0*. Initial submission to OMG RFP ad/00-09-01 (UML 2.0 Infrastructure RFP) and ad/00-09-02 (UML 2.0 Superstructure RFP): Proposal version 0.63 (draft). <http://www.omg.org/>.
- [11] Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, N.J., Prentice Hall (1996)
- [12] Conallen, J.: *Building Web Applications with UML*. Addison-Wesley (2000)
- [13] IBM: *Patterns for e-business*. At <http://www.ibm.com/developerworks/patterns> (2001)
- [14] Silva, C. T. L. L., Castro, J. F. B.: *Detailing Architectural Design in the Tropos Methodology. CAISE03 - The 15Th Conference on Advanced Information Systems Engineering*, 2003, Klagenfurt/Velden, Austria (to appear).

Eliciting Architectural Decisions from Requirements using a Rule-based Framework

WenQian Liu Steve Easterbrook
Department of Computer Science
University of Toronto
Toronto, ON, M5S 3G4, Canada
{wl,sme}@cs.utoronto.ca

Abstract

Making architectural decisions based on requirements, analyzing cost-benefit trade-offs, and keeping design options open is a difficult task. Existing work on classification of architectural styles and features of reusable components, and derivation of relevant architectural styles provides useful heuristics to the task, but it remains to be largely a labor-intensive activity.

In this paper, we propose a rule-based framework with automated reasoning for eliciting architectural decisions from requirements. Our goal is to gain a deeper understanding of the relationships between requirements and architectural decisions, define generic mappings based on these relationships, and use these mappings to guide architectural design with a higher degree of automation.

Keywords

Architecture, requirements, mapping, decision elicitation, design guidance, rule-base

1 Introduction

It has been recognized by the research community that building a systematic bridge between requirements and software architecture plays an important role in software engineering [1]. In particular, making architectural decisions based on requirements, analyzing cost-benefit trade-offs, and keeping design options open remains to be a labor-intensive task. A number of approaches have made progress towards providing assistance to software architects.

Early works include Shaw and Clements' classification of architectural styles that had appeared in the published literature [8]. Each style is categorized according to its characteristics with respect to constituent parts (components and

connectors), control issues, data issues, control and data interaction issues, and reasoning. Moreover, intuition and rules of thumb on choosing styles to fit the problem are discussed as a preliminary step to design guidance.

Around the same time, Kazman, Clements, and Bass provided a classification on architectural elements in terms of features, which can be used to identify reusable elements that match required feature criteria [5]. In their approach, temporal and static features are defined for classifying architectural elements and describing the matching criteria of requirements.

More recently, Egyed et al. addressed this problem using the CBSP (Component-Bus-System, and Properties) approach [4]. In this work, the WinWin negotiation model [2] is adapted to classify the requirements according to the CBSP properties in the architectural context. Based on these properties, a CBSP model can be built to derive and validate architectural styles.

There are five characteristics in common in these approaches:

1. classification of requirements and architectural properties
2. definition of a partial mapping from requirements properties to architectural elements or decisions using a common language
3. provision of design alternatives and trade-off analysis
4. abstraction of information
5. reuse through styles by condition matching

Despite these advances, a number of key issues in bridging the gap between requirements and software architecture are not well addressed to date.

Unified description language In order to bridge the gap between requirements and architecture, we need to define mappings between them. To establish these mappings, requirement specifications and architectural descriptions must be formulated in a common language. This motivates the development of a unified language. We have seen that this is done implicitly in the above approaches. But we do not yet know the following.

- How feasible is it to use a unified language?
- How to express requirements and architectural descriptions effectively using a unified language?
- What are the key properties of such a language?

Relationship between requirements and architectural decisions

- What kind of architectural decisions are frequently made in building large systems?
- Clearly, the architectural decisions made are related to the benefits and risks that are induced. Are we able to define relationships between them in assisting the trade-off analysis?
- How do architectural decisions relate to the system's requirements?
- Are we able to classify relations between requirements and architectural decisions that are generic and reusable?
- How do we abstract key architectural decisions made in existing systems?

Studying decision making processes in existing systems may provide insight into general relationship between requirements and architectural decisions.

Architectural decisions deferral and trade-off In practice, it is often necessary to defer an architectural decision until further information is acquired and to keep design options open. Therefore, it is undesirable to make every decision up front and have little flexibility in making changes. However, having too many open ends will make decision making difficult and prevent the development progress. This leads to the questions below. Answers to these questions can help analyzing the trade-offs between different architectural decisions, and project architectural evolutions with changing requirements.

- At what stage must these decisions be made before proceeding further? How much can they be deferred?

- To what extent does architectural evaluation help in choosing the best solutions for deferred decisions?
- To what extent do architectural decisions precede and shape identification of requirements?
- Are there any common factors for deferring decisions? Do they relate to specific classes of requirements?

The earlier work has provided insights to the questions posed above, but answers to many of them remain unknown. In particular, answers to the question of what are the generic and reusable mappings between requirements, architectural properties, and decisions can lead to significant progress. Our research is mostly motivated by these questions. In answering these questions, we could gain a deeper understanding of the relationships between requirements and architectural properties, define generic mappings based on the relationships, and use the mappings to guide the architectural design with a higher degree of automation.

In this paper, we propose a framework that can be used to elicit architectural decisions from requirements, and describe a potential rule-based implementation with automated reasoning capability. Although user interaction is required in this framework, we believe it is a worthwhile experiment. Our reasons are the following. Firstly, the framework can be customized for any application domain, and the rule-base can be easily updated as new mappings are required. Secondly, existing architectural decision making knowledge can be evaluated using this framework. Thirdly, the evaluation of knowledge can help us define the relationships between requirements and architectural properties. Lastly, this framework can be extended with higher degree of automation once the reasoning system covers enough decision making strategies.

We plan to build a rule-based tool to capture the mappings, and in the process of doing so, to study how decisions are made, what is the essential knowledge required, and the structure of the knowledge. Our prior experience [6, 7] shows that attempting to develop a rule-based (or production) system raises useful questions about what knowledge and heuristics to apply and how they interrelate.

In section 2, we describe our proposal of a general design guidance framework for eliciting architectural decisions. In section 3, we outline a rule-based implementation of the framework. In section 4, we present concluding remarks.

2 Architectural Decision Elicitation Framework

Requirements need to be obtained from stakeholders. Likewise, architectural decisions need to be elicited from requirements. Even though a large body of research results and practical heuristics is available for making architectural

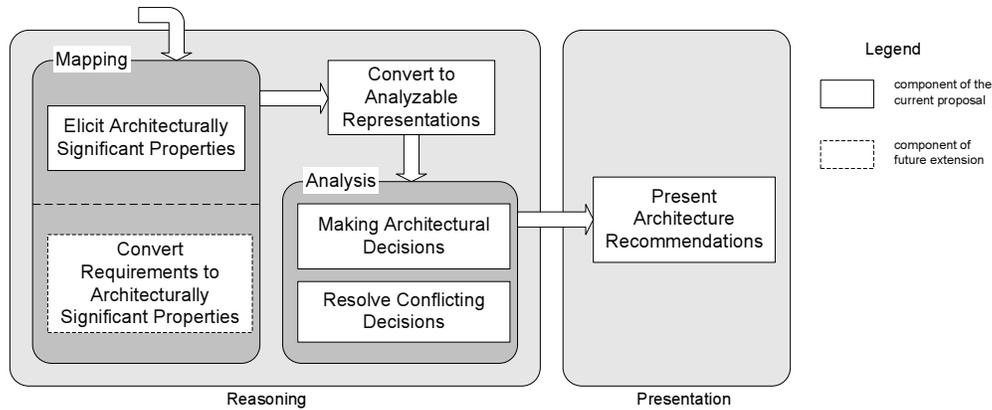


Figure 1. The Architectural Decision Elicitation Framework

decisions, architects still need to carefully go through their knowledge-base (usually their experience) to identify relevant information, and analyze cost-benefit trade-offs, before making a decision. We propose an architectural decision elicitation framework (ADEF) that encapsulates the knowledge required of making architectural decisions, and provides automated mapping from architecturally significant properties to architectural decisions. This framework adapts a general Waterfall model.

There are two main modules, *Reasoning* and *Presentation* in ADEF, as shown in figure 1.

The **Reasoning** module encapsulates the decision making knowledge, and reasons about the requirements to elicit relevant architectural decisions. This module consists three parts: *mapping*, *conversion*, and *analysis*.

The **Mapping** submodule uses built-in decision trees (directed acyclic graphs) to provide guidance to the user in manually mapping each requirement specification to one or more architecturally significant properties. Figure 2 illustrates an example of a partial decision tree with only the properties that are significant in choosing architectural styles (the ideas in this example are adapted from [4, 8]). We use *decision node* to refer to both interior node and leaf node in the decision tree, and *property node* to refer to leaf node only.

Here is how the mapping is achieved. For each requirement specified, starting at the root of the decision tree, present the user with the choices represented by the decision nodes associated (i.e. immediately below and connected) to the root, and ask the user to decide whether each choice is relevant to the current requirement. For each relevant decision node chosen by the user, its associated decision nodes are then presented to the user in a depth-first fashion until no more nodes are available as a choice. The descriptions of the *property nodes* chosen by the user are the architecturally significant properties. Such a property and its deci-

sion making history, including the nodes along the branch and the source requirement, are described as a *decision unit* and sent to the conversion submodule.

A future extension to this module is the automated mapping from requirements to architecturally significant properties shown as the dotted box in figure 1. This extension dictates that the requirements be stated in a formal language.

The **Convert to Analyzable Representation** (conversion) submodule converts the *decision units* created in the mapping module to a form that can be interpreted by the analysis module. The converted decision units are then sent to the analysis module as new facts.

The **Analysis** submodule provides ongoing automated reasoning of the following types using a predefined knowledge-base:

1. *making architectural decisions*: based on change in the fact base (either a newly converted decision unit, or a newly made decision), make appropriate architectural decisions using heuristics defined in the knowledge-base, then store the decisions as new facts
2. *resolve conflicting decisions*: provide resolution and explanation when multiple conflicting decisions are made for the same part of the system

The **Presentation** module presents the resulting architectural decisions to the user and updates changes made to previous results. The process then is repeated for another requirement specification.

Next, we describe a rule-based implementation proposal for this framework.

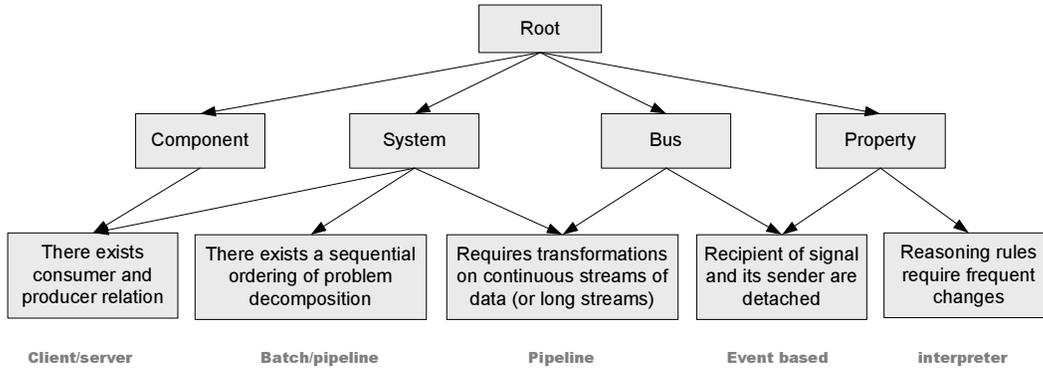


Figure 2. An Example Decision Tree

3 A Rule-based Implementation

In section 2, we have seen that the knowledge-base for the *mapping submodule* is implemented in a decision tree. In this section, we describe a rule-based approach to achieve the automated reasoning capability provided by the *analysis submodule* using a production system.

A production system keeps the fact knowledge-base (or fact base) separate from the rule base. Rules can be defined a priori and maintained independently in the rule base. The fact base is updated during the system execution. Not only can new facts be added, but also the results made by each reasoning cycle are fed back to the fact base as updates. Ongoing reasoning is performed whenever updates to the fact base are received. These characteristics of production system help to achieve dynamic analysis required of the framework. In addition, our prior experience has shown positive results in applying the rule-based approach for automated reasoning [6, 7]. Thus, we believe it is viable to use a rule-based implementation for the *analysis submodule*.

We first give a brief overview of production systems, then discuss the rule-based implementation for the *analysis submodule*.

3.1 Production System Overview

A *production system* is a reasoning system that uses forward-chaining derivation techniques. It uses rules, called *production rules* or *productions* in short, to represent its general knowledge, and keeps an active memory, known as the *working memory* (WM), of facts (or assertions) which are called *working memory elements* (WMEs) [3].

A *production rule* is usually written as:

```
IF conditions THEN actions
```

The *conditions*, also known as *patterns*, are partial descriptions of working memory elements, which will be

tested against the current state of the working memory. For example, the following rule debits a bank account.

```

IF      (transaction
        (type debit)
        (amt ?x)
        (account ?a))
        (account
        (id ?a)
        (balance ?y ∧ {≥?x}))
THEN   REMOVE 1
        MODIFY 2 (balance [?y-?x])
  
```

where $?a$, $?x$ and $?y$ are variables; $\{≥?x\}$ is a test for $balance ≥ x$; REMOVE 1 deletes the first (i.e. transaction) WME from the working memory; and MODIFY 2 selects the second WME and assigns the value of $y-x$ to balance.

Each condition can be either positive or negative. A negative condition is of the form $-cond$, where $cond$ represents a positive condition. A rule is applicable if all of the variables can be evaluated using the WMEs in the current WM such that the conditions are met. A positive condition is satisfied if there is a matching WME in the WM; a negative condition is satisfied if there is no matching WME in the WM.

A *working memory element* has the following form, $(type (attribute_1 value_1) \dots (attribute_n value_n))$ where $type$ and $attribute_i$ are atoms, i.e. a string, a word, or a numeral; and $value_i$ is an atom or a list.

The basic operation of a production system is a cyclic application of three steps until no more rules can be applied:

1. *recognize*: identify applicable rules whose conditions are satisfied by the WM;
2. *resolve conflict*: among all applicable rules (or *conflict set*), choose one to execute;
3. *act*: apply the action given in the consequent of the executed rule.

3.2 Rule-based Analysis

In the *analysis submodule*, there are two goals to achieve: architectural decision making, and conflicting decision resolution. We use production rules to capture the knowledge and strategies for these goals. The challenges here are to identify commonly used architectural decisions and the architectural properties required for making these decisions, choose an effective and concise representation scheme for facts, identify conflict conditions and resolutions, and design rules to reflect these properties.

To illustrate how a decision making rule is defined and executed, we use the example decision tree shown in figure 2. In this decision tree, the property nodes are closely related to some well known architectural styles. We use this information to design the rules. For example, taking the first property node from the left, we can characterize it with a rule of the following form:

```
IF      (there exists consumer and producer
        relation)
THEN ADD (use client/server style)
```

However, this rule does not capture key decision nodes along the branch and the source requirement. In addition, the representation used in the rule is not concise. We refine the rule to be the following:

```
IF      (property
        (uid ?id)
        (relation consumer-producer)
        (source ?reqID)
        (concern ?part))
THEN ADD (style
        (name client-server)
        (property (uid ?id)))
```

where the variable *?id* is matched to the unique identifier of the decision unit (denoted as *property*); *?reqID* is matched to the unique source requirement identifier; and *?part* is matched to the part that the decision unit is concerned about at the top level of the decision tree: *component*, *system*, *bus*, or *property*.

When this rule matches a WME and is executed by the production system, the client-server style WME is then added to the fact base as a recommended architectural style to use for the part concerned.

Note that this rule is defined in close relation to what is in the sample decision tree. This is indeed the case when designing rules. They must provide means to match the decision units to architectural decisions.

When two decisions made by the rules are associated with the same part of concern, and cannot be used together in the architecture consistently, a conflict is identified. The conflicting decision resolution rules are designed to capture such conditions and provide solutions thereto.

4 Conclusions

In this paper, we have posed many open questions for bridging the gap between requirements and software architecture. Our research is motivated by these questions. In particular, we are interested in exploring the applicability of a unified description language for requirement specifications and architecturally significant properties, classifying architectural knowledge for building decision trees and production rules for requirement mapping and analysis, identifying the relationships between requirements, architectural decisions and properties.

We have proposed a framework to provide design guidance in eliciting architectural decisions from requirements, and a rule-based implementation. Although human interaction is required to map requirements to architecturally significant properties in the framework, we believe that using a tool implementing the framework can help us evaluate existing architectural decision-making knowledge, and define the relationships between requirements, architectural decisions and properties. Understanding of such relationships can help us to provide higher degree of automation and minimize human involvement.

References

- [1] In J. Castro and J. Kramer, editors, “*The First International Workshop on From Software Requirements to Architectures (STRAW’01)*”. At the 23rd International Conference on Software Engineering, Toronto, 2001.
- [2] B. W. Boehm and R. Ross. “Theory W Software Project Management: Principles and Examples”. *IEEE Transactions on Software Engineering*, pages 902–916, 1989.
- [3] R. J. Brachman and H. J. Levesque. “Knowledge Representation and Reasoning”. In preparation, 2001.
- [4] A. Egyed, P. Grünbacher, and N. Medvidovic. “Refinement and Evolution Issues in Bridging Requirements and Architecture - The CBSP Approach”. In J. Castro and J. Kramer, editors, “*The First International Workshop on From Software Requirements to Architectures (STRAW’01)*”, pages 42–47. At the 23rd International Conference on Software Engineering, Toronto, 2001.
- [5] R. Kazman, P. Clements, G. Abowd, and L. Bass. “Classifying Architectural Elements as a Foundation for Mechanism Matching”. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.
- [6] W. Liu. “Rule-based Detection of Inconsistency in Software Design”. Master’s thesis, University of Toronto, Department of Computer Science, July 2002.
- [7] W. Liu, S. M. Easterbrook, and J. Mylopoulos. “Rule-Based Detection of Inconsistency in UML Models”. In L. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar, editors, *Workshop on Consistency Problems in UML-Based Software Development*, pages 106–123. At the Fifth International Conference on the Unified Modeling Language, Dresden, Germany, 2002.

- [8] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems, 1996.

Requirements, Architectures and Risks

James D. Kiper
*Dept. of Computer Science and Systems
Analysis
Miami University
Oxford, OH 45056
kiperd@muohio.edu*

Martin S. Feather
*Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr
Pasadena CA 91109-8099
Martin.S.Feather@Jpl.Nasa.Gov*

Abstract

There is wide agreement that architecture plays a prominent role in large, complex software systems. Selection of an appropriate architecture – one that matches the system requirements and implementation resources – is a critically important development step.

We advocate the use of risk-based reasoning to help make good architectural decisions. In this paper, we explore the adaptation of a risk management process and tool to this purpose.

1. Introduction

Software design for complex software systems is difficult. The past decade has seen a convergence of opinion about the importance of using established architectures and design patterns. At the system level, styles of software architecture [1, 11] like pipes-and-filters or event-driven provide a starting point for design of complex software systems. At the more detailed level, architectural treatments capture well-reasoned decisions whose strengths and weakness are understood, e.g., software design patterns like wrapper or builder. [7] This paper will focus on the system level use of architecture, although the approach should also be applicable to the finer grained use of design patterns.

Choosing a *good* architecture is a critically important step in the design of a system. A poor choice at this level is difficult to repair at a more detailed design level. We define the adjective *good* with respect to architecture to mean an architecture that matches system requirements *and* can be implemented within the resources allocated to it. The implementation itself is a non-trivial task, and induces a further set of critical decisions.

The primary thesis of this paper is that risk can be used to guide these decisions. Use of risk-based reasoning enables software engineers and managers to make choices of software architecture and architecture implementation that satisfy both criteria – meeting system requirements and adhering to resource limitations.

This paper is organized as follows: section 2 describes the current risk-based design process and the tool that has

been developed to support this process; section 3 discusses some shortcomings in this process that are caused by the failure to capture of explicit design and, in particular, architectural aspects; section 4 describes ways in which we are incorporating software architectural decisions into this process and tool.

2. Basis for the approach - risk-based design

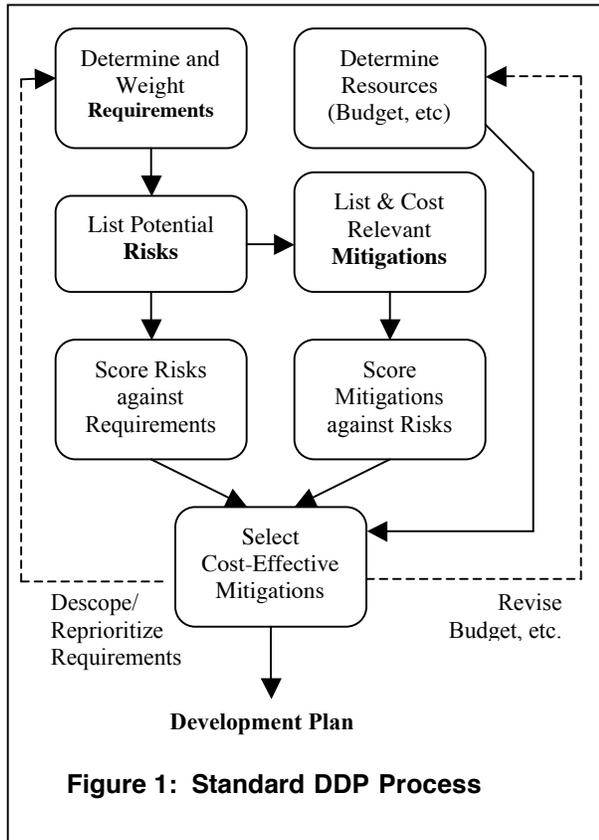
The approach advocated herein begins from an existing risk-based design process and its accompanying tool support. This is the “Defect Detection and Prevention (DDP)” process [4], developed and used at JPL to help engineers manage the trade space of choices in designing spacecraft and associated technology.

DDP has three primary sets of issues that it captures and tracks: requirements, risks, and mitigations. The DDP tool is typically used to collect and maintain decisions and information discussed in several meetings with a group of experienced engineers and domain experts. The process used in these DDP sessions is diagrammatically explained in figure 1. The first step is the collection and weighting of requirements. Given the requirements, the domain experts determine the risks that these system requirements entail. Each of these risks is then scored as to its impact on each of the requirements. After risks are determined in step 2, the activities that can mitigate these risks are then listed. Each of these mitigations is scored as to its effectiveness at reducing each risk.

DDP is unique in bringing a quantitative risk-based approach to bear at early stages of decision-making. The scoring of the links between risks and requirements, and between mitigations and risks, are given a quantitative, probabilistic interpretation. This allows DDP to add up the cumulative impact of all risks, compare an individual risk’s cumulative impact, compute how much of requirements are being attained, compute how much net benefit the use of a mitigation conveys, etc. [5]

This information is used together with budget information on the cost of mitigations to make choices about which mitigations to select. The goal is to reduce the risks to sufficient levels (and so adequately attain

requirements) while remaining within resource limitations.



In this process, risk is used as the intermediary through which link requirements are indirectly linked to mitigations. Our experience is that this indirection is particularly useful. For example, the phenomenon of “diminishing returns” as more and more mitigations are applied to the same risks falls out naturally from this approach. In contrast, attempts to link requirements directly to solutions (development plans) often fail to capture the multiplicity of problems and solutions.

3. Shortcomings in the current process

The standard DDP process depicted in figure 1 involves the gathering and linking of three major concepts: system *requirements* (weighted to reflect their relative importance), *risks* that threaten to detract from attainment of those requirements, and *mitigations* to help quell those risks (and so lead to improved attainment of requirements). We have found this risk-centric approach to be quite effective in guiding experts to make their choices of mitigations. (The reader may wonder why choices have to be made among mitigations. The answer is one of resource limitation. Choosing to do all

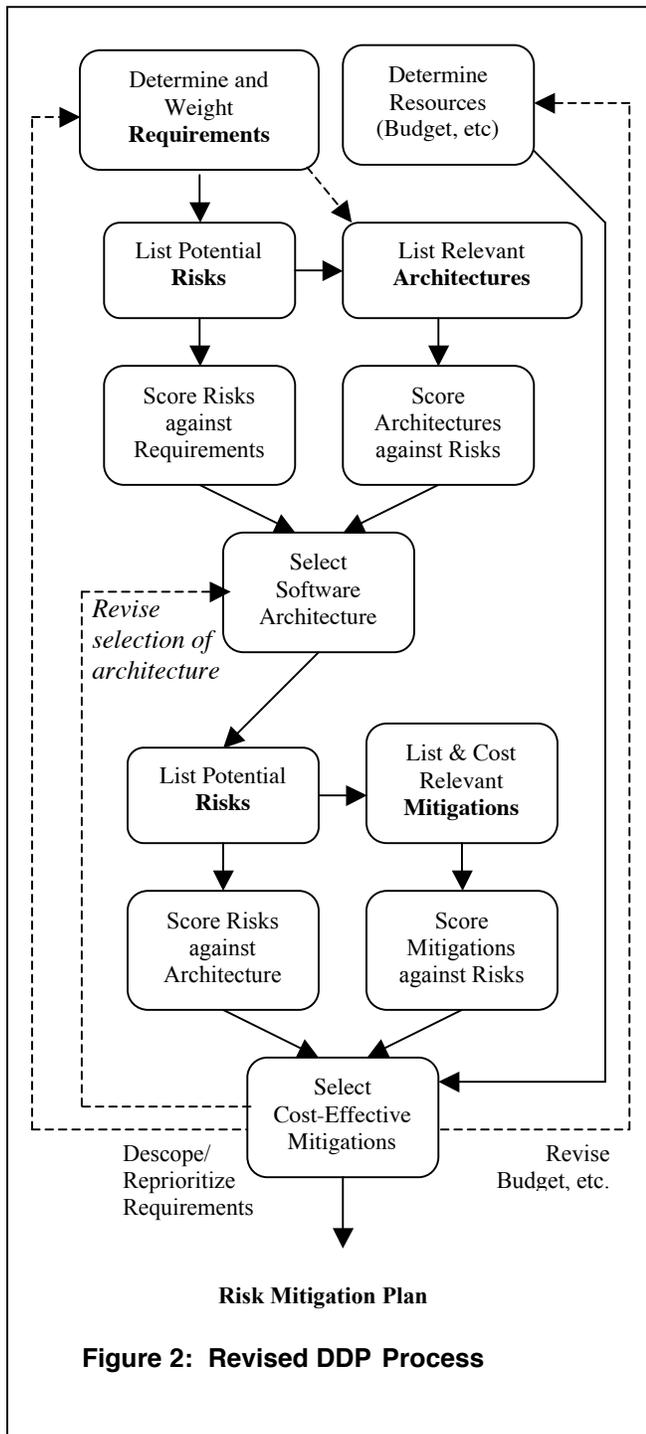
mitigations is typically not possible from a budget and time perspective.)

We have observed that in use of the DDP tool and process on JPL applications, there is some additional structure to the concepts involved that the current process is not adequately capturing. We describe how these observations lead us to now propose to include *architecture* as a first-class concept within the DDP process.

Our first step in this direction stemmed from the observation that some mitigations induce and/or exacerbate risks. For example, a vibration test may be used to check that a piece of hardware will operate correctly when subject to vibration, thus decreasing the risk of launching a spacecraft that is unable to operate under mission conditions. However, there is some risk that the test itself will cause problems (e.g., break something). The risk of those problems we term *induced* risk. Another example is of a protective coating applied to a piece of circuitry, say. Its purpose is to protect the circuitry from future damage, i.e., decrease those kinds of risks. However, should there be need to modify the circuit, that protective coating will make it harder, perhaps even impossible, to effect the modification. We describe the risks that would lead to the need for modification as *exacerbated* by the protective coating (i.e., while their likelihoods remain the same, their impact, should they occur, is increased). Software analogies of these phenomena are well known – fixing one bug may introduce new ones; introducing monitoring code may aid testing, but decrease performance (or lead to changed timing behavior when that test-time code is dropped from the final delivered code).

The standard DDP process (and its tool support) was evolved to accommodate these phenomena by extending the allowable range of the values attached to the links between mitigations and risks. Initially all such values were restricted to being positive proportions (i.e., in the range (0, 1]), indicating the proportion by which application of the mitigation would eliminate risk. Lack of a link between a risk and a mitigation indicated that the mitigation would have no effect whatsoever on that risk. The extension was to allow the expression of negative values as measures of effectiveness, where a negative value in the range [-1, 0) indicated induced risk (the more negative, the more the likelihood of the risk being induced), and a negative value in the range [-1000000, -1) indicated exacerbated risk (any existing risks’ impacts would be multiplied by the abs(value)). For example, a value of -3 means triple the impact of risks.

These extensions served their intended purpose to allow DDP studies to take into account mitigation induced/exacerbated risks. However, they opened the door to (mis?)use as a way to represent design alternatives.



To illustrate this we will first give a hypothetical and simplistic system design example. Suppose that one of the requirements for a planetary rover is to gather science data on planetary formation, using a drill to extract a core sample from rocks. Use of the drill demands a large amount of power, so lack of available power is a particularly serious risk against that science requirement.

One possible mitigation to that risk is to deploy large solar panels, capable of generating sufficient power. (An alternative could be to drill more slowly but for a longer duration). The large-solar-panel mitigation has its own risks (rover is now prone to tipping; higher overall power levels lead to the risk of electrical shorts; etc.).

A comparable software-domain example is the design of a software subsystem with the requirement that the software be able to respond to the position of the cursor by displaying context-sensitive information that the user needs. One risk to this requirement is that this information will be displayed after an unreasonably long delay. One possible mitigation is to design this system as an event-driven system with an event loop that is designed to catch and respond to mouse movements that affect cursor position.

Our first inclination was to use mitigation-induced risk as the means to represent these design options. For example, the large solar panels mitigation for the risk of lack of power we encoded as a DDP mitigation that induces the rover tipping risk, the electrical shorts risk, etc. Each of these induced risks were added into the same list of potential risks, but with the unusual characteristic that their a-priori likelihoods were set at zero (i.e., the only way those risks could occur is through being induced when the solar panel mitigation is selected). This enabled us to avoid the need for further extension to the DDP tool.

From these latter examples, it is clear that the activities that we have encoded as mitigations are, in fact, design choices. In the software arena, these choices are **software architectural decisions**. We are dissatisfied with encoding of these as just more “mitigation” choices, albeit with some unusual characteristics. At the very least, we should call these out as architectural decisions, and so be poised to take advantage of detailed methods for architectural evaluation. We would also like to avoid the need to start DDP from a “blank slate”, where all the information must be supplied anew. Clearly, the body of knowledge that pertains to architectures should be used to pre-populate DDP. Finally, and most importantly, we observe many of the risks and mitigations that derive from an architectural choice affect how well that architecture mitigates the original risks it was selected to address. For example, suppose a pipes-and-filters architecture was selected to mitigate the risk of system ossification (inability to easily make system modifications). The more the development of the system strays from strict adherence to that architecture, the more it diminishes the effectiveness of that architecture at mitigating the ossification risk. In DDP-speak, the architecture itself can be attained in whole or only in part (the latter due to the cumulative impact of risks on the realization of that architecture). Its effectiveness at mitigating risks is determined by how successfully its own risks are mitigated. We will see further examples of this in the next section.

With this observation, we propose the capture of architecture decisions *explicitly* in the DDP tool.

Before explaining this idea further, it is important to remind the reader that not all mitigations are design decisions. For example, one risk that may pertain to a piece of software is that requirements are inconsistent. One mitigation to this risk is a formal inspection process, a form of analysis. The use of formal inspections is clearly not a design decision. Indeed, in typical DDP applications, a significant proportion of mitigations fall into this testing/analysis category.

4. Incorporating software architectures

To incorporate software architectures into the DDP tool without radically changing the tool, we have

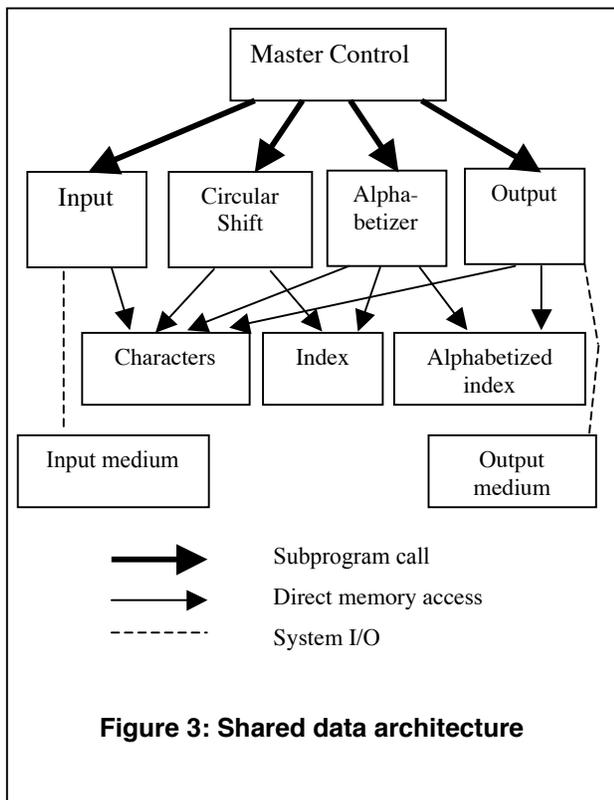


Figure 3: Shared data architecture

proposed a two-phase process as depicted in figure 2.

First we go through the original DDP process with requirements-risks-*architecture* rather than requirements-risks-*mitigations*. Thus, we are explicitly capturing alternative design architectures that will reduce or eliminate certain risks. Note that there may be a choice among several architectures that reduce a particular risk to acceptable levels.

To make this step easier, we propose seeding the DDP tool with possible classic software architectures. [1, 11] These architecture styles, e.g. pipes-and-filters,

repository, object-oriented, serve as a starting point for the architecture-selection process. Designers may, of course, add their own hybrid designs.

The architectures that result from this first step become the starting point for another iteration of the original process, one that deals with *architectures-risks-mitigations*. Thus, the architecture serves both as a mitigation of risks in the first phase, and as an *induced* requirement in the second phase. Note that the selection of architecture is an important outcome of the DDP process. Although we argued that risks themselves are merely intermediaries, we do not make the argument that architectures have a similarly nebulous status.

4.1 Examples

As a small but illustrative example, consider the classic key word in context problem [10] proposed by Parnas in 1972 (and discussed by many other researchers since.)

The KWIC [Key Word in Context] index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

We will treat this paragraph as a first-order approximation to a set of requirements. In the DDP process and tool, this set is represented in a structured form, and the importance of each is evaluated and scored. For example, we might prioritize the generation of the list of all circular shifts as the most important, with the alphabetizing of this list as being important, but having a lower priority.

Now, let us consider some of the risks that might be associated with these requirements. Parnas suggests two potential risks (although he labels these as potential design changes rather than risks.)

- 1.Changes to the processing algorithm
- 2.Changes in data representation

Garlan, et al [8] add three other risks to those of Parnas.

1. Enhancement to system function
2. Performance
3. Reuse

(A nice discussion of this example and possible architectures is provided by Shaw and Garlan. [11])

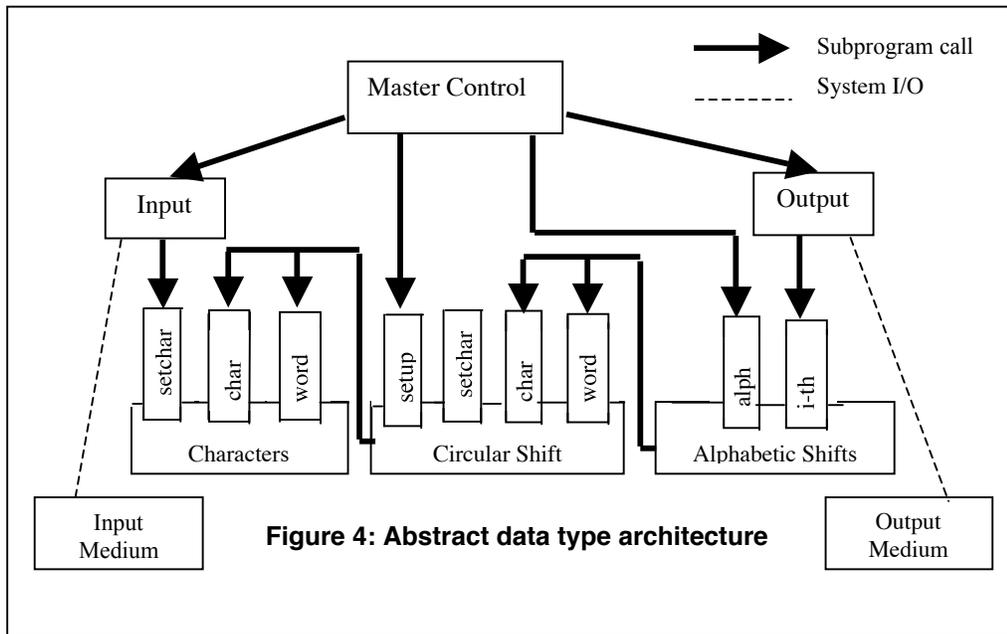
These risks are scored against requirements to see, if they occur, how they would affect each requirement.

Now, we consider possible architectures for a solution to this problem. First, consider two architectures suggested by Parnas. [10] Figure 3 illustrates shared memory architecture. Figure 4 gives an abstract data type

solution. Another possible architecture is the pipes-and-filters style as inspired by the Unix index utility and described by Shaw and Garlan. [11] This is depicted in figure 5.

type of change much more difficult. (This analysis is that of Shaw and Garlan.[11])

In the DDP process we would push the software engineers to *quantitatively* value these linkages between



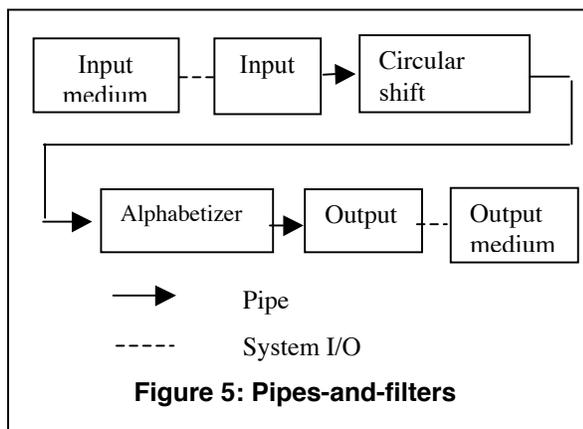
risks and architectures. For example, suppose that the engineers estimate the abstract data type design has a very small likelihood of being impacted by the risk of a change in data representation, while they estimate that performance risk of a pipes-and-filters architecture is relatively problematic. Table 1 illustrates the linkage data that engineers might produce in analyzing these architectures in light of particular risks. The numeric

The mechanism that we use to evaluate the strengths and weaknesses of each potential architecture is to score each architecture against risks that we have identified. For example, we may determine that a pipes-and-filters architecture may have performance (i.e. speed) issues although the other two possibilities are likely to perform more adequately. Conversely, the shared data and the abstract data type architectures are likely to have trouble if the algorithm for generating the index is changed. The pipes-and-filters can more easily adapt its algorithm (by

entries are in the range 0 to 1, where 0 means no effect, and 1 means that the architecture choice in that column completely eliminates the risk in that row. The DDP tool provides support for much larger matrices, and provides other views of this linkage data in addition to the tabular format.

In a realistic design, the number of requirements and potential risks can be large. In DDP applications at the component level (e.g., a memory device), it is typical to deal with 50 – 100 each of requirements, risks and mitigations, with hundreds of links between them. Even if the number of viable architecture choices is relatively small, the relationships between architecture and risks, and risk and requirements can make the choice of the preferred architecture quite complex. Addressing this complexity is a strength of DDP.

With the assistance of DDP, the design team can now select a tentative architecture. (This is a tentative architecture because the entire process is iterative. For example, the phenomena of requirements volatility and requirement creep are well known.) This begins the second phase of the DDP process. The starting point for this phase is this tentative architecture. We list potential risks inherent in this architecture. The risks enumerated in the previous phase were those associated with requirements regardless of architecture choice. Here we are looking for design and implementation risks. What things stand in the way of successfully implementing this system with this architecture? If the system is highly



merely changing or adding a filter.) However, the abstract data type obviously can change its data representation more easily; the other two would find this

interactive, a pipes-and-filters architecture style is quite risky. However, an event-driven style would have much lower risks in this area. Because of the paradigm shift needed in object-oriented design (OOD) from traditional procedural design, OOD may have a high dependency on having a trained staff.

Table 1: Risk – Architecture matrix

Risks	Architectures		
	Shared data store	Abstract data type	Pipes and filters
Algorithm change	0.9	0.7	0.1
Data representation change	0.7	0.1	0.9
Performance issues	0.1	0.1	0.7

The process of listing risks and evaluating the impact of each against the tentative architecture can be a tedious one. It is clear that many software risks are common across projects. We have preloaded DDP with a set of common software risks. (We have used the risk taxonomy identified by researchers at the Software Engineering Institute. [2]) Furthermore, we have entered linkages between these risks and a set of common architecture styles. [11] Thus, a choice of architecture obtains an associated set of risks and impacts. The design team can use this as a starting point, adding additional or more specific risks, and modifying or adding linkages.

Having identified software risks associated with this architecture, we now identify those activities, i.e. mitigations, that we can perform to eliminate, avoid, or reduce the impact of risks. For example, if there is the risk that our staff is not experienced in OOD, we could give them additional training or hire some experienced OO designers. Each such mitigation has a cost – the cost of training materials and time, or salaries and benefits for experienced designers.

We evaluate each mitigation against each risk to score its effect at reducing that risk. The effect of experience designers is likely to be greater against the risk of inexperienced staff than is training. (A new design method is often not fully understood until a certain level of experience is reached that cannot be provided by even the best training.)

Table 2 illustrates this matrix. Again, the numeric entries are in the range 0 to 1, where 0 means no effect and 1 means that the mitigation in that column completely eliminates the risk in that row.

Finally, this collection of information (risks x architecture, risks x mitigations) is combined with

budgeting information to make decisions about which set of mitigations will achieve the system requirements using the tentative architecture and within budget and resource constraints. This is typically a complex decision given the enormous number of interactions among requirements (with their relative weights), risks (with their likelihoods), the tentative architecture, mitigations (with their costs), and linkages among these. DDP provides graphical displays of this information that helps the design team explore this complex trade space. An optimizer is available that uses simulated annealing to find near optimal choices of mitigations within a specified cost bound.

Table 2: Risk – Mitigation matrix

Risks	Mitigations		
	Provide OOD training	Hire experienced OOD staff	Perform formal inspections
Inexperienced staff	0.7	0.9	0.0
Inconsistent requirements	0.0	0.1	0.9

As mentioned previously, this is an iterative process. In these activities, it is common for the design team to discover additional requirements or learn of the infeasibility of certain requirements (resulting in the need for *descopeing* [6]). Additional risks of a particular architecture choice may not be apparent until very late in the process. Thus, the entire DDP process may be iterated to capture these changes. However, note that subsequent iterations are likely to be more efficient because of the leverage of information derived during previous iterations.

The reader may be struck by the length and complexity of this process. We assert that this is the nature of the task, not a side effect of our process. Design of a complex software system is difficult.

5. Conclusions, Status, and Related Work

The argument set forth in this paper is that risk can and should be used to guide architectural decisions. These include both the choice of architecture itself, and the decisions that flow from that choice. We have shown how we arrived at this position through our observations of a risk-based decision process in use in real-world design activities. The gradual evolution of that process has led to the point where we believe that architecture deserves a place as a first-class object within the process itself. These points have been illustrated using a small but familiar

example, the key word in context problem introduced by Parnas.

The status of this work is that all the aspects of DDP described in section 3 exist and have seen use in actual spacecraft technology risk studies. Instances of the phenomena we described in that section, of mitigation induced or exacerbated risks, and of design decisions encoded via this mechanism, have arisen in these same actual studies. The extensions needed of the DDP tool to support the two-phase approach, with mitigations leading to derived requirements, have been incorporated in an, as yet, unreleased version. We have used this within our own experimentation, but it has not yet seen field use in real project applications. Likewise, our encoding of architectural considerations is also at the stage of internal experiments that have yet to see actual customer application. Additional information DDP can be found at the Defect Detection and Prevention website, <http://ddptool.jpl.nasa.gov>

A full comparison with related work is beyond the scope of this workshop paper. We do draw attention to a distinguishing characteristic of DDP, namely that it is able to accommodate both architectural design decision concerns, and other elements of project planning (analysis, testing and process, represented as mitigations in the DDP framework). Furthermore, DDP does so in a quantitative manner. The combination of these aspects sets DDP apart from many of the other approaches to architectural decision making, e.g., the influence diagrams of [3] (shown in use in [9]), or the goal graphs of [12]. A key common thread that we have with those referenced bodies of work is the reliance on computer support for decision-making. Real-world problems involve a myriad of concerns, whose number and complex interconnections warrant support.

Acknowledgements

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology and at Miami University, Oxford, Ohio, both under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, the Jet Propulsion Laboratory, California Institute of Technology, or Miami University.

Dr. Steve Cornford leads the DDP project at JPL. DDP technology has been developed with NASA's Software IV&V, Code R and Code Q support, managed in part through JPL's Assurance Technology Program Office.

6. References

- [1] Bass, L., P. Clements, et al. (1998). *Software Architecture in Practice*. Boston, Addison-Wesley.
- [2] Carr, M. J., S. L. Konda, et al. (1993). *Taxonomy-Based Risk Identification*. Pittsburg, PA., Software Engineering Institute.
- [3] Chung, L., B.A. Nixon, B.A., E. Yu, E., and Mylopoulos, J., 2000 "Non-Functional Requirements in Software Engineering" Kluwer Academic Publishers.
- [4] Cornford, S. L., M. S. Feather, et al. (2001). DDP – A tool for life-cycle risk management. IEEE Aerospace Conference, Big Sky, Montana.
- [5] Feather, M.S., Cornford, S.L. Dunphy, J. & Hicks, K.A. (2002). A Quantitative Risk Model for Early Lifecycle Decision Making; Proceedings of the Conference on Integrated Design and Process Technology, Pasadena, California, June 2002. Society for Design and Process Science
- [6] Feather, M.S., S.L. Cornford & K.A. Hicks (2002) Descoping; Proceedings of the 27th IEEE/NASA Software Engineering Workshop, Greenbelt, Maryland, Dec 2002. IEEE Computer Society.
- [7] Gamma, E., R. Helm, et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts, Addison-Wesley.
- [8] Garlan, D., G. E. Kaiser, et al. (1992). "Using tool abstraction to compose systems." IEEE Computer 25(6).
- [9] Mylopoulos, J., L. Chung, S. Liao, H. Wang & E. Yu. "Exploring Alternatives during Requirements Analysis", IEEE Software 18(1), Jan-Feb 2001, pp 92-96.
- [10] Parnas, D. L. (1972). "On the criteria to be used in decomposing systems into modules." Communications of the ACM 15(12): 1053-1058.
- [11] Shaw, M. and D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, Prentice-Hall.
- [12] van Lamsweerde, A., 2001, "Goal-Oriented Requirements Engineering: A Guided Tour", Proceedings 5th IEEE International Symposium on Requirements Engineering, Toronto, Canada, August, pp. 249-2

From Goal-Oriented Requirements to Architectural Prescriptions: The Preskriptor Process

Manuel Brandozzi
UT – ARISE

*Advanced Research in Software Engineering
The University of Texas at Austin
manuelbr@mail.utexas.edu*

Dewayne E. Perry
UT – ARISE

*Advanced Research in Software Engineering
The University of Texas at Austin
perry@ece.utexas.edu*

Abstract

The step from the requirements for a software system to an Architecture for the system has traditionally been the most complex one in the software development process. This step goes from what the system has to achieve, to how it achieves it. In order to make this step easier, we propose the use of Preskriptor, a prescriptive architectural specification language, and of its associated process, the Preskriptor process. Architectural prescriptions consist in the specification of the system's basic topology and of the constraints associated with it and its components and interactions. The Preskriptor process provides a systematic way to satisfy both the functional and non functional requirements from the problem domain, as well to integrate architectural structures from solution domains.

1. Introduction

The most difficult transition in the development process for a non-trivial software system is likely the one from the requirements for the system to the system's architecture. This step involves going from the problem's domain to the domain of its solution [1]. One of the factors that makes the design of software systems so challenging is that they have to satisfy many different requirements (problems) at the same time, and there is often more than a single solution to a particular requirement.

Requirements specifications can be viewed as a contract between the customer and the software developers. Hence, they should be not only easy to understand by the software architects and engineers but also by the domain experts and users.

We propose the use of architectural prescriptions [2] to perform the step from requirements to architecture. An architectural prescription is the architecture of the system in terms of its components, the constraints on them and the interrelationships among the component (i.e., the constraints on their interactions). At least initially, the

constraints are only those coming from the problem domain. While architectural descriptions provide more or less complete details to the designers, prescriptions make the step from requirements to architecture easier to model and to perform. Prescriptions may also provide a means of deeper understanding about the architecture. We will show how we can perform this step from goal-oriented requirements. Another advantage of prescriptions is that, being at a higher level of abstraction, they can be reused more easily, and they enable more creative designs.

The same prescription could be used for an entire software family [3] of applications that differ only in deployment requirements. If the applications differ also in some requirements coming from the problem domain, like the interaction with different types of users, we can first develop the prescription for an ancestor system that has all and only the requirements common to the whole family and then get, by extending this prescription, the prescriptions for all the descendent applications.

Because Architectural Prescription Languages APLs, which we introduced in [4], are written in an elementary ontology, they enable new, innovative designs. Let's consider, for example, a distributed system. An architecture description language may include elements such as clients and servers. It may be that the architect writing a specification in such an architecture description language uses client and server components also when, for example, a multi-peer architecture might be a better solution. The designer will then be constrained by such architecture to a low-level design that adopts a client-server solution. By describing the system at a higher level of abstraction, a specification in an architectural prescription language would instead permit the designer to choose the best solution at the design level and even let him/her take different choices for different members of the family.

The paper is structured as follows: we first give an overview of KAOS, the requirements specification language our process uses as a starting point; then we introduce the Preskriptor architectural prescription language and process illustrating them with a practical example; we conclude by summarizing the fundamental

results of the paper, and by discussing the future directions of our research.

2. Overview of the KAOS Specification Language

KAOS is a goal oriented requirements specification language [5]. Its ontology is composed of objects, operations and goals. Objects can be agents (active objects), entities (passive objects), events (instantaneous objects), or relationships (objects depending on other objects). Operations are performed by an agent, and change the state of one or more objects. They are characterized by pre-, post- and trigger- conditions.

Goals are the objectives that the system has to achieve. In general, a goal can be AND/OR refined till we obtain a set of achievable sub-goals. The goal refinement process generates a goal refinement tree. All the nodes of the tree represent goals. The leaves may also be called requisites. The requisites that are assigned to the software system are called requirements; those assigned to the interacting environment are called assumptions.

Let's briefly see how obtain a requirements specification in KAOS. 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 that are 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 [6]. It's up to the requirements engineer to generate a "good" refinement tree. By "good" refinement tree we mean one that does not contain conflicts among refinements of different goals and from which it is possible to derive an architecture that achieves those goals. In addition to iterations with the requirements specification process, there may also be iterations between the requirements specification process and the architecture prescription process.

In figure 1., there is an example of a goal specified in KAOS, taken from the example we'll use in next section.

```
Goal Maintain[ConfidentialityOfSubmissions]
InstanceOf SecurityGoal
Concerns DocumentCopy, Knows, People
ReducedTo
  ConfidentialityOfSubmissionDocument
  ConfidentialityOfIndirectSubmission
InformalDef A submission must remain
  confidential. A paper that has to
  be submitted has to remain
  confidential.
```

Figure 1. Example of a goal specification in KAOS

The keyword *Goal* 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* contains the names of the sub-goals into which the goal is resolved. *InformalDef* is the informal definition of the goal. Then there could be *FormalDef*, n optional attribute; it contains a formal definition of the goal (which can be expressed in any formal notation such as first order logic).

3. The Preskriptor Process

We will illustrate our technique with an example. In the example, we shall obtain an architectural prescription for a system that automates some of the functions in the paper selection process for a scientific magazine (or a conference). Our starting point is a specification of this software system in KAOS. The fundamental goal of the paper selection system is to keep high the quality of the magazine.

We have to determine the fundamental goal (root goal) that the system has to achieve; this goal is the only unavoidable constraint coming from the problem domain. By using a KAOS specification as a starting point, we can gradually increase the degree of constraint of the solution by considering the goals that refine the root goal. We can keep on refining goals to an appropriate level. The Preskriptor process can take as input goals in any level of the resulting goal refinement tree.

If we take the root of the tree, although the resulting prescription will enable new, innovative solutions to the problem, it will generally provide too little guidance to the system's designers.

On the other hand, taking the leaves of the goal refinement tree (or even a further refining of the prescription to achieve qualities as performance, reusability, etc.) may produce a specification that constraints too much the lower level designs. As Parnas once noted, if in order to design washing machines we used all the requirements coming from how we wash the clothes by hand, we wouldn't have got the very effective rotary washing machines of nowadays.

Our approach leaves the software architect free to choose the degree of constraint desired on the architecture. Also, he or she could change the degree of constraint during the architecture process according to necessity. In the example that follows we use a high degree of constraint (i.e. we consider goals deep in the goal refinement tree) only for demonstration purposes.

The process of deriving the prescription is composed of three steps that can be followed by an optional one, and which may be iterated. In the first step we derive the basic prescription from the root goal for the system. This root goal is either already given or it can be obtained by abstracting its sub-goals. In the second step we get the

components that are potential sub-components of the basic architecture considering the objects that are in the KAOS specification. In the third step we choose a level of refinement of the goal refinement tree that we consider appropriate, we decide which of the sub-goals at this level are achieved or co-achieved by the software system, and we assign them to the sub-components which we derived at step 2. As a last step, the architectural prescription may be further refined to achieve additional non-functional properties.

Our example considers the KAOS specification for the paper selection process developed in the thesis [7]. We shall transform this KAOS specification into a prescription for a Software System that is to assist in the paper selection process. Figure 2. illustrates the first three steps of the process.

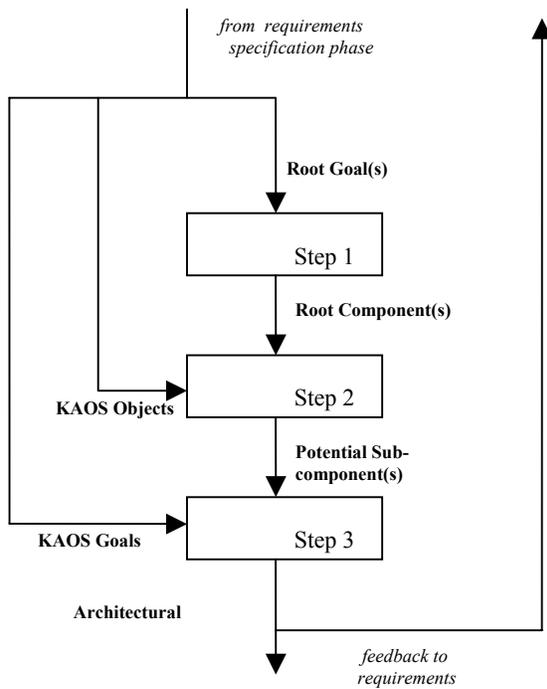


Figure 2: The fundamental steps of the Preskriptor process

3.1 The First Step of the Methodology

The software system, that we hereafter denote as “SelectionManager”, is co-responsible for the root goal “Maintain[QualityOfTheScientificMagazine]” together with the system composed of the people involved. The software system performs different functions that can be automated and it interacts with the human system. Its purpose is to speed up the paper selection process and to improve its confidentiality.

The Preskriptor language is an implementation of the APL introduced in [4].

Preskriptor Specification: ScientificPaperManager
KAOS Specification: PaperSelectionProcess
Components:

Component SelectionManager [1,1]
Type Processing
Constraints
 Maintain[QualityOfTheScientificMagazine]
Composed of ...
Uses PeopleConnect *to interact with* (AutorAgent,
 ChiefEditorAgent,
 AssociatedEditorAgent, EvaluatorAgent)

Figure 3: Example of a specification in Preskriptor

At the beginning of a Preskriptor specification is the declaration of its name. It’s followed by the declaration of the KAOS specification from which the prescription is derived. A prescription may derive from only one KAOS specification, and if the prescription derives from several different KAOS specifications, it’s better to merge the specifications first and then to architect the system. By doing so, if there are conflicts between goals in different specifications they will be solved early at the requirements phase. So, all the components of a prescription derive from the same KAOS specification, which may be the union of several KAOS specifications. Following are the definitions of the components.

The field *Component* specifies the name of the component. *Type* denotes the type of the component. *Constraints* is the most important attribute of a component. It denotes which are the requirements that the component is responsible for. We use here the term constraint to denote both functional and non-functional constraints (both corresponding to requirements on the system). *Composed of* identifies the subcomponents that implement the component. The last attribute, *Uses*, indicates which are the components used by the component. Since interactions can only happen through a connector, the *Uses* attribute has the additional keyword *to interact with* denoting which components the component interacts with using a particular connector.

At the highest layer of abstraction, to which the first step of the specification corresponds, we have to write next to the name of a component its possible number of instances in the system. At the other layers this information is optional because it will be contained anyway in the *Composed of* field of the super-component of the component considered. For example, [1,n] means that the component can have any number of instances from 1 to an arbitrary number n.

We will fill in the *Composed of* field after we decide how to refine the system at the third step. The software

system has to interact with the people involved in the process. To do so, it uses the (fairly complex) connector “PeopleConnect”. To distinguish the people involved in the process (agents) from the data components that may be used in the software system to represent them, we added the Agent suffix to their names. PeopleConnect is specified as follows:

```

Component PeopleConnect [1,n]
Type Connector
Constraints
  Maintain[QualityOfTheScientificMagazine]
Composed of ...
Uses /

```

Figure 4. Example of a connector specification

The symbol “/” means none and, for now, we will omit the fields whose value is none. The formal specification of the Preskriptor language is in the Appendix.

3.2 The Second Step

From the objects in the KAOS specification we derive potential data, processing and connector components that can implement SelectionManager. If in the third step we don’t attribute any constraint to these potential components, they won’t be part of the prescription. In that case, in fact, they won’t be necessary to achieve the goals of the KAOS specification. In figure 5. is a sample this set for the paper selection process.

```

Component Document
Type Data
Constraints ...

Component Paper
Type Data
Constraints ...

Component People
Type Data
Constraints ...

Component Knows
Type Data
Constraints ...
Composed of People[0,m], Document[0,n]

```

Figure 5. Sample of potential components for the paper selection system

The notation, used in the *Composed of* field of the last component, means that the component is composed of 0 or more “People” sub-components and by 0 or more “Document” sub-components. Obviously, the number of instances assigned to different sub-components doesn’t have to be the same.

“SelectionManager” could be composed also of the following processing component, and the following connectors, which connect the processing component to the data ones.

```

Component SelectionManagerEngine
Type Processing
Constraints
  Maintain[QualityOfTheScientificMagazine]
Composed of ...
Uses
  PeopleConnect to interact with
    (AuthorAgent, ChiefEditorAgent,
     AssociatedEditorAgent, EvaluatorAgent),
  Conn1 to interact with Document,
  Conn2 to interact with Paper,
  ...

Component Conn1
Type Connector
Constraints ...

...

```

Figure 6. SelectionManagerEnging and associated connectors

3.3 The Third Step

Now we will complete the architectural prescription by taking into account the goals that are at the goal refinement tree level that we selected. We show how to put constraints on the architectural components we got at step 2.

Let’s first refine our root goal. After a first refinement, the subgoals of the root that the software system needs to achieve are:

```

Maintain[OriginalityOfSubmission],
Maintain[QualityOfPublishedArticles],
Maintain[QualityOfPrint],
Achieve[EnoughQuantityOfPublishedArticles].

```

By refining the first of these goals, we obtain the following sub-goals:

```

Maintain[QualityOfEditorialDecisions],
Maintain[PertinenceOfPublishedArticles].

```

After two more refinements we obtain:

```

Avoid[ConflictOfInterestsWithAssociatedEditor]

```

This goal can translate directly into a constraint on the “SelectionManagerEngine” and “People” subcomponents. “SelectionManagerEngine” will somehow keep track of the different ways the various people represented by the People data component may know each other. The two

constrained components are able to achieve this requirement and the existence of this requirement is a sufficient condition for the existence of the two components given our architectural rationale. By this we mean that these components ought to exist even if they have no other goals to achieve. On the other hand, if we don't care anymore about this requirement and there are no further constraints assigned to these components, there is no point in keeping them. By proceeding in a similar fashion with the rest of the goal refinements, we obtain the first version of a complete Preskriptor specification:

```

Preskriptor Specification: ScientificPaperSelector
KAOS Specification: PaperSelectionProcess
Components:

Component SelectionManagerEngine [1,1]
Type Processing
Constraints
  Avoid[ConflictOfInterestsWithAssociatedEditor]
  Avoid[SurchargeAssociatedEditor],
  Achieve[ListOfPotentialEvaluators],
  Avoid[ConflictsWithEvaluator],
  Maintain[CommittedEvaluator],
  Avoid[SurchargeEvaluator],
  Maintain[FeedbackOnPaper],
  Maintain[ConfidentialityOfPapers],
  Maintain[IntegrityOfPapers],
  Maintain[ConfidentialityOfSubmission],
  Maintain[IntegrityOfEvaluation],
  Maintain[ConfidentialityOfSensibleDocument]
Composed of ...
Uses
  PeopleConnect to interact with (AutorAgent,
    ChiefEditorAgent,
    AssociatedEditorAgent,
    EvaluatorAgent),
  Conn1 to interact with Document,
  Conn2 to interact with Paper,
  ...

Component Document [0,n]
Type Data
Constraints
  Maintain[FeedbackOnPaper],
  Maintain[IntegrityOfEvaluation]

Component Paper [0,n]
Type Data
Constraints Maintain[IntegrityOfPapers],

Component Conn1 [1,n]
Type Connector [1,n]
Constraints
  Maintain[IntegrityOfEvaluation],
  Maintain[ConfidentialityOfSensibleDocument]
  ...

```

Figure 7. A prescription for the paper selection process after step 3

We omitted the complete specification, but if we included it, it would be possible to notice that the components: ChiefEditor, Author, Knows, Holds, IsAuthorOf, Supervise, InChargeOf and Evaluates, which were potential sub-components at step 2, were removed from the prescription because they are not necessary to achieve the sub-goals for the system. This is due to the rationale that we took in prescribing the system. Different architects may use different rationales and produce different prescriptions.

At the third step (and at the optional fourth) we first consider the functional goals and then the non-functional ones. The goals of the latter type have a more complex effect on the system to achieve. In the most general case, apart from further constraining already existing components, they introduce new components and they transform the system's topology (i.e. they change the relationships among the system's components). Details on how the Preskriptor process manages non-functional requirements can be found in [8].

3.4 The fourth step

At this step of the prescription design process, the architectural prescription is further refined to make the system achieve goals that are not from the problem domain. These additional goals are typically introduced for a variety of reasons (for example architectural, economic, etc.).

These goals can be classified as follows: useful architectural properties, even though not required by the problem (such as reusability, evolvability, etc.), conformance to a particular architectural style, and compatibility goals (such as compatibility with a given platform or industry standard, or platform independency).

Examples of architectural goals are reusability, location transparency and dynamic reconfiguration. These goals can modify the prescription at the component level, at the sub-system level, or affect the whole system.

As practical experience has shown [8], architectural styles can be chosen as a particular solution to achieve some goals or to refine some components. For example, we can achieve the architectural goal of dynamic reconfiguration by making all the components adhere to the reconfigurable architectural style. By dynamic reconfiguration we mean that the application can evolve after it has been already deployed as demands change for new and different kinds of configuration. A reconfigurable architectural style is the following set of constraints: provide location independence; initialization must provide facilities for start, restart, rebuilding dynamic data, allocating resources, and initializing the component; finalization must provide facilities for preserving dynamic data, releasing resources, and terminating the component.

The last kind of goals that don't come from the problem domain are compatibility goals. They further constrain a prescription to take into account, already at this architectural design level, the need to assure the compatibility of the system with one or more industry standard(s) and/or platform(s). For example we may want to make a system CORBA or Linux compatible. This may be motivated by the need to assure compatibility with legacy systems, other vendors systems, available machines, or just for some marketing strategies.

Fig. 11 shows how step 4 interacts with the previous steps of the Preskriptor process.

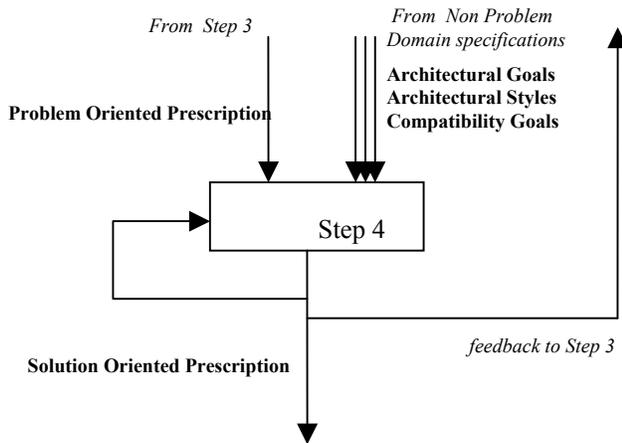


Figure 8: Step 4 of the Preskriptor process

As we can see, in general, the fourth step is iterated till we have achieved all of the non-domain goals. This step may also be iterated with step three. In that case, alternative problem domain goal refinements and/or components may be chosen to make the later prescription design steps possible or easier to perform.

It's important to distinguish between the artifact of the third step and the one of the fourth. The third step produces an artifact whose only constraints come from the problem domain, which can be reused with similar systems without over-constraining them. On the other hand after the fourth step we obtain a prescription that takes into account also constraints that we introduced for the particular product we are developing, such as the use of a particular architectural style or the compatibility with a certain industry standard. While the artifact of step four may be reused with other systems that we want to develop in a similar manner, we also want to be able to easily reuse a prescription in systems that are to be implemented with different non domain constraints, like with different architectural styles. For this reason we distinguish between the specification of the prescription after step 3.,

which we call Problem Oriented Prescription (POP), from the one after step 4, which we call Solution Oriented Prescription (SOP).

Given the Problem Oriented Prescription for the system and the non-domain driven goals, step 4 proceeds similarly to step 3. It takes as inputs a POP and the non problem domain goals, and gives a SOP as a result. In this step the non-domain goals are assigned as constraints to some POP components and/or the topology of the POP may be modified in order to achieve them (in this step we may reintroduce some of the KAOS components that we discarded at step three).

A Solution Oriented Prescription specification is similar to a POP specification, but it includes one or more of the following additional attributes: Architectural Goals, Architectural Styles and Compatibility Goals Specification. These new attributes are needed to keep track of the specifications of the goals, which don't come from the problem domain.

4. Conclusion

This paper presents an introduction to Preskriptor a method for transforming a requirements specification into an architectural prescription. Architectural prescriptions are a higher-level form of architectural specifications that interface more easily with requirements specifications and that do not include implementation oriented entities such as client-server which are often default components in architectural descriptions. We illustrated how to derive a prescription with a practical example. The key steps in the prescription specification process are: the selection of the right level of goal refinement, the choice of the potential components for the architecture, the assignment of the constraints to the potential components for the architecture and, often in the case of non-functional requirements, the modification of the architecture's topology.

Preskriptor is a systematic and rigorous process to make sure that none of the requirements are neglected, that no useless requirements and/or components are introduced and that the means for easily modifying the architecture are provided. The generality of our approach will allow the architects to choose their favorite ADL, or design specification, to describe at a lower level an architecture prescribed in Preskriptor.

The objectives for the future of our research are the extension of the methodology to take into account the most common non-functional requirements, the test of the methodology with case studies and empirical studies, and the development of a supporting tool.

5. References

- [1] Jackson, M., “The world and the machine”. *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington (USA), April 1995. Keynote speech
- [2] Perry, D.E., Wolf, A. L., “Foundations for the Study of Software Architecture”, *Software Engineering Notes*, ACM SIGSOFT, October 1992, pp. 40-52
- [3] Parnas, D. L., “On the Design and Development of Program Families”, *IEEE Transactions on Software Engineering*, IEEE Computer Society, March 1976, pp. 1-9
- [4] Brandozzi, M., and Perry, D. E., “Transforming Goal Oriented requirements specifications into Architectural Prescriptions”, *Proceedings of STRAW '01, ICSE 2001*, Toronto, May 2001, 54-61
- [5] Van Lamweerde, A., Darimont, R., and Massonet, P., “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
- [6] Van Lamweerde, A., Darimont, R., and Letier, E., “Managing Conflicts in Goal-Driven Requirements Engineering”, *IEEE Transactions on Software Engineering*, IEEE Computer Society, November 1998, pp. 908-925
- [7] Cordier, C., and Van Lamweerde, A., “Analyse des contraintes de sécurité pour la gestion électronique d’une revue scientifique”, Université Catholique de Louvain, 1997

[8] Brandozzi, M., and Perry, D., “Architectural Prescriptions for Dependable Systems”, *Proceedings of WADS, ICSE 2002*, Orlando, May 2002

[9] Dewayne E. Perry. “A Product Line Architecture for a Network Product” *ARES III: Software Architectures for Product Families 2000*, Los Pamos, Gran Canaria, Spain, March 2000

6. Appendix

Prescriptor Specification: [Prescription’s name]

(KAOS Specification: [Requirements specification’s name])[?]

Components:

(

Component [Component’s name] ([num1, num2])[&]

Type {Processing | Data | Connector}

Constraints ([Constraint’s name],)⁺

(Composed of ([Component’s name] [num1, num2],)^{*})[?]

(Extends [Component’s name])[?]

(Generalizes ([Component’s name],)⁺)[?]

(Uses [Connector’s name] to interact with ([Component’s name],)⁺)^{*}

)⁺

The terms between brackets denote the meaning of the identifier that will be at their place. “*” means that the preceding expression can be present 0 to an arbitrary number of times. “+” is the same except that it has to be present at least once. “?” means the expression can be present 0 or 1 time only. The new symbol “&” means that the expression is required only for the specification of the components at the first level of the components refinement tree.

Integrating Organizational Requirements and Socio-Intentional Architectural Styles

Lúcia R. D. Bastos¹, Jaelson F. B. Castro¹, John Mylopoulos²

¹ *Centro de Informática, Universidade Federal de Pernambuco, Av. Prof. Luiz Freire S/N, Recife PE, Brazil 50732-970, +1 5581 {lrdb, jbc}@cin.ufpe.br*

² *Dept. of Computer Science University of Toronto, 10 King's College Road Toronto M5S3G4, Canada, +1 416 978 5180 jm@cs.toronto.edu*

Abstract

Software systems of today are characterized by increasing size, complexity, distribution, heterogeneity, and lifespan. Understanding and supporting the interaction between software requirements and architectures remains one of the challenging problems in software engineering research. To address these challenges we are proposing an integration framework developed within the context of the Tropos project. The proposal aims at identifying the key architectural elements and the dependencies among those elements, based on the stated system requirements.

1. Introduction

Requirements Engineering and Software Architecture have become established areas of research, education and practice within the software engineering community.

Evolving and elaborating system requirements into a viable software architecture satisfying those requirements is still a difficult task, mainly based on intuition. It also remains a challenge 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.

In this work we show an approach for this integration of systems requirements and software architectures within the context of the Tropos project, an information system development framework which

is requirements-driven in the sense that it adopts concepts used during early requirements analysis. To model and understand issues of the application domain (the enterprise) we use the *i** technique [2],[3], which allows a better description of the organizational relationships among the various agents of a system as well as an understanding of the rationale of the decisions taken. In the architectural design we use a catalogue of socio-intentional structures adopting a set of architectural styles for multi-agent systems motivated in organization theory and strategic alliances [4], [5], [6].

The paper is structured as follows. Section 2 presents the Tropos ontology, including a modeling framework for requirements analysis namely the *i** technique, and the organizational-inspired architectural styles. Section 3 emphasize the existence of conceptual differences between requirements and architecture. Section 4 introduces the baseline of our proposal to integrating organizational requirements and socio-intentional styles. Finally, Section 5 summarizes the related work, concludes the papers with contributions and points to further work.

2. The Tropos Methodology

The Tropos methodology adopts the view of information systems as social structures. By social structures, we mean a collection of social actors, human or software, which act as agents, positions, or roles and have social dependencies among them. Tropos is intended as a seamless methodology tailored to describe both the organizational environment of a system and the system itself in terms of the same concepts.

The Tropos ontology is described at three levels of granularity [1]. At the lowest (finest granularity) level, Tropos adopts concepts offered by the i* organizational modeling framework [2], [3], [4], such as actor, agent, position, role, and social dependency.

At a second, coarser-grain level the ontology includes possible social patterns, such as mediator, broker and embassy. At a third, more macroscopic level the ontology offers a set of organizational styles inspired by organization theory and strategic alliances literature. All three levels are defined in terms of the i* concepts.

Tropos methodology spans four phases:

- Early requirements - concerned with the understanding of a problem by studying an organizational setting; the output is an organizational model that includes relevant actors, their goals and dependencies.
- Late requirements - the system-to-be is described within its operational environment, along with relevant functions and qualities.
- Architectural design - the system's global architecture is defined in terms of subsystems, interconnected through data, control and dependencies.
- Detailed design - behavior of each architectural component is defined in further detail.

More details about *Tropos Methodology* can be found in [1].

2.1 Requirements in the I* framework

This section will review the main concepts of the i* technique [2], [4]. It is a framework, which focuses on the modeling of strategic actor relationships of a richer conceptual model of business processes in their organizational settings. The ontology of the i* technique [4] caters to some of these advanced concepts. It can be used for: (i) obtaining a better understanding of the Organizational relationships among the various system agents; (ii) understanding the rationale of the decisions taken; and (iii) illustrating the various characteristics found in the early phases of requirements specification. According to this technique, the participants of the organizational setting are actors with intentional properties, such as, goals, beliefs, abilities and compromises. These actors depend upon each other in order to fulfill their objectives and have their tasks performed.

The i* technique consists of two models: The Strategic Dependency Model (SD) and the Strategic Rationale Model (SR).

The Strategic Dependency Model (SD) consists of a set of nodes and links connecting them, where nodes represent actors and each link indicates a dependency between two actors. Hence, a model is described in

terms of network of dependency relationships among various actors, capturing the motivation and why of activities. We can distinguish, four types of dependencies, three of them related to existing intentions – *goal dependency*, *resource dependency* and *task dependency* – while the fourth is associated with the notion of non-functional requirements, the so called *soft-goal dependency*. In the *goal dependency*, an agent depends on another one to provide the desired condition, and it does not worry about how this condition is achieved. In the *resource dependency*, the agent depends on the availability of physical resource or information. In the *task dependency*, the agent informs the other what (and how) should be done. The *soft-goal dependency* is similar to the *goal dependency*, except that the condition is not precisely defined at the start of the process, i.e., the goals in a sense involves subjective aspects, that gradually are clarified during the development process. This type of dependency provides an important link connecting two important aspects in software engineering: (i) the technical and (ii) managerial side. We still can identify different degrees of dependencies: open, committed and critical [5]. We can distinguish actors as agents, *roles* and *positions*. An agent is an actor with concrete physical manifestations. It is a person or artificial agents (hardware/software). A role is an abstract characterization of the behavior of a social actor within some specialized context, domain or endeavor. A position is a set of roles typically played by one agent. Moreover we can analyze opportunities and vulnerabilities of the chain dependency [3].

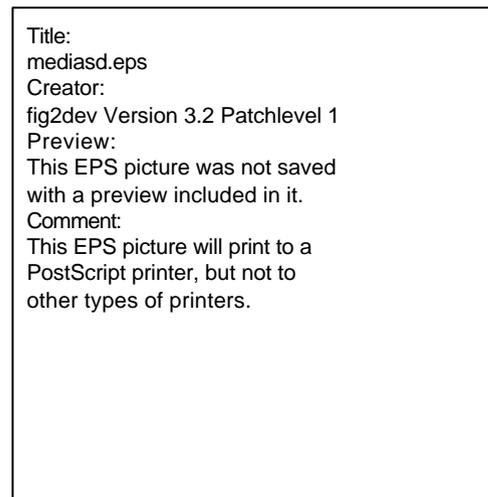


Figure 1 – SD model for Media Shop

In the Figure 1, we have the Strategic Dependency (SD) model of the e-commerce example. The *Media Shop* is a store selling and shipping different kinds of media items such as books, newspapers, magazines, audio CDs, videotapes, and the like [1]. To increase

market share, *Media Shop* has decided to open up a B2C retail sales front on the internet. With the new setup, a customer can order *Media Shop* items in person, by phone, or through the internet. The system has been named *Medi@* and is available on the world-wide-web using communication facilities provided by *Telecom Cpy*. It also uses financial services supplied by *Bank Cpy*, which specializes on on-line transactions. *Medi@* system is introduced as an actor in this strategic dependency model depicted.

Title:
 mediasr.eps
 Creator:
 fig2dev Version 3.2 Patchlevel 1
 Preview:
 This EPS picture was not saved with a preview included in it.
 Comment:
 This EPS picture will print to a PostScript printer, but not to other types of printers.

Figure 2 – SR model for Medi@

The second model of the technique *i** is the Strategic Rationale Model (SR). It is used to: (i) describe the interests, concerns and motivations of participants process; (ii) enable the assessment of the possible alternatives in the definition of the process; and (iii) research in more detail the existing reasons behind the dependencies between the various actors. Nodes and links also are part of this model. It includes the previous four types of nodes (present in the SD model): *goal*, *task*, *resource* and *soft-goal*. There are two new types of relationship, *means-end* that suggests that there may be other means of achieving the objective (alternatives) and *task-decomposition* that describes what should be done in order to perform a certain task.

The analysis in Figure 2 focuses on the software (Media), instead of an external stakeholder. The figure postulates a root task *Internet Shop Managed* providing sufficient support (++) [13] to the softgoal

Increase Market Share. That task is firstly refined into goals *Internet Order Handled* and *Item Searching Handled*, softgoals *Attract New Customer*, *Secure* and *Usable* and tasks *Produce Statistics* and *Maintenance*. *Internet Order Handled* is achieved through the task *Shopping Cart*, which is decomposed into subtasks: *Select Item*, *Add Item*, *Check Out*, and *Get Identification Detail*. These are the main process activities required to design an operational on-line shopping cart. More details can be founded in [1].

In next section we will detail the organizational-inspired architectural styles *Tropos*, which consider information systems as social structures all along the development life cycle.

2.2. Socio-Intentional Architectural Styles

A system architecture constitutes a relatively small, intellectually manageable model of system structure, which describes how system components work together. Unfortunately, traditional architectural styles for e-business applications [12],[13] focus on web concepts, protocols and underlying technologies but not on business processes nor non functional requirements of the application. As a result, the organizational architecture styles are not described nor the conceptual high-level perspective of the e-business application.

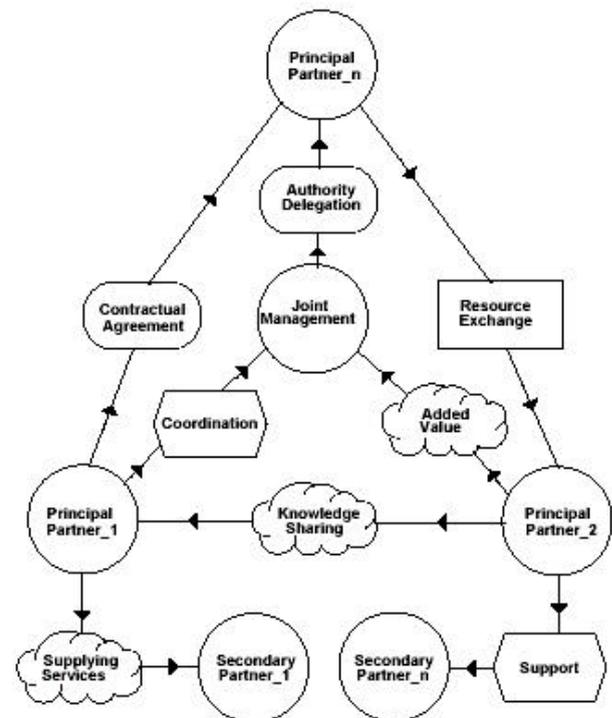


Figure 3 – The joint venture pattern

Tropos has defined organizational architectural styles [1],[5],[6],[7] for agent, cooperative, dynamic

and distributed applications to guide the design of the system architecture. These architectural styles (*pyramid*, *joint venture* (Figure 3), *structure in 5*, *takeover*, *arm's length*, *vertical integration*, *co-optation*, *bidding*) are based on concepts and design alternatives coming from research on organization management. The proposal is to use human organizations as a metaphor to suggest a set of generic styles for agent systems, with a preference for organizational design theories over social emergence theories.

For example, the joint venture architectural style in Figure 3. The joint venture style is a more decentralized style based on an agreement between two or more principal partners who benefit from operating at a larger scale and reuse the experience and knowledge of their partners. Each principal partner is autonomous on a local dimension and interacts directly with other principal partners to exchange services, data and knowledge. However, the strategic operation and coordination of the joint venture is delegated to a Joint Management actor, who coordinates tasks and manages the sharing of knowledge and resources. Outside the joint venture, secondary partners supply services or support tasks for the organization core.

The organizational architectural styles have been described in UML, in order to provide detailed representation in architectural phase of Tropos Methodology, as well as to represent the organizational styles into a industrial notation [16].

3. The Gap Between Requirements and Architectural Description

The inter-dependencies and constraints between requirements elements and architectural elements are thus not well-understood and subsequently only little guidance is available in bridging requirements and architecture. The semantic gap between requirements and software design is substantial [12].

Requirements Engineering is concerned with identifying the purpose of a software system, and the contexts in which it will be used. Software architecture is related to the principled study of large grained software components, including their properties, relationships, and pattern of combination [9]. In addition to specifying the structure and topology of the system, the architecture should show the intended correspondence between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, and component compatibility [14].

The existence of conceptual differences between what to do (requirements) versus how to do it

(architecture, design and code) constitutes a semantic gap. Filling this gap requires better models and notations for the intermediate step. There are some critical challenges when trying to reconcile requirements and architectures [8]:

- Requirements are frequently captured informally in a natural language. On the other hand, entities in a software architecture specification are usually specified in a formal manner [11].
- System properties described in non-functional requirements are commonly hard to specify in an architectural model [11].
- Iterative, concurrent evolution of requirements and architectures demands that the development of an architecture be based on incomplete requirements. Also, certain requirements can only be understood after modeling and even partially implementing the system architecture [12].
- Mapping requirements into architectures and maintaining the consistency and traceability between the two is complicated since a single requirement may address multiple architectural concerns and a single architectural element may have numerous non-trivial relations to various requirements.
- Real-world, large-scale systems have to satisfy hundreds, possibly thousands of requirements. It is difficult to identify and refine the architecturally relevant information contained in the requirements due to this scale.
- Requirements and the software architecture emerge in a process involving heterogeneous stakeholders with conflicting goals, expectations, and terminology. Supporting the different stakeholders demands finding the right balance across these divergent interests.

The following section outlines the basis of our approach.

4. The Integrating Framework Proposal

This section describes an informal four-steps process to address the transition between requirements and architectural design. This proposal is a framework to identifying and mapping the architectural decision from a requirements specifications.

4.1. Mapping Architectural Elements from i^*

This proposal focuses on finding a systematic process to support the transition from requirements specification to architectural design.

As showed in Figure 4 the proposal are composed by two modules: i^* *Architectural Extension and*

Integration Process. Our approach for integration process takes as input a goal oriented requirements specifications in i* technique and returns as output an architectural model. The main concerns are related to the identification, classification and support a variety of architectural elements from system requirements.

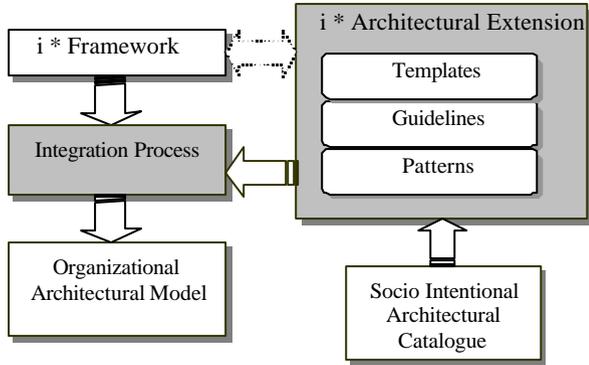


Figure 4 – i* Architectural extension

This extension includes:

- Templates – To extend and refine the properties from i* architectural elements (possibly actors, goals, softgoals resource, task, dependency and links). The identified architectural elements from i* framework are:
 1. Components - The computational elements (possibly systems actors) of the architecture bound together by connectors;
 2. Connections - The relations between components (possibly dependencies between actors or relationships to archive goals, like means-end or task decompositions);
 3. Constraints – assertions and constraints that apply to the entire system or components (possibly extracted from the non-functional requirements, goals, dependency sequences or architectural patterns);
- Guidelines – To support the mapping from SR description into organizational architectural styles elements.
- Architectural Patterns – Compositions or styles in which architectural elements are connected in a particular way. In this work we are using the architectural styles of the socio intentional catalogue (e.g., Joint Venture style).

Figure 5 shows the four-steps Integration Process to mapping and relating i* systems requirements and organizational architectural elements:

- Step 1: Capturing the architectural requirements. This step covers an analysis using as input the i* requirements model and architectural guidelines to identifying architectural elements and capture additional architecture-relevant information. As output we have some templates for architectural elements;

- Step 2: Applying the NFR Framework to select among the socio-intentional architectural style using the non-functional requirements;
- Step 3: Relating i* architectural requirements with the architectural elements from the socio-intentional catalogue applying the guidelines;
- Step 4: Generating the i* architectural model.

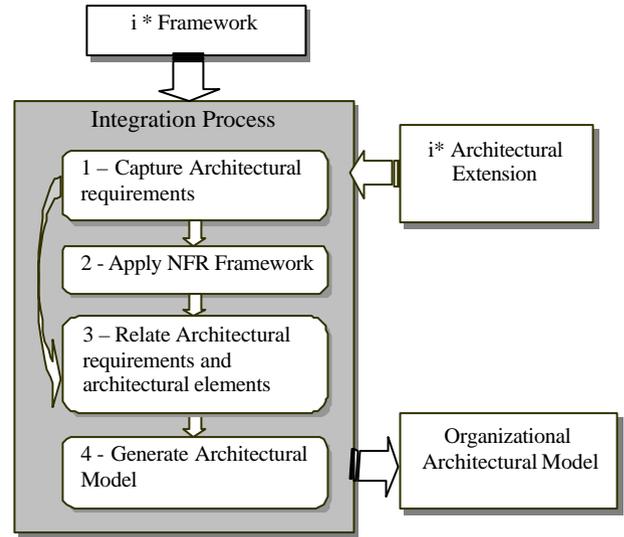


Figure 5 – Integration process

Capture Architectural Requirements - The primary activity is to identify an i* architectural elements composed by requirements elements, showing in Table 1, with complementary architectural definitions.

Table 1 – Mapping the i* architectural elements

I* Elements	Architectural Elements
Actor	<i>System component</i>
Task	<i>Responsibility</i>
Goal	<i>Responsibility/ constraint</i>
Soft-goal	<i>Constraint</i>
Dependency	<i>Connection/Relationship/constraints</i>
Resource	<i>System entity</i>
Link	<i>Connection</i>

In the sequence we show an architectural template example for the component Medi@ system showed in Figure 2.

The Table 2 shows the partial template definition of a component. The *Name* attribute is the i* specification from which the element (actor) is derived. In our example “Medi@” it is a system component. The *Responsibilities* attribute is a list of assignment of system responsibilities (tasks and goals), the sub-components that implement the component. The *Interface* attribute denotes the connectors (dependencies) between others components or sub components. The *Constraints* attribute denotes which

goals the sub-components satisfy, the soft-goals list and architectural style selected.

Table 2 – Architectural templates

<p><i>Type: System Component</i> <i>Name: Medi@</i> Responsibilities: {list of task and goal} Interface: {list of dependencies} <i>Constraints: {Assertions in use, relationship};</i> Architectural Pattern: {organizational style} <i>Composed of: {components}</i> {responsibilities} </p>

The organizational architectures offer a set of design parameters (such direct supervision, standardization of skills, outputs and work processes) that can influence the division of labor and the coordination mechanisms. This design parameters, include, among others task assignments. Tasks are partially ordered sequences of steps intended to accomplish some goal. Tasks can be decomposed into goals and/or subtasks, whose collective fulfillment completes the task. These decompositions also allow to identify actors that can accomplish a goal, carry out a task, or deliver some resource needed by another actor. Fulfillment of an actor’s obligations can be accomplished through delegation and through decomposition of the actor into components actors.

To define the roles in the organizational architectures we propose an initial classification of the responsibilities (tasks and goals) as show in Table 3.

Table 3 – Task type

Basic	The input, processing and output associated with the running the organization
Manager	The coordination and managerial activities
Controller	Standardization of work process
Support	The non-operational services that are outside the basic flow of operational tasks.

Applying NFR Framework - An important task during architectural design is to select among alternative architectural styles using as criteria the desired qualities identified in the previous phase (Late Requirements). They will guide the selection process of the appropriate architectural style. The analysis involves refining these qualities, represented as softgoals, to sub-goals that are more specific and more precise and then evaluating alternative architectural styles against them, as showed in Figure 6.

The analysis resulting in a softgoal dependency graph is intended to make explicit the space of alternatives for fulfilling a top-level attribute. The organizational patterns are represented as operationalized attributes (saying, roughly, “fulfilled by the pattern structure-in-5/joint-venture”) [7].

The evaluation results in contribution relationships from the social structures to the quality attributes, labeled “+”, “++”, “-”, “--” that mean respectively *partially satisfied*, *satisfied*, *partially denied* and *denied*. Design rationale is represented by claims drawn as dashed clouds. They make it possible for domain characteristics such as priorities to be considered and properly reflected into the decision making process. Exclamation marks are used to mark priority attributes while a check-mark “✓” indicates an accepted attribute and a cross “✗” labels a denied attribute.

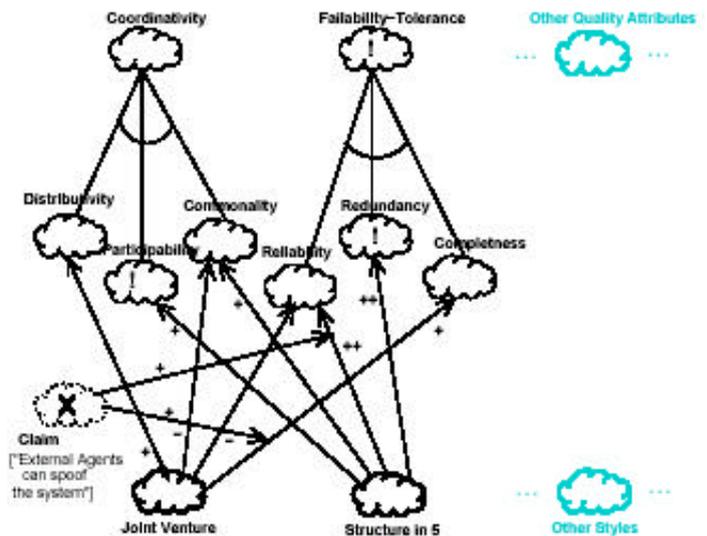


Figure 6 – Partial evaluation for selecting architectural styles

More details about the selection and non-functional requirements decomposition process can be found in [6],[7].

Relating the i* architectural elements and Socio Intentional elements -

The architectural level of design requires a different form of abstraction to reveal high-level structure. In particular, should be possible to represent as first class abstractions new architectural patterns and new forms of interaction between architectural requirements elements, so that the distinct roles of each requirement elements in the structure are clearer.

The organizational pattern adopts the abstractions offered by organizational theory. The structure of an organization defines the roles of various intentional components (actor), their responsibilities, defined in

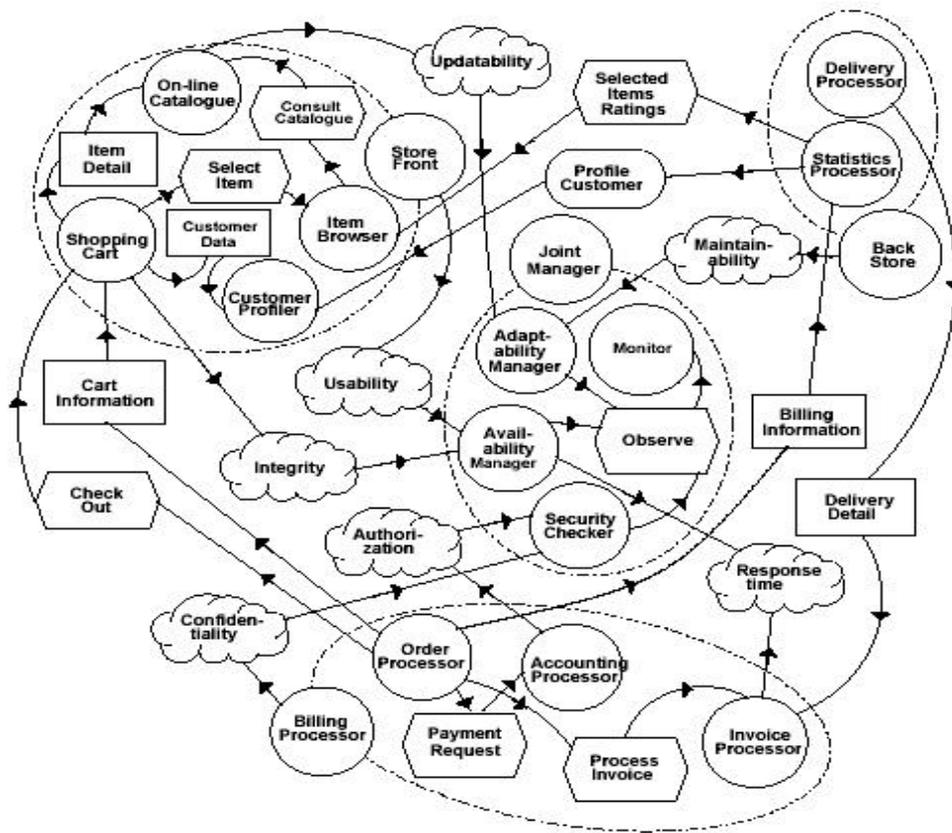


Figure 7 – Medi@ system as joint venture architecture

terms of tasks and goals they have assigned and resources they have been allocated.

A *role* is an abstract characterization of the behaviour of an actor within some specialized context, domain or endeavour. Its characteristics are easily transferable to other actors. Dependencies are associated with a role when these dependencies apply regardless of who plays the role. In order to describing this relationship it is necessary to analyse the responsibilities and roles in the system requirements.

Our work consists of extending the *i** with guidelines to support the mapping of *i** requirements elements to *i** architectural elements.

Guideline 1.1: The *i** systems (or *i** roles) can be mapped to a system component in architectural model.

For instance, the Figure 7 suggest a possible assignment of system responsibilities for the business-to-consumer (B2C) part of Media System. Following the joint venture style, the architecture is decomposed into three principal partner actor (*Store Front, Billing Processor and Back Store*).

Guideline 1.2: The *i** relationship between systems (or roles) can be mapped as interface in architectural model.

The partners control themselves on a local dimension for exchanging, providing and receiving services, data and resources with each other. For

instance, the *Store Front* interacts primarily with the customers and provides them with a usable front-end web application for consulting and shopping media items. See Figure 7.

Guideline 1.3: The *i** task (or goal decomposition into task) can be mapped as responsibility in architectural model.

For instance, some of the responsibilities (see table 1) in Medi@ system are “*Internet Shop Managed*”, “*Secure Form Order*”, “*Internet Orders Handled*”, “*Maintenance*”, as seen in Figure 2.

Guideline 1.4: The *i** tasks-type (or goal-type) defines the roles of various intentional architectural components (actor).

For instance, *Billing Processor* is in charge for the secure management of orders and bills, and other financial data. And the *Joint Manager* manages the system on a global dimension. See Figure 7.

Further guidelines are required to describe a complete mapping between requirements and architecture. Of course not all concepts captured in the requirements phase will correspond to architectural system models. The models do not have a one-one relationship; many elements of the organizational requirements model are not part of the architectural model, since not all of the organizational tasks require a software system. Many tasks contain activities that are performed outside the software system, and so do

not become part of the architectural system model. Likewise, many elements in the architectural model comprise detailed technical software solutions and constructs that are not part of the organizational model.

5. Conclusion

The relationship between requirements and architectures has received increased attention recently [15]. A number of goal-based requirements approaches, most notably KAOS [9] [10] and the NFR framework [13], have proposed the explicit use of the notion of 'goals' to structure system requirements and architecture. A proposal KAOS/APL presented in [15] has suggested the use of intermediate descriptions between requirements and architecture that they call 'architectural prescriptions', which describe the mappings relationship between requirements and architectures. The CBSP approach [8] explores the relationships between software requirements and architectures, and proposes a technique to reconciling mismatches between requirements terminology and concepts with those of architectures.

The purpose of this paper is to present our work on the development of a framework to complement the specification of architectural elements and mapping the relationship between requirements and architectural elements using a set of organizational styles.

Future research directions will extend the architectural catalogue with classical software pattern proposed in the literature (pipes-and-filters, layers, event-based).

6. References

[1] Castro, J., Kolp, M., Mylopoulos, J.: "Towards Requirements Driven Information Systems Engineering: The Tropos Project". In Information Systems, Vol. 27. Elsevier, Amsterdam, The Netherlands (2002) 365–389.

[2] Yu, E.: "Modelling Strategic Relationships for Process Reengineering". Ph.D. thesis, Department of Computer Science, University of Toronto, Canada (1995).

[3] Yu, E., Liu, L.: "Modelling Trust in the i^* Strategic Actors Framework". Proceedings of the 3rd Workshop on Deception, Fraud and Trust in Agent Societies. Barcelona, Spain (at Agents2000), June 3-4, 2000.

[4] Yu, E.: "Agent Orientation as a Modelling Paradigm". *Wirtschaftsinformatik*. 43(2) April 2001. pp. 123-132.

[5] Kolp, M., Castro, J., Mylopoulos, J.: "A social organization perspective on software architectures". In

Proc. of the 1st Int. Workshop From Software Requirements to Architectures. STRAW'01, Toronto, Canada (2001) 5–12.

[6] Kolp, M. and Giorgini, P.: "Information Systems Development through Social Structures". Submitted to the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02), Toronto, Canada, May 2002.

[7] M. Kolp, P. Giorgini and J. Mylopoulos. "Organizational Patterns for Early Requirements Analysis". 15th International Conference on Advanced Information Systems Engineering (CAiSE'03), Velden, Austria, June 2003.

[8] Grünbacher, P., Egyed, A. and Medvidovic, N.: "Reconciling Software Requirements and Architecture: The CBSP Approach". Proceedings RE'01, 5th International Symposium on Requirements Engineering. Toronto, Canada. August 2001.

[9] Lamsweerde, A. van.: "Requirements Engineering in the Year 00: A Research Perspective". 22nd Proceedings of International Conference on Software Engineering, Limerick, Ireland. Jun. 2000.

[10] Lamsweerde, A. van.: "Goal-Oriented Requirements Engineering: A Guided Tour". Proceedings RE'01, 5th International Symposium on Requirements Engineering. Toronto, Canada. August 2001, 249-263.

[11] Medvidovic N., Taylor R.N.: "A Classification and Comparison Framework for Software Architecture Description Languages". *IEEE Transactions on Software Engineering*, 26/1:70-93, 2000.

[12] Nuseibeh, B.: "Weaving the Software Development Process between Requirements and Architectures". First International Workshop From Software Requirements to Architectures (STRAW'01). May, 2001.

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

[14] Shaw, M.: "Abstraction for Software Architecture and Tools to Support Them". *IEEE Transactions on Software Engineering*, 21(4): pp.314-335, April 1995.

[15] STRAW'01. Proceedings of First International Workshop From Software Requirements to Architectures (STRAW'01), 2001. <http://www.cin.ufpe.br/~straw01>.

[16] Silva, C.T.L.L and Castro, J.F.B.: "Detailing Architectural Design in the Tropos Methodology". Proceedings of the 15th Conference on Advanced Information System Engineering – CAiSE 03. Klagenfurt, Velden, Austria, 2003.

Moving from quality attribute requirements to architectural decisions¹

Felix Bachmann, Len Bass, Mark Klein
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pa USA 15213
{fb,ljb,mk}@sei.cmu.edu

Abstract

Quality attribute models are proposed as the linkage between a specification of a quality attribute requirement and a design fragment that is focused on achieving that requirement. Each quality attribute model has a collection of parameters that must be specified in order to determine from the model whether a requirement will be met. These parameters can be bound through design decisions, through values given from a quality requirement, or through knowledge of the designer. Architectural tactics are designed to relate design decisions to control of a quality attribute model parameter in order to achieve particular responses.

In this paper, we present a series of steps that enable moving from a single quality attribute requirement to a design fragment focused on achieving that requirement. We demonstrate these steps through application to an embedded system.

1. Introduction

It is well accepted that the satisfaction of quality attribute requirements for a software system depends heavily on the design of the software architecture for that system. From this a plausible design approach is to use the quality attribute requirements as primary when designing the software architecture. In order for this approach to be successful, four pieces must be in place: precise specification of quality attribute requirements, enumeration of fundamental design approaches to achieve various quality attributes, a linkage between the specification of the requirements and the appropriate design approaches that yields a design fragment focused on achieving the requirement, and a method for composing the design fragments into an actual design.

In this paper, we focus on the third of these pieces: the linkage between a specification of quality attribute

requirements and a design fragment focused on achieving that requirement. We build on our prior work on quality attribute scenarios and architectural tactics and propose the use of quality attribute models as the linkage mechanism. We demonstrate the linkage through deriving a design fragment based on a performance requirement. An application of these steps to an additional modifiability scenario is precluded by space limitations but is available in [2].

We begin by briefly summarizing our prior work in quality attribute scenarios and architectural tactics. We then discuss why quality attribute models are the missing link and how they can be exploited to derive design fragments from quality attribute requirements. We illustrate the linkage through an example of a garage door opener.

2. Quality attribute scenarios

Quality attributes as defined in standards such as ISO 9126 [7] are not adequate for design. This is because the definitions do not reflect the context in which they are applied. For example, all systems are modifiable for some set of changes and not modifiable for others. The key for design is characterizing the set of changes that a particular system will be subjected to. Similar comments hold for other attributes.

We characterize quality attributes through quality attribute scenarios and have used this characterization in the ATAMsm [5] evaluation method as well as in other methods. Our current definition of a quality attribute scenario has 6 parts – stimulus, source of stimulus, environment, artifact being stimulated, response, and response measure. Quality requirements for a particular system can be cast in terms of these six parts.

In Chapter 4 of [3], we present scenario generation tables for the quality attributes of availability, modifiability, performance, security, testability, and

¹ This work supported by the U.S. Department of Defense

usability. Table 1 gives the scenario generation tables for performance – the attribute we use for our illustration. The scenarios generated by the tables in [3] cover most of the common meanings of these attributes [4].

The scenarios generated by these tables are “general” in that they are system independent. In order to make them act as requirements for a particular system, they must be instantiated for that system and made “concrete”.

Portion of scenario	Possible Values
Source	<ul style="list-style-type: none"> – one of a number of independent sources – possibly from within the system
Stimulus	<ul style="list-style-type: none"> – periodic events arrive – sporadic events arrive – stochastic events arrive
Environment	<ul style="list-style-type: none"> – normal conditions – overload conditions
Artifact	<ul style="list-style-type: none"> – System – Process
Response	<ul style="list-style-type: none"> – processes stimuli – changes level of service
Response measure	<ul style="list-style-type: none"> – latency – deadline – throughput – jitter – miss rate – data loss

Table 1: performance scenario generation table

3. Architectural tactics

Experienced architects have a collection of techniques that they use to improve a system response with respect to a particular quality attribute. Some of these techniques are captured in patterns of various sorts but others, such as “reduce computational overhead” or “limit options the system will support”, are not.

We have coined the term “architectural tactic” to describe these techniques and define an architectural tactic as a means of controlling a quality attribute measure by manipulating some aspect of a quality attribute model through architectural design decisions. In Chapter 5 of [3], we provide an enumeration of architectural tactics, albeit with a different definition.

Observe that an architectural tactic is concerned with the relationship between design decisions and a quality-attribute response. This response is usually something that would be specified as a requirement (e.g., an average

latency requirement). Therefore architectural tactics (by definition) are points of leverage for achieving quality-attribute requirements even though, as yet, no guidance is provided as to how to choose appropriate tactics in particular situations.

Table 2 enumerates the architectural tactics used to achieve performance. See [2] for a description of the meaning of each tactic.

Category of tactic	Architectural tactic name
Manage Demand	<ul style="list-style-type: none"> – manage event rate – control frequency of sampling external events – reduce computational overhead – bound execution times – bound queue sizes – increase computational efficiency of algorithms
Arbitrate Demand	<ul style="list-style-type: none"> – increase logical concurrency – determine appropriate scheduling policy – use synchronization protocols
Manage Multiple Resources	<ul style="list-style-type: none"> – increase physical concurrency – balance resource allocation – increase locality of data

Table 2: performance architectural tactics

4. Quality attribute models

Associated with every quality attribute are one or more “reasoning frameworks” that allow prediction of the response of a system with respect to particular attributes. Performance frameworks such as queuing theory or scheduling theory are the best known and studied and they are very quantitative in nature. Frameworks for modifiability include those based on coupling and cohesion [6] and those based on dependency analysis [1]. These are much more qualitative but still allow prediction of the difficulty of a modification. Other frameworks exist for other attributes. Each framework has uncertainty in terms of the accuracy of its predictions but these frameworks have proven useful in assisting designers.

It is these quality attribute reasoning frameworks and their associated models that we exploit to link quality

attribute requirements (specified as concrete quality scenarios) and architectural design decisions (as embodied in architectural tactics).

Every quality attribute reasoning framework has a collection of types of entities that are included in the framework. Performance models, for example, have units of concurrency such as threads or processes, dependency among these units of concurrency, and resources. There is some collection of inputs (arrival rates, resource requirements) that drives the model. We call all of these “parameters” of the models. These are the items that a designer may potentially control to enable the achievement of a desired response.

5. Linking concrete scenarios to architectural tactics

Our goal in this section is to describe how to derive a set of tactics that are relevant for achieving a particular concrete scenario and then use this to derive candidate design fragments. This carried out using the following set of steps. We assume that input to the set of steps is a concrete scenario and some set of already made design decisions exists.

1. *Identify candidate modeling frameworks.* It may be that some of the information from the concrete scenarios will eliminate possible modeling frameworks. For example, if we know that arrivals are periodic then the queuing modeling framework is eliminated from considerations. Each reasoning framework has a collection of parameters that must be set before the reasoning framework can be applied.
2. *Determine bound and free parameters.* The candidate modeling framework has a number of parameters. Some of these may be given by the concrete scenarios and some may be given by elements of the existing design that are not changeable. For example, a concrete scenario may specify “events arrive periodically”. This may require a specific scheduling model. Another element of the existing design might be that a particular operating system is to be used. This determines the execution time associated with processing one event. This is a parameter of the model that is bound. All parameters not bound are considered free.
3. *Enumerate tactics associated with the free parameters.* Because a tactic controls one of the parameters of a model in the reasoning framework, we can list the tactics associated with the free parameters, which we use as candidate tactics for the next steps.
4. *Assign free parameters an initial set of values.* The designer makes an estimate for each free parameter based on intuition or knowledge. If the designer has no intuition or knowledge for a particular parameter

then an arbitrary value might be chosen. If this parameter is important to the system, an implementation of a prototype might be appropriate to get an estimate.

5. *Use tactics to develop satisfactory bindings for all free parameters.* This step has two degrees of freedom – the list of candidate tactics and the set of free parameters. We begin our description by considering the situation where there is only one free parameter. Each of the candidate tactics for this free parameter controls its value – that is, it allows the adjustment of the free parameter. For each candidate tactic, determine whether it can adjust the value of the free parameter to a new value where the solution of the resulting model satisfies the response measure of the concrete parameter. If it can, then it becomes a relevant tactic. If it cannot, then it is discarded. Now consider multiple free parameters. In this situation, we need to consider simultaneously adjusting all free parameters. That is, if tactic one controls parameter 1 and tactic two controls parameter two, we need to determine whether we can move the value for parameter 1 through tactic 1 and the value for parameter 2 through tactic 2 until the dependent variable for a resulting model satisfies the response measure given by the concrete scenario. If we can then we add both tactics to our list of relevant tactics, if we cannot then we discard both tactics. If we have more than one tactic for each parameter, we need to consider all possible combinations of tactics for the parameters.
6. *Allocate responsibilities to architectural elements.* Every tactic enumerated in table 2 has a design fragment assigned, if appropriate. For example one performance tactic suggests using a certain type of scheduler, or a modifiability tactic recommends the use of an intermediary. Applying those fragments to an existing design moves the architecture to a state that supports the scenario, as demonstrated by the modeling framework.

Design fragments come with their own responsibilities and a set of rules that help to:

 - Create/delete/refine design elements
 - Add responsibilities to existing design elements
 - Reallocate responsibilities of already existing design elements
 - Refine responsibilities and allocate them to design elements

For example using the tactic *semantic-importance-based scheduling* includes applying the following rules:

 - Create a design element “scheduler”

- Allocate the responsibilities with higher importance to units of concurrency with higher priority

or using the tactic *break the dependency chain* includes applying the following rules:

- Create a design element “intermediary”
- Add responsibilities to the intermediary that translate from the more abstract interface provided to the secondary modules to the concrete interface provided by the primary module
- Refine the responsibilities of the secondary modules to use the services of the intermediary

6. Garage door example

Our sample design problem is that of a garage door opener. The controller for a garage door opener is an embedded real-time system that reacts to open and close commands from several buttons installed in the house and from a remote control unit, usually located in a car. The controller then controls the speed and direction of the motor, which opens and closes the garage door. The controller also reacts to signals from several sensors attached to the garage door. One of the sensors detects resistance to the movement of the door. If the amount of resistance measured by this sensor is above a certain limit, then the controller interprets this as an obstacle between the garage door and the floor. As a reaction, the motor closing the garage door is stopped.

There are many scenarios that specify the requirements for the controller software. In [2] we present both a performance and a modifiability scenario. Here space limits us to just discussing the performance scenario.

If an obstacle (person or object) is detected by the garage door during descent, it must halt within 0.1 seconds.

We now exemplify our steps for this scenario.

1. Identify candidate reasoning frameworks

There are two performance reasoning frameworks that might be applicable to a performance scenario: queuing theory and scheduling theory. We know from looking at our concrete scenario that we have sporadic event arrivals and a hard deadline requirement. The hard deadline requirement suggests that the applicable reasoning framework is scheduling theory. Sporadic arrivals are arrivals that cannot occur arbitrarily often. This is an indicator that there is a bound on the arrival rate variability, again indicating that scheduling theory is relevant. The other relevant parameters are: execution time, number of units of concurrency, and number of processors.

2. Determine bound and free parameters

In this step the scenario is recast in terms of the bound and free parameters of the applicable reasoning frameworks. Scheduling theory is concerned with calculating worst case latency associated with carrying out each scenario, given the execution time, arrival period associated with each unit of concurrency, the number of units of concurrency and how each unit is allocated to one or more processors. Worst-case latency can then be compared with the hard deadline to determine if the requirement is satisfied or not.

For these parameters we first determine which ones our concrete scenario binds. One parameter is the arrival distribution. In this case the arrival distribution describes how often an obstacle is detected. We assume this happens infrequently and there is a bound on how frequently it occurs (known as a sporadic arrival distribution). We assume from the business context of the garage door opener that a single processor will always be adequate.

Since this is the one performance scenario considered in this example we do not yet have any bound parameters in the selected reasoning framework from previously made decisions.

To summarize:

- Bound parameters: arrival distribution and number of processors
- Free parameters: number of units of concurrency and execution time of responsibilities

3. Enumerate tactics associated with the free parameters

This is where we start to employ our “decision procedures”, which are really a loosely structured set of rules for which tactics to try (see Table 3). In this step the decisions are based strictly on what parameters are considered fixed and which are considered free.

1) Which parameters are fixed?

- Arrival distribution – arrivals are infrequent.
- Number of processors – we will assume that our platform constrains us to a single processor

From the first rule in Table 3 we conclude that the fixed arrival distribution rules out the following tactics: *Manage event rate* and *Control the frequency of sampling external events*

The architect constrains the solution to a single processor because of the business context and this rules out the following tactics: *Increase physical concurrency*, *balance resource allocation*.

2) Which parameters are free?

- Execution time – The responsibilities will suggest a likely range, but this is not yet fixed.
- Number of units of concurrency – This is free and will be determined later in design

The following tactics are concerned with manipulating execution time: *Reduce computational overhead, Increase computation efficiency, Control the demand for resources* and *Bound execution time*

Some of rules of our performance decision procedure that are applicable for this step are shown in Table 3.

Table 3 Example rules of our performance decision procedure

- If the arrival distribution is fixed then *Manage event rate* and *Control the frequency of sampling external events* are not tactics that can be used to control worst-case latency.
- If execution time is a free parameter then consider using the following tactics: *Reduce computational overhead, Increase computation efficiency, Control the demand for resources* and *Bound execution time*
- If the number of processors is bound then eliminate the following tactics as candidates: *Increase physical concurrency* and *balance resource allocation*.

4. Assign free parameters an initial set of values

Two things occur at this step. First, the architect offers his/her best guess for values for the free parameters. The list of applicable tactics suggests factors that impact the setting of these values. Secondly, rules of the decision procedure call attention to possibly problematic situations.

From the previous steps we know that two of the tactics are relevant to estimating execution time: *Reduce computational overhead* and *Bound execution time*. The architect might guess that the sum of the execution time of the 3 responsibilities is about 5 msec. *Bound execution time* calls our attention to the effects of execution time variability, however the architect predicts that these responsibilities have very little variability. *Reduce computational overhead* calls attention to various sources of overhead that represent extra execution time. It is conceivable that each one of the responsibilities involved in obstacle detecting - “detect obstacle”, “determine that garage door is descending”, and “halt garage door” - incur some OS overhead for some pre-selected real-time operating system. Consequently the architect estimates

that the operating system adds an additional 1 msec of overhead. The architect also assumes that all of this scenario’s responsibilities are allocated to a single unit of concurrency. This last assumption is possible because this is the sole scenario considered. We discuss some of the issues involved in multiple scenarios in a further section.

While the architect does not yet know all of the details of the other responsibilities in the system, he or she does know that there will be other responsibilities with associated execution times and these other responsibilities hold potential for adversely affecting the ability of this scenario to be realized. The architect is not yet ready to assign values to the execution times associated with these other responsibilities.

The second consideration at this stage is to examine the scenario to determine if it is unreasonable or problematic. For example, if execution times or arrival rates vary considerably, but deadlines can never be missed, this might be problematic. Examples of rules that call attention such potentially problematic situations are in the table below. However, for the current scenario none of these situations apply.

Some of rules of our performance decision procedure that are applicable for this step are shown in Table 4.

Table 4 More example rules of our performance decision procedure

- If the scenario has a hard deadline response requirement that cannot be and if arrivals can occur arbitrarily close to one another then use one of the following tactics to ensure a lower bound for the inter-arrival interval: *Manage event rate* and *Control sampling frequency*.
- If the scenario has a hard deadline response requirement that cannot be relaxed and if execution times vary considerably to the point that they can approach or exceed the hard deadline then consider applying the following tactic: *Bound execution time*.
- If either of the above “unbounded” conditions apply, but arrival rate and execution time are bound parameters then declare the requirement untenable

5. Use tactics to develop satisfactory bindings for all free parameters

At this point all of the parameters have values and there is a candidate list of applicable tactics. The first thing is to look at one or more of the applicable tactics and apply the reasoning framework (in this case scheduling theory) to determine if the current concrete scenario is satisfied without “violating” any of the scenarios that have already been satisfied.

The relevant tactics entering into this step are:

- Controlling resource demand through *Reduce computational overhead, Increase computation efficiency, Control the demand for resources* and *Bound execution time* have a bearing on how execution time affects worst case latency.

- *Increase logical concurrency* and *determine scheduling policy* both have a bearing on understanding how this scenario's responsibilities affect and/or are affected by the other responsibilities in the system

Without considering the effects of other responsibilities, the model is fairly simple. The only contributors to latency are execution time and overhead, 5 msec and 1 msec respectively. Their sum is well under the deadline of 100 msec (that is, .1seconds), leaving 94 msec to spare.

On the other hand it is very conceivable that the other responsibilities in the system take more than 94 msec. Using the last rule in the table below suggests that the design decisions made in the next step be consistent with our simple model, that is, they ensure that the latency associated with this scenario's responsibilities is not affected by any of the other responsibilities.

Some of rules of our performance decision procedure that are applicable so far for this step are shown in Table 5.

Table 5 More example rules of our performance decision procedure

- If the execution time associated with the arrival is close to the deadline consider reducing execution time by using the following tactics: *Reduce overhead, Bound execution time, and/or Increase computation efficiency.*

- If the difference between the worst and best case is significant then review the following tactics and apply their modeling techniques to assess miss rates and average latency respectively: *Bound execution times* and/or *Bound queue sizes*

- If the response requirement for all scenarios can be achieved even with the worst-case delay due to all of the other responsibilities of all of the other scenarios, then use any the following tactics:

- Allocate responsibilities to one of the existing units of concurrency
 - *Offline scheduling*
- Or allocate responsibilities to a new unit

of concurrency

- *Increase logical concurrency*
- *Time-based scheduling*

If the current scenario cannot suffer the worst-case delay due to some or all of the other responsibilities then consider them to be time-sensitive and use the following tactics to create an appropriate scheduling policy: *Offline scheduling, Time-based scheduling (such as deadline monotonic scheduling), and/or Increase logical concurrency.*

Up to now in this step tactics have been used to set and/or adjust model parameters to satisfy the current concrete scenario's response measure. However, it might be the case that either the scenario poses an untenable requirement or the collection of scenarios considered up to this point are untenable in aggregate. If this is the case, tactics should offer some ideas for how to relax requirements or design constraints.

Some of rules of our performance decision procedure that are useful for identify and relaxing requirements and/or design constraints are shown in Table 6.

Table 6 Some of rules for relaxing requirements and/or design constraints

- If the response requirement is specified as a hard but limited misses can actually be tolerated then re-characterize deadlines as follows:

- Firm deadlines: Completing before the deadline is very important. Missing occasionally can be tolerated. A specific bound on miss rate needs to be specified.
- Soft deadlines: In this case the term "deadline" is a misnomer. A specification of an average latency requirement is what is needed.

- If the time-sensitive set of responsibilities is not schedulable then incorporate a notion of importance-based scheduling to handle overload situations using *Semantic-importance-based scheduling* or add more resource using *Increase physical concurrency.*

6. Allocate responsibilities to architectural elements

Each tactic will suggest associated design fragments. The tactics of primary concern so far in this example are:

- Controlling resource demand through *Reduce computational overhead, Increase computation efficiency, Control the demand for resources* and *Bound execution time* have a bearing on how execution time affects worst case latency.

- *Increase logical concurrency* and *determine scheduling policy* both have a bearing on

understanding how this scenario's responsibilities affect and/or are affected by the other responsibilities in the system

Reducing computational overhead can map to many design decisions such as:

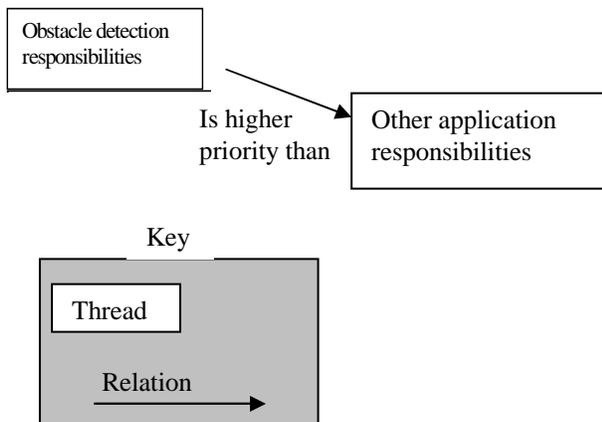
- choice of operating system
- choice of operating system services used in implementing responsibilities
- choice of communication mechanisms, ...

We have accounted for OS responsibilities by assuming 1 msec overhead. We have also assumed that all of the scenario's responsibilities have been allocated to a single unit of concurrency that we will assume is a thread.

The responsibilities for this scenario are pretty straightforward;

Tactics of *Increase computation efficiency*, *Control the demand for resources* and *Bound execution time* are likely not relevant whereas tactics of *Increase logical concurrency* and *determine scheduling policy* are relevant. They suggest allocating the obstacle detection responsibilities to a particular module under one thread of control and assigning this thread a suitably high scheduling priority. This results in an design fragment with two threads: the one containing the obstacle detection responsibilities and the one containing other responsibilities. We do not show the scheduler (which is part of the OS) although that also is a portion of the design fragment. We show a component and connector view of this fragment in Figure 1.

Figure 1: Design fragment



In an ideal world obstacle detection will take only as long as it takes to execute the obstacle detection

responsibilities plus a little overhead. However, the possibility exists that properties of the *Other responsibilities*, such as non-preemptability or execution within an interrupt handler might not have been accounted for. Therefore these potentially problematic properties need to be discovered and/or ruled out. We expect that rules in the *time-based scheduling* tactics would cause us to look for and/or ensure against such properties

Observe the relationship between the design fragment and the associated analysis model. The model states that obstacle detection responsibilities must be scheduled with a priority high than other responsibilities. The design fragment captures this by placing these responsibilities into separate threads and showing the priority relationships of those threads.

7. Composition

We have shown how to use the tactics to link one quality attribute performance requirement to a design fragment with active assistance from an architect. The gaping open issue is what happens with multiple scenarios involving multiple quality requirements, especially for other attributes. How to compose the design fragments into a design is the fourth step of moving from quality requirements to design and it must clearly be solved for this approach to be successful.

Some of the problems that must be solved to achieve the composition of design fragments into designs are:

- How to consider the impact of design decisions already made.
- How to choose among the myriad of possibilities of composing design fragments. In [2] we identified a design fragment for modifiability as well as one for performance and there were multiple composition possibilities
- How to maintain view consistency. Each quality attribute framework has a vocabulary that maps into one or more software architecture views. Maintaining consistency between fragments that come from one reasoning framework with those that come from another is a problem that must be solved.

8. Other open issues and conclusions

In addition to the composition problems there are two other problems that must be overcome.

1. What is the availability and utility of the various reasoning frameworks for other quality attributes? Involving the architect, as we did, in the design process allows judgment to be used in application of the reasoning frameworks. We can predict that, over time, reasoning frameworks for various quality attributes will improve.

2. How do the steps we have presented here become embedded into a design method? Once quality requirements become recognized as important to design they will begin to be specified in the 1000s as are functional requirements. This leads to over specification of the requirements. A design method must be sensitive to this over specification.

Regardless of the problems, focusing on quality attribute requirements and using them to drive towards an appropriate architectural design must be a useful approach. The utilization of quality attribute models and tactics in this process is our attempt to move design toward a more scientific basis.

9. References

- [1] Bachmann, F., Bass, L., and Klein, M. Illuminating the fundamental contributors to Software Architecture Quality. CMU/SEI-2002-TR-025
- [2] Bachmann, F., Bass, L., and Klein, M. Deriving Architectural Tactics – A Step toward Methodical Architectural Design CMU/SEI-2003-TR-004

[3] Bass, L., Clements, P. and Kazman, R. Software Architecture in Practice, 2nd edition. 2003, Addison-Wesley.

[4] Bass, L., Klein, M., Moreno, G., Applicability of General Scenarios to the Architecture Tradeoff Analysis Method. CMU/SEI-2001-TR-014

[5] Clements, P., Kazman, R., and Klein, M., Evaluating Software Architectures. Addison Wesley, 2002

[6] Henry, S. and Kafura, D. Software Metrics Based on Information Flow. IEEE Transactions on Software Engineering. SE-7(5), Sept. 1981

[7] . *International Standard ISO/IEC 9126. Information technology -- Software product evaluation -- Quality characteristics and guidelines for their use*, International Organization for Standardization, International Electrotechnical Commission, Geneva.

Using Problem Frames with Distributed Architectures: A Case for Cardinality on Interfaces

Charles B. Haley

The American University of Paris, Paris, France

The Open University, Milton Keynes, UK

charles [at] the-haleys.com

Abstract

Certain classes of problems amenable to description using Problem Frames, in particular ones intended to be implemented using a distributed architecture, can benefit by the addition of a cardinality specification on the domain interfaces. This paper presents an example of such a problem, demonstrates the need for relationship cardinality, and proposes a notation to represent cardinality on domain interfaces.

1. Introduction

In a Problem Frames analysis [3, 4], *domains* share *phenomena* at their *interfaces*. One of the domains in the analysis is the *machine domain*, which represents the software to be constructed by the developer. Phenomena are the externally visible characteristics of the domains. The phenomena visible at the machine domain's interfaces drive much of the analysis process.

The existence of certain phenomena can be predetermined by purchased products to be used in the system [1] or by considering architectural implications early in the requirements cycle [6, 7]. Hall et al [2] argued for extending Problem Frames to take architectural considerations *within* the machine domain into account, thus incorporating domain knowledge into the analysis. This paper takes the argument one step further, arguing that there are architectural considerations that affect the propagation of phenomena *between* domains, and that it is helpful to explicitly note these considerations in the diagrams.

In a 'standard' Problem Frames analysis, phenomena are considered shared and instantaneous. All domains that participate in a given interface share the phenomena; participation is a relationship. The question of cardinality of the relationship does not arise, because the phenomena are always shared by all. However, a class of problems exists wherein it is convenient to define more precisely

how phenomena are shared over an interface. The case comes up when the implementation of a system is to contain redundancy or be partitioned into semi-autonomous units, such as what occurs when using a distributed architecture. The originating domains may need to know about how phenomena are propagated, either for correctness or for efficiency. Using explicit *connection domains* can resolve the problem, but they introduce complexity. The author argues that by noting *cardinality* on the interfaces, appropriate information can be included in the analysis without a significant increase in complexity.

Section 2 of this paper describe a small lighting control system using Problem Frames. Section 3 presents one possible implementation, showing a case where the current shared phenomena notions do not expose certain difficulties. Section 4 proposes an extension to Problem Frames notation to correct the problem, and Section 5 presents conclusions.

2. The Lighting System

2.1. The Problem Statement

A lighting control system is to be built that conforms to the following problem statement, provided by the firm constructing the building.

The architect wishes to have a lighting control system for a building. From the user's perspective, the system consists of switches and lighting units (lights) associated with a room. When a user actuates a switch, the associated light or lights in the room are turned on or off.

The architect requires the use of up/down momentary contact switches. A momentary contact switch must cause its lighting units to change to the state indicated by the switch's motion, if needed: up turns the lights on if they are not already on and down turns the lights off if they are not already off.

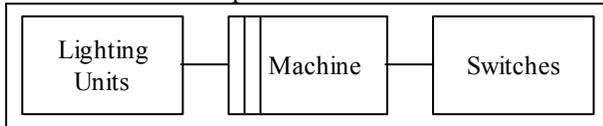
The system is to be built using networked components and to include redundancy where appropriate.

Discussions with the architect and the vendors of the lighting equipment establish the following facts:

1. Switches and lighting units are connected by a network. They are not able to converse directly with each other.
2. A *room* is a logical concept, covering from part of a 'real room' to multiple floors of a building.

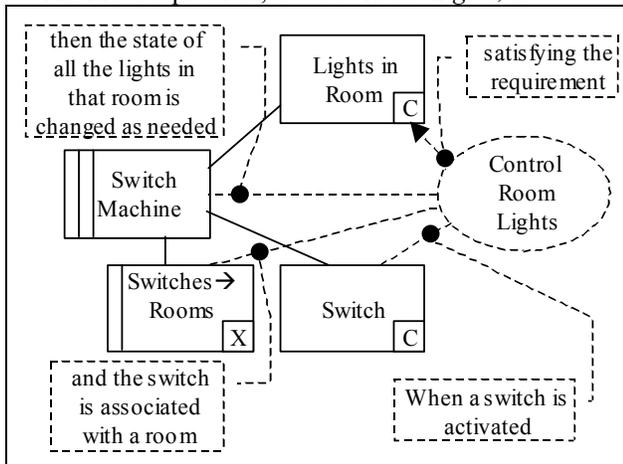
2.2. The Problem Diagrams

The following is the context diagram for the environment. It appears to describe a straightforward commanded behavior problem.

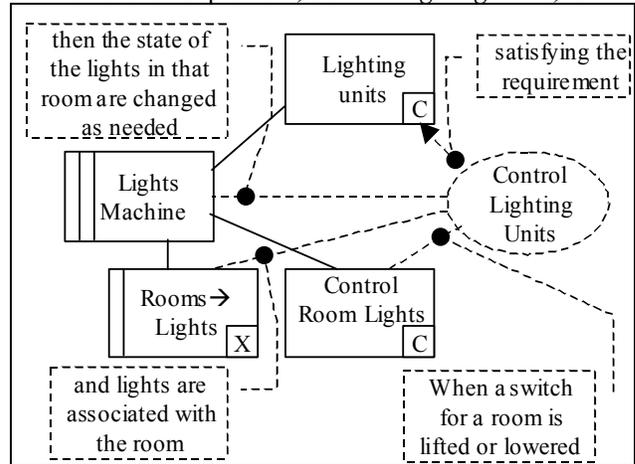


The problem decomposes into two commanded behavior subproblems¹. The first maps *switch events* to the rooms that they control, using a lexical domain as a *Switches* → *Rooms* map. The second maps *room events* to the lighting units in that room, using a *Rooms* → *Lights* map.

The first subproblem, *Control Room Lights*, is:



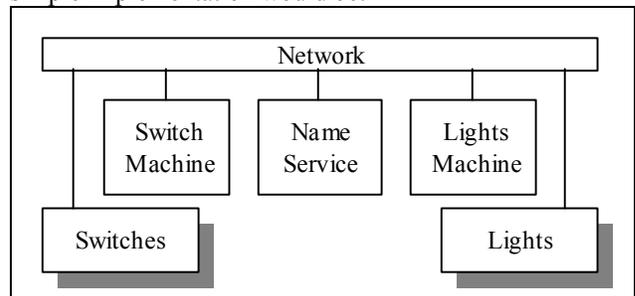
The second subproblem, *Control Lighting Units*, is:



Looking at the diagrams, we see that lifting or lowering a switch causes an event that is a phenomenon shared with the Switch Machine. The machine determines which logical room is to have its lights changed, and is the source of a phenomenon shared with the Lights Machine, as shown in the second diagram. The second machine determines which lighting units are involved, and then is the source of phenomena shared with the appropriate lighting units.

3. A Possible Implementation

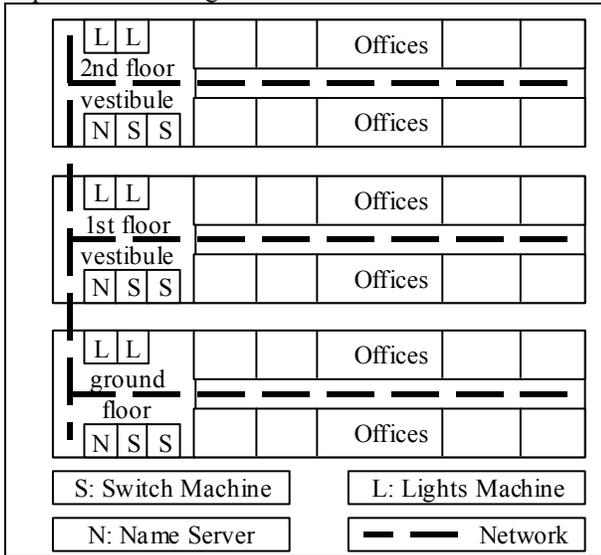
One can imagine constructing this system using a Jini-like distributed architecture [5]. In a Jini-based system, when a switch is actuated it uses a *name service* to find an appropriate service to process the event. Maintaining correspondence with the problem diagrams, the switch will next find the *switch machine*. The switch machine will use its map to determine which rooms need to know about the switch actuation, and then use the name service to find the *lights machine* to contact. A diagram of a simple implementation would be:



If we consider the name service to be part of the network, then the above implementation corresponds very closely to the subproblem diagrams.

¹ The simple workpiece problems needed to maintain the lexical domains are not discussed in this paper.

However, one might choose a different implementation for a larger building. If the building has multiple floors, then for performance we might put switches and lights machines on each floor. To improve reliability, we might put multiple machines of the same type on each floor, where any instance of a machine type can substitute for any other (i.e. introduce redundancy). Such an implementation might look like:



To complicate things a bit more, assume the existence of a logical room consisting of lights in all three of the vestibules.

Assume that the architect specifies the following two rules:

1. A switch on a given floor can select either of the switch machines on its floor, choosing at random. If that machine does not answer, another machine is tried.
2. Either of the lights servers on a floor can control the lights on that floor. The server to use is chosen at random. If that machine does not answer, another server is tried.

Therefore, when a user lifts a *vestibule switch* on the ground floor, the switch chooses either of the *switch servers* on the ground floor. That switch server subsequently must contact either one of the two light servers on each floor, requesting that the lights be turned on.

The problem diagrams shown in Section 2.2 do not express the added complexity of the multiple servers, and thus it is difficult to reason about the system's behavior under certain conditions. For example, analyzing the effects of particular concerns such as initialization, fault recovery, and component maintenance pose problems. Adding explicit connection domain subproblems to the

problem can show the missing behavior, but the domains also add significant additional complexity.

4. Extension of Problem Frames Notation

The deficiency in Problem Frames notation exposed by the above example is the inability to accurately specify a *limited many* relationship on an interface. In the example, from the point of view of the switch there are many candidates for the switch machine, but only one of them is to be used. From the point of view of the switch machine, there are many candidate lights machines, where potentially many of them are to be used. These relationships have a form of *cardinality*.

Relationships on an interface are directed. All phenomena have a source domain and some number of destination domains. From the point of view of a source or a destination, there can be from one to N domains on the other side of the relation. Thus, the cardinality of a relationship can be described as follows:

$N(b) \rightarrow M(c)$: there are N sources of phenomena on an interface where b sources are to be considered interchangeable, and M destinations for the phenomena where c destinations participate.

For convenience, if the parenthesized portion is omitted, it is assumed to be identical to the number that would be in front of it. Thus $1 \rightarrow N$ is the same as $1(1) \rightarrow N(N)$.

Referring to the more complicated example above, the cardinality of the *switch to switch machine* interface is $N(1) \rightarrow 2(1)$. The left side is $N(1)$ because only one of the N switches participates in a given switch actuation. However, the example specifies that there are two interchangeable switch machines available to the switch, and the switch must choose which one to use. Thus, the cardinality of the switch machine is $2(1)$.

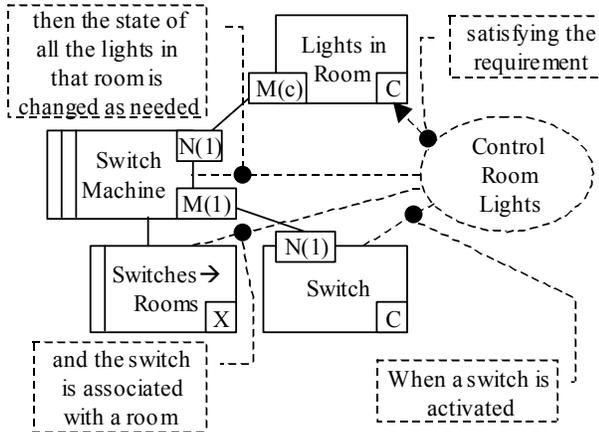
Still referring to the example, the cardinality of the *switch machine to lights machine* interface is $6(1) \rightarrow 6(3)$. There are six switch machines on three floors, but only one of them can be the source of a phenomenon on the interface. There are three groups of two identical light machines, thus three of them participate as destinations of a phenomenon.

Finishing the example, we see that the cardinality of the *lights machine to lighting units* interface is $2(1) \rightarrow M(M)$ (or $2(1) \rightarrow M$). Two lights machines can share phenomena with any given lighting unit, but only one at a time. Each lighting unit is an individual, meaning that all M lighting units must share phenomena with the given lights machine.

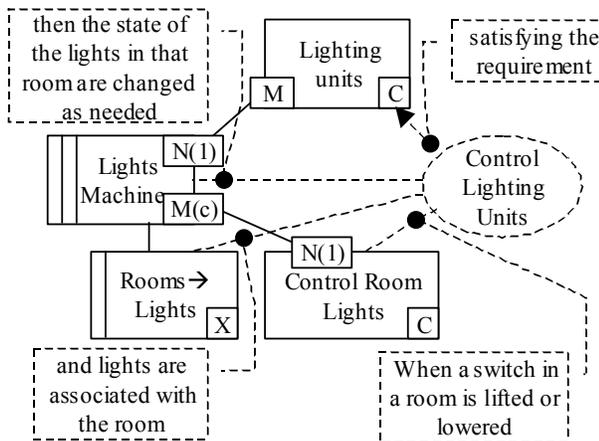
Clearly one would not use such specific notations on a problem diagram unless the numbers are fixed in the problem statement, which is not the case in this example. The *switches to switch machine* cardinality is better

written as $N(1) \rightarrow M(1)$. The *switch machine to lights machine* cardinality is $N(1) \rightarrow M(c \text{ s.t. } c \leq M)$ and the *lights machine to lighting units* is $N(1) \rightarrow M$.

Applying these cardinality notes to the subproblem diagrams, we arrive at:



and



5. Conclusions

Adding cardinality notations to Problem Frames diagrams conveys information about how phenomena are to propagate. The engineers responsible for implementing the system would use this information to ensure that the system behaves as desired and to verify correctness in the face of errors, such as partial loss of power and machine failure. Using cardinality avoids the complexity of adding connection domains to provide equivalent information.

References

1. B. Boehm, "Requirements That Handle IKIWISI, COTS, and Rapid Change," *IEEE Computer*, vol. 33, no. 7, Jul, pp. 99-102, 2000.
2. J. G. Hall, M. Jackson, R. Laney, B. Nuseibeh, and L. Rapanotti, "Relating Software Requirements and Architectures Using Problem Frames," in *Proceedings of the IEEE Joint International Requirements Engineering Conference (RE'02)*. Essen, Germany, 9-13 Sep 2002.
3. M. Jackson, *Software Requirements & Specifications*. Addison Wesley, 1995.
4. M. Jackson, *Problem Frames*. Addison Wesley, 2001.
5. "Jini Network Technology," <http://www.sun.com/software/jini/>. Sun Microsystems, 1999-2002. (31/01/2003).
6. B. Nuseibeh, "Weaving the Software Development Process Between Requirements and Architecture," in *From Software Requirements to Architectures (STRAW '01)*. 23rd International Conference on Software Engineering, ICSE 2001. Toronto, Ontario, Canada, 12-19 May, 2001.
7. B. Nuseibeh, "Weaving Together Requirements and Architectures," *IEEE Computer*, vol. 24, no. 3, March, pp. 115-119, 2001.

Towards a Systems Engineering Pattern Language: Applying *i** to Model Requirements-Architecture Patterns

P. Pavan, N.A.M. Maiden & X. Zhu

Centre for HCI Design, City University
Northampton Square, London UK

Abstract

*This paper reports the results of exploratory research to develop a pilot pattern language for systems engineers at BAE SYSTEMS. The pattern language was designed to encapsulate knowledge about possible trade-offs made by systems engineers about architecture designs that satisfied different system requirements for submarine manoeuvring systems. Our intention is that this knowledge can be reused in future systems engineering processes using our ART-SCENE environment. Knowledge about requirements, design alternatives and the complex trade-off space was elicited from systems engineers. To model this knowledge we applied the *i** formalism to represent the design space and design trade-offs, and to communicate the resulting patterns back to the engineers for validation and improvement. The research was a success, in that we produced a pattern language of 4 key patterns and their interactions for a submarine manoeuvring system, all using the *i** formalism. The paper ends with a review of this research and how we plan to exploit the language to inform scenario-driven trade-offs between requirements satisfaction and architecture choice using the ART-SCENE environment.*

1. Patterns of Patterns – Linking Requirements and Architectures

There is increasing recognition of the need for systems engineers to link system requirements and architecture designs. Considerable research is being undertaken into a range of topics – from tracing architectural decisions to requirements to relating architectural patterns to requirements patterns (Jackson 1995) and formal foundations of the requirements-architecture relationship (Hall et al. 2002). However this research tends not to investigate the systems engineering processes that its results are intended to support. Rather we argue that requirements-architecture research must be based on sound process models of concurrent requirements-architecture engineering. Using one such process model, this paper presents a novel pattern-based approach for exploring

requirements-architecture trade-offs, and introduces an innovative pattern language that underpins such trade-offs.

Our ART-SCENE (Analysing Requirements Trade-offs – Scenario Evaluations) approach advocates a process in which systems engineers and stakeholders concurrently:

1. Generate and walk through system-level scenarios using our established CREWS-SAVRE process and software tools (Sutcliffe et al. 1998);
2. Acquire stakeholder requirements using the ACRE framework (Maiden & Rugg 1996) and model them using the *i** formalism (Chung et al. 2000) with our REDEPEND tool (Maiden et al. 2002);
3. Model candidate system architectures using object-oriented modelling techniques supported by the AUTOFOCUS software tool (Huber et al. 1998);
4. Trade-off the satisfaction of different requirements by different architecture designs using scenarios to link requirements and architectures, then to simulate system and agent behaviours to compute their outcomes (Zhu et al. 2003).

A simple overview of the process is shown in Figure 1. Systematic scenario walkthroughs lead to the acquisition of more complete stakeholder requirements (Maiden et al. 2003). Candidate system architectures lead to the rejection of stakeholder requirements that are not viable. Simulations of models of candidate system architectures using the scenarios compute the emergent properties of the system that can then be tested for compliance with the measurable fit criteria of stakeholder requirements (Robertson & Robertson 1999). Based on these processes we are developing a suite of integrated software tools that, we hope, will provide systems engineers with an effective plug-and-play environment for exploring requirements-architecture trade-offs in the presence of system scenarios. The ART-SCENE software tools and techniques developed to implement these processes are described elsewhere (Zhu et al. 2003, Maiden et al. 2002).

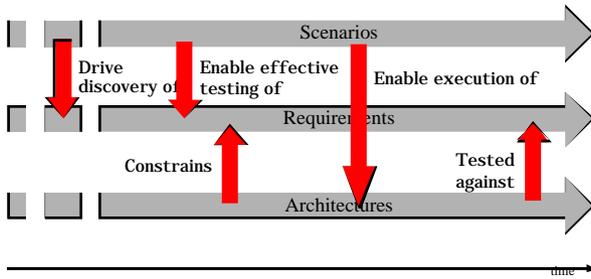


Figure 1. Simplified ART-SCENE process from slide

In this paper we report research that led to one of ART-SCENE's features to support this process – the use of systems engineering patterns to explore requirement-architecture trade-offs in the presence of scenarios. The paper describes how we overcame 3 challenges to produce a simple pattern language produced in collaboration with system engineers at BAE SYSTEMS as part of the UK EPSRC-funded SIMP project:

1. How to elicit a pattern language that could be applied to inform the essential task of making requirement-architecture trade-offs;
2. How to model the patterns using a formalism that was sufficiently expressive and computational;
3. How to implement the resulting pattern language in the ART-SCENE environment to inform requirement-architecture trade-offs.

The remainder of this paper is in 4 sections. Section 2 reports previous research in systems patterns undertaken by the authors which shapes the direction of the reported research. Section 3 describes how we elicited the patterns and applied the i^* formalism to model them. Section 4 present a pattern in detail and reviews the success of the elicitation exercise. The last section reports planned future use of such pattern languages in the ART-SCENE environment, as well as implications for future such pattern modeling in system and software engineering.

2. Researching Requirements Engineering Patterns: Lessons from the Trenches

There has been considerable recent interest in relating classes of problem domains (i.e. requirements) to classes of software solutions (i.e. architectures and designs). Jackson (1995), for example, advocates problem frames, a generalisation of a class of problem consisting of the principal problem parts and a solution task. Hall et al. (2002) extend these problem frames by linking them to simple system functions. Elsewhere Konrad & Cheng (2002) present system specification patterns using an object-oriented specification. Earlier work on requirements clichés (Reubenstein & Waters 1991), patterns (Coad et al. 1995, Buschmann et al.

1996) and generalised application frames (Constantopoulos 1991) are all examples of the research undertaken previously in this area. However, our experience in research area suggests major pitfalls await researchers who fail to learn from these past experiences.

The ESPRIT III 6353 'NATURE' basic research action, in which the authors were partners, undertook between 1992 and 1995 one of most comprehensive classifications of problem domains and software architecture styles to date (Sutcliffe & Maiden 1998). The NATURE approach argued that most requirements engineering problem domains are instances of a tractable set of object system models. Each model contains general features shared by all instances of that problem domain. For instance, one model contains general features of all resource hiring problem domains, examples of which are lending libraries, car rental and video hiring. Another contains general features of all object sensing problem domains. NATURE produced the first extensive categorisation of requirements engineering problem domains and derived a set of over 200 object system models (Sutcliffe & Maiden 1998) from domain analysis, case studies and textbooks.

Object system models were defined in a hierarchical class structure. The 13 highest-level object system models define the fundamental state transitions, sequences of transitions, states and objects in categories of problem domains, and indicate the breadth of the models specified. These models (with a prototypical example of each) are *resource returning* (e.g. car rental), *resource supplying* (e.g. order purchasing), *resource usage* (e.g. sales orders), *item composition* (e.g. goods manufacturing), *item decomposition* (e.g. unpacking deliveries), *resource allocation* (e.g. production planning), *logistics* (e.g. complex production scheduling), *object sensing* (e.g. aircraft detection), *object messaging* (e.g. electronic mail), *agent-object control* (e.g. air traffic control), *domain simulation* (e.g. cockpit simulation), *workpiece manipulation* (e.g. text processing) and *object reading* (e.g. a town's computerised information point).

Specialisation of these high-level object system models is achieved by adding different dimensions, in the form of fact types, at different levels in the hierarchies. For example, the specialisation of the object system model for object sensing generates a large number of lower-level, more detailed hierarchical object system models. The top-level model, which describes the sensing of an object by a sensor agent, is specialised at level-1 according to whether the sensor is sensing the physical location (e.g. position) or internal state (e.g. temperature) of the object, and whether there are one or numerous objects to sense. NATURE specialises each level-1 model further using:

- Different goal states (detect forbidden state, warn if forbidden state arises, forbid state to arise, etc);

- Different events and stative conditions on state transitions (e.g. a monitoring agent blocks an object from changing to a forbidden state);
- Different object and agent types (e.g. physical, conceptual and financial).

Such specialisation gave rise to over 30 level-4 models that are sub-classes of the original object sensing model alone.

We validated NATURE's object system models against natural mental categories elicited using card sorts with experienced software engineers. Results of this empirical validation led to some revision of the structure and contents of several models and how these models might be retrieved and used (Maiden & Hare 1998). Furthermore, to relate these classes of problem domain to software solutions, we developed an orthogonal classification of information system models that we linked to object system models to represent candidate information system solutions for different problem domain classes (Sutcliffe & Maiden 1998). To exploit the library of object system models we developed computational models of analogical reasoning (Maiden & Sutcliffe 1996a, 1996b) to retrieve object and information system models that matched a new application to enable reuse of knowledge about the problem domain and possible information system solutions to it. This reuse-driven approach to requirements engineering and high-level design was validated using several application case studies.

So, what conclusions did we draw from this extensive 5-year programme of research? Although the results of the basic research provides important insights into the nature of abstraction and classifications of problem domains, the more direct benefits to systems engineering were limited. Lessons learned included:

1. The object system models encapsulate problem domain knowledge that is often already known and accessible to systems engineers;
2. NATURE's classification of problem domains was too fine-grain for cost-effective reuse – most applications were an aggregation of instances of a large number of object system model classes, which made model retrieval and instantiation difficult;
3. Systems engineers gained little from directly reusing small fragments of knowledge from the object and information system models – indeed the emergence of large business reference models implemented in successful ERP solutions such as SAP R/3 (Curran & Ladd 1998) suggests that reuse of large, domain-specific models tend to be more effective;
4. Linking the object and information system models did not provide systems engineers with useful knowledge with which to make requirements-driven architectural decisions. The information

system models, by their definition, described functions, classes and structures rather than the non-functional requirements and quality attributes essential to such decision-making (e.g. Franch & Carvalho 2003). The models did not capture the required richness and context or different design alternatives and their comparison.

These experiences directed us to a different approach to modeling patterns in systems engineering – one that captures the essence of architecture design alternatives in terms of how they satisfy a number of related system requirements. As an input to the ART-SCENE environment we sought to develop a pilot pattern language to do just that to inform requirements-architecture trade-offs. The remainder of this paper reports the results of research motivated to determine such patterns.

3. Eliciting Systems Engineering Patterns in ART-SCENE

Our research uses Alexander's (1979) original definition of a pattern as a solution to a problem in a context of use. Alexander defines a *pattern* as a three-part construct:

1. The **context** - conditions under which the pattern holds;
2. A **system of forces** - the 'problem' or 'goal' that the solution solves;
3. The **solution** – a configuration that balances the system of forces and solves the problem.

This definition contrasts markedly with the nature of most software and systems engineering patterns reported earlier. The patterns, from requirements cliches to problem frames, all use a weaker definition of the problem or goal without an explicit description of a system of forces that describes it. Likewise most patterns describe a single reusable solution without explaining how the configuration implemented in the solution satisfies the goal or solves the problem. One exception is reported in Gross & Yu (2001) who highlighted the need for a modeling approach that supports how business goals relate to the architectural decision-making process, and how changing business goals give rise to alternative architectural choices and solution structures. They also showed how the need to describe organisational stakeholders, their goals and how these are affected by alternative choices during the design process using agents and goals. 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.

So how can we apply Alexander's pattern definition to a modern systems engineering process? First of all we need to map Alexander's 3 parts of the pattern to systems engineering models and artefacts. We assert that:

- The **system of forces** represents a set of interconnected system requirements that a designed architecture configuration must satisfy;
- The **solution** represents the architecture design and to what degree that design alternative satisfies each system requirement;
- The **context** represents the conditions under which the pattern holds – that is the project or problem environment in which the solution applies. In ART-SCENE we equate the problem environment to one or more scenarios in which the architecture design must satisfy the system requirements.

Therefore the systems engineering patterns in ART-SCENE encapsulate knowledge about important design decisions that sought to balance the satisfaction of competing requirements in different scenarios for a previous but relevant system. To elicit and model these patterns we designed and applied a rigorous method described in the next section.

3.1. Pattern Elicitation and Modeling Method

We developed the pilot pattern language with BAE SYSTEMS, one of our partners in the EPSRC-funded SIMP project. Our objectives were to model patterns in a domain that was complex but could be understood by the academic researchers, did not require high-level security access, and could be scoped in order to provide results within the time frame of the exercise. The result was a decision to develop a pattern language for submarine manoeuvring systems – that is the systems that enable a naval submarine to steer when under water.

We elicited pattern knowledge from BAE SYSTEMS engineers in 3 phases:

1. Discover and elaborate key design decisions made on previous projects;
2. Model and validate the context, solution and system of forces for each pattern;
3. Elicit, model and validate the key relationships between the patterns established in the first 3 phases to produce the first-cut pattern language.

Each elicitation session took place with 2 systems engineers with shared engineering experience of the submarine manoeuvring system. Throughout each session we encouraged the systems engineers to converse with each other. This technique, known as constructive interaction (Miyake 1986), overcomes the unnatural aspects other elicitation techniques and provided supplementary data about the patterns at each phase.

In the first phase we combined brainstorming with semi-structured interviews to discover and prioritise previous design decisions made about manoeuvring systems. We used the interview structure to elicit data about different candidate architecture designs, why each was chosen or rejected, and conditions for its use.

All data was recorded on flipchart sheets as informal sketches and written notes.

In the second phase we used the data to describe each pattern with the following attributes:

Name: A unique and meaningful pattern name;

Authors: The main contributors to the pattern;

Problem: The main trade-offs to be made and the different forces to be balanced to achieve an acceptable solution;

Principle: The principle behind the pattern;

Context: The pre-conditions under which the problem and its solution seem to recur, and for which the solution is desirable;

Forces: A definition of the relevant *forces*;

Solution: Descriptions of architecture solution that can be reused;

Rationale: A justification of the solution and the pattern as a whole in terms of how it resolves its forces to be in line with the desired outcome;

Known Uses: Known occurrences of the pattern and its application within existing systems;

Models: The *i** models that describe the pattern;

Further questions: Questions that need answering in order to further refine the pattern.

One innovation was to produce *i** models (Chung et al. 2000) for each pattern. Inspired by the earlier use of *i** to model requirement-architecture patterns (Gross & Yu 2001), we chose the *i** formalism to model our pattern language for 3 reasons:

1. *i** SD (Strategic Dependency) models allowed us to model each pattern as a network of dependency relationships among actors characteristic of large socio-technical systems – the types of system found in submarine design;
2. *i** SR (Strategic Rationale) models allowed us to model how different candidate **solutions**, represented as tasks in *i**, satisfied different actor goals and soft goals using *i** means-ends links – Alexander's **systems of forces**;
3. *i** contributes-to soft goal links in the SR models allowed us to represent complex trade-offs between requirements that occur when making choices about one solution over another – the **systems of forces** again.

Our aim was to produce one SD model and one SR model to represent each pattern. Other researchers have recognised the potential benefits of providing graphical representations of pattern solution spaces. Thomas (2001) claims that “providing people with a variety of potential representations and some process to encourage the exploration of alternatives... could probably improve performance significantly”. The use of *i** models in our pattern language enabled us to explore whether such benefits accrue empirically.

In the third phase we combined semi-structured interviews with direct SD and SR modeling to model dependencies between actors identified in the 4 modeled patterns and contributes-to soft goal links

between important soft goals in these patterns. Such modeling provided the associations between the patterns to form a first-cut pattern language.

4. The Submarine Manoeuvring Pattern Language

The resulting submarine manoeuvring pattern language was elicited from 3 BAE SYSTEMS engineers working as pairs during 5 sessions over a two-and-a-half month period. Each session lasted approximately 2 hours and took place at BAE SYSTEMS premises.

The pattern language consisted of 4 principal patterns linked using additional *i** SD and SR models. The patterns were:

1. The Manoeuvring-Noise-Accuracy (MNA) pattern, describing trade-offs between accurate and quiet steering of the submarine;
2. The Manoeuvring-Weight-Distribution (MWD) pattern, describing, describing trade-offs between accurate manoeuvring and maintaining the stability of the submarine;
3. The Manoeuvring-Console-Manning (MCM) pattern, describing trade-offs about the number of operators who control the manoeuvring of the submarine;
4. The Manoeuvring-Hydroplane-Configuration (MHC) pattern, that describes trade-offs associated with possible configurations of the hydroplanes that steer the submarine.

The full pattern language is described in Maiden & Pavan (2001). In this paper we describe one of these patterns – the Manoeuvring-Noise-Accuracy (MNA) pattern – and the models that link the individual patterns to provide the pattern language. Each is described using its important attributes.

4.1. The Manoeuvring-Noise-Accuracy (MNA) Pattern

Problem: A submarine is required to be both quiet and accurate when manoeuvring. However to be more accurate it must activate the hydroplanes, which leads to more noise.

Context: When manoeuvring, especially in advanced underwater warfare, the submarine needs to avoid detection by alien systems and to navigate with a high level of accuracy. It avoids detection by controlling and regulating the radiated noise. However, there is a trade off between the quietness (low hydroplane activity) and accuracy (high hydroplane activity). This trade off holds true for accurate navigation as high hydroplane activity implies more accurate navigation and low hydroplane activity leads to less accurate navigation as the hydroplane activity is necessary to change the submarine's direction. Similarly, high hydroplane

activity implies a high level of accuracy as well as a high level of noise.

Forces: Performance, quietness, accuracy, reliability, safety, technologies and cost.

Solution: Designers are given noise, manoeuvring and performance targets for the manoeuvring system to attain. They do this by exploring dependencies between the key agents involved in the noise-accuracy domain in order to achieve acceptable trade-offs. Historical data about the effectiveness of this design in different scenarios is available for reuse.

SD Model: We modeled 3 actors that influence manoeuvring – submarine, manoeuvring system and hydroplanes. Modeled actor dependencies were:

- The submarine depends on the manoeuvring system for the soft goal of *manoeuvre submarine quietly*;
- The submarine depends on the manoeuvring system for the soft goal of *manoeuvre submarine accurately*;
- The submarine depends on the manoeuvring system for the task of *manoeuvre submarine*;
- The submarine depends on the manoeuvring system for the goal of *navigate the submarine*;
- The manoeuvring system depends on the hydroplanes for the soft goal of *manoeuvre system is quiet*;
- The manoeuvring system depends on the hydroplanes for the soft goal of *manoeuvre system is accurate*.

The resulting SD model is shown in Figure 2.

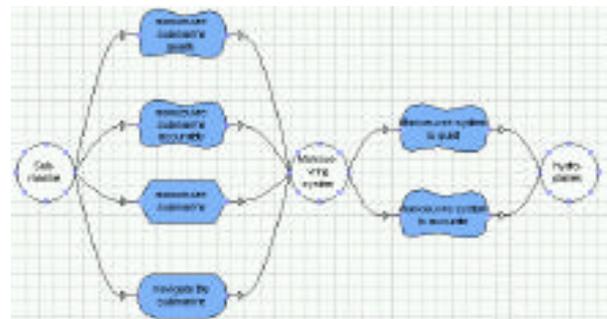


Figure 2. The SD model for the MNA pattern

SR Model: The SR model models the goal structure of each actor in the SD model. The submarine has two high-level soft goals. The first is *avoid detection by alien systems*, which is achieved by attaining the goal of *control radiated noise* and undertaking the task *regulate radiated noise*. Successfully undertaking the task of *regulate radiated noise* depends on successfully achieving the soft goal *manoeuvre system is accurate* by the manoeuvring system actor. The SR model is shown in Figure 3.

failure of one actor to achieve a goal or undertake a task can lead to serious difficulties to manoeuvre the submarine. The model calls into question the robustness of the design that this implies by this actor structure.

The 4 patterns also identified recurring structures of contributes-to soft goal links. A separate elicitation session was undertaken to determine these structures and to extract them from the specific patterns. The resulting model is shown in Figure 5.

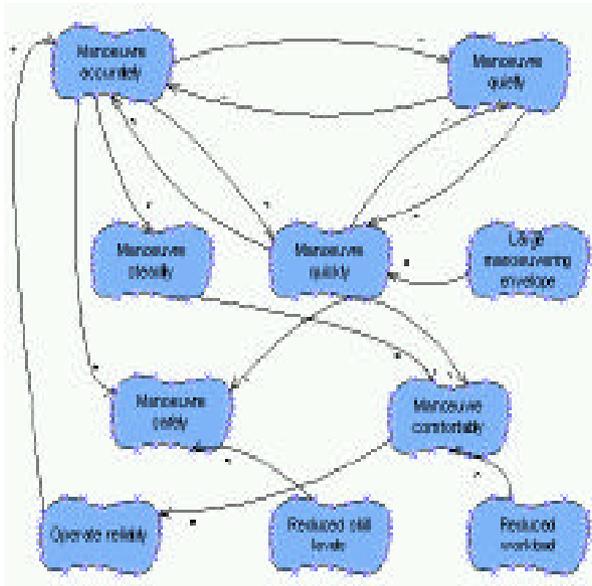


Figure 5. Recurring contributes-to soft goal links in the patterns

5. Conclusions and Future Work

This paper reports results from research to produce a pilot pattern language in systems engineering. We worked with BAE SYSTEMS engineers to elicit, model and validate a pattern language of issues to consider when designing the manoeuvring systems of naval submarines. We adopted a formal knowledge elicitation approach with the systems engineers. The resulting knowledge was modelled using the i^* formalism to show the allocation of capabilities in terms of goals, soft goals and tasks to different actors in the architecture, and the dependencies between the actors and the trade-offs to be made between the satisfaction of competing soft goals. The pattern language was accepted by the engineers as an accurate and useful representation of design alternatives for manoeuvring system. Although developed for a systems engineering problem, the definition of a pattern that we adopted for this research and the elicitation and modelling approach that was successfully applied has important consequences about

linking software system requirements and architectures.

One important finding was the usefulness of the i^* formalism for representing patterns about complex requirements and design decisions. The use of *contribute-to soft goal links* in the SR model enabled us to go beyond existing pattern-approaches in software engineering and model not just one solution to a pattern but all of the design alternatives. In our language we chose to use the i^* concept of a task to model each candidate solution. Whilst an effective representation of solutions in business and information system applications, the use of tasks (e.g. steer with low hydroplane movement) is not an ideal representation for complex system architectures, and suggests the need to extend the i^* semantics and syntax in the future. Such an extension will need to take into account the differences in discrete and continuous solution spaces. Whereas some patterns, such as the Manoeuvring-Console-Manning (MCM) pattern had discrete solutions such as *manoeuvre with 1 operator* and *manoeuvre with 2 operators*, others such the reported Manoeuvring-Noise-Accuracy (MNA) pattern has a large space of possible solutions that are modelled mathematically in BAE SYSTEMS.

During the sessions with BAE SYSTEMS engineers we found that the patterns provided the common shared point of reference that was needed to address the issues relating to requirements trade-offs. The text attributes of each pattern provided the necessary background and contextual information while the i^* models provided the engineers with powerful visual depiction of the trade-off envelope. This representation also enabled the researchers and other BAE SYSTEMS participants not as familiar with manoeuvring systems still to participate in the sessions. We believe that this was in part because the i^* semantics provided homogeneity through its powerful, but easy-to-understand processes, semantics and syntax which participants used when referring to the different attributes of the system. This went some way towards creating a more egalitarian discussion arena among the diverse participants.

Furthermore, in the latter sessions, the BAE SYSTEMS engineers became sufficiently adept with the i^* formalism that they would arrive at the elicitation or validation session with SD and SR model sketches already produced, thus saving time and improving communication. This last result was a surprise to us as we had anticipated a large learning curve with the i^* approach. This finding supports other experiences with the i^* approach that it can be learned quickly and applied successfully with stakeholders with some engineering experience. Our decision to use i^* models to express the pattern language itself, that is the associations between individual patterns, also enabled us to explore the claim that pattern languages provide a lingua franca or common language that is

accessible to all the participants in a design process (Erickson, 2000). One possible role for this and other pattern languages in BAE SYSTEMS is to aid collaboration through communication between different stakeholders in order to provide a coherent but flexible framework for problem solving, rather than forcing systems engineers to implement the specified pattern solutions for future designs.

Another interesting side effect from developing the pattern language was that it provided the systems engineers with an opportunity to reflect on their designs and design practice often denied them due to project deadline pressures. Once reflection was recognised as a characteristic of the sessions, engineers were more motivated to participate and share their knowledge with others.

The next stage of this research is to integrate this and other pattern languages within the ART-SCENE environment described at the beginning of this paper. ART-SCENE is designed to trade-off satisfaction of different requirements by different architecture designs using scenarios to link requirements and architectures, then to simulate system and agent behaviours to compute their outcomes (Zhu et al. 2003). The correctness of scenario outcomes upon which we determine an architecture's compliance with system requirements depends upon the accuracy of the model and the simulation. We seek to make ART-SCENE's simulations more dependable by diversifying the sources of information, and in particular by reusing historical data about the performance of a design in previous similar contexts based on pattern languages. We will extend the pattern language to represent architectural designs using object-oriented constructs. We will then evolve NATURE's original computational analogical reasoning mechanisms (Maiden & Sutcliffe 1996) to match candidate patterns' requirements and architecture models to the models of the system under specification to retrieve historical data about that design's performance in previous scenarios, and input this data into scenario simulations within ART-SCENE. We look forward to reporting this next challenge in the next future.

6. Acknowledgements

This research is funded by the UK EPSRC's SII 'SIMP' project. Our gratitude to BAE SYSTEMS who provides the engineering and consent for the studies reported in the paper.

7. References

Alexander C., 1979, 'The Timeless Way of Building', NY: Oxford University Press.
 Konrad S. & Cheng B., 2002, 'Requirements Patterns for Embedded Systems', Proceedings 10th International Joint

Conference on Requirements Engineering', IEEE Computer Society Press, 127-136.
 Chung L., Nixon B., Yu E. and Mylopoulos J., 2000, 'Non-Functional Requirements in Software Engineering', Kluwer Academic Publishers.
 Curran T.A. & Ladd A., 1998, "SAP R/3 Business Blueprint", Prentice-Hall.
 Franch X. & Carvallo J., 2003, 'Using Quality Models in Software Package Selection', IEEE Software Jan/Feb 2003, 34-41.
 Gross D. & Yu E., 2001, 'Evolving System Architecture to Meet Changing Business Goals: An Agent and Goal-Oriented Approach', Proceedings STRAW'2001 Workshop.
 Hall J., Jackson M., Laney R., Nuseibeh B. & Rapanotti L., 2002, 'Relating Software Requirements and Architectures using Problem Frames', Proceedings 10th International Joint Conference on Requirements Engineering', IEEE Computer Society Press, 137-144.
 Franz Huber, Sascha Molterer, Andreas Rausch, Bernhard Schatz, Marc Sihling, Oscar Slotosch, "Tool Supported Specification and Simulation of Distributed systems", Proceedings International Symposium on Software Engineering for parallel and Distributed Systems 1998, pp. 155-164.
 Jackson M., 1995, 'Software Requirements and Specifications', Addison-Wesley.
 Mijake N., 1986, 'Constructive Interaction and the Iterative Process of Understanding', Cognitive Science 10, 151-177.
 Maiden N.A.M. & Pavan P., 2001, 'Manoeuvring Pattern Language version 1.3', Technical Report, Centre HCI Design, City University London, October 2001.
 Maiden N.A.M., Jones S. & Flynn M., 2003, 'Innovative Requirements Engineering Applied to ATM', submitted to conference.
 Maiden N.A.M., Pavan P., Gizikis A., Clause O., Kim H. & Zhu X., 2002, 'Making Decisions with Requirements: Integrating i* Goal Modelling and the AHP', Proceedings REFSQ'2002 Workshop, 9-10 September 2002, Essen, Germany.
 Maiden N.A.M. & Rugg G., 1996, 'ACRE: Selecting Methods For Requirements Acquisition, Software Engineering Journal 11(3), 183-192.
 Maiden N.A.M. & Sutcliffe A.G., 1996a, 'Analogical Retrieval in Reuse-Oriented Requirements Engineering', Software Engineering Journal 11(5), 281-292.
 Maiden N.A.M. & Sutcliffe A.G., 1996b, 'Computational Mechanisms for Parallel Problem Decomposition During Requirements Engineering', Proceedings 8th International Workshop on Software Specification and Design, IEEE Computer Society Press, 159-163.
 Robertson S. & Robertson J., 1999, 'Mastering the Requirements Process', Addison-Wesley-Longman.
 Sutcliffe A.G. & Maiden N.A.M., 1998, 'The Domain Theory for Requirements Engineering, IEEE Transactions on Software Engineering, 24(3), 174-196.
 Sutcliffe A.G., Maiden N.A.M., Minocha S. & Manuel D., 1998, 'Supporting Scenario-Based Requirements Engineering', IEEE Transactions on Software Engineering, 24(12), 1072-1088.
 Zhu X., Maiden N.A.M. & Pavan P., 2003, 'Scenarios: Bringing Requirements and Architectures Together', submitted for publication

An Experience-Based Approach for Integrating Architecture and Requirements Engineering[#]

B. Paech*, A. Von Knethen*, J. Doerr*, J. Bayer*, D. Kerkow*, R. Kolb*, A. Trendowicz*, T. Punter*, A. Dutoit[†]

*Fraunhofer IESE, Sauerwiesen 6, 67661 Kaiserslautern, Germany

[†]TU München, Institut für Informatik,

{paech, vknethen, doerrj, bayer, kerkow, kolb, trend, punter}@iese.fraunhofer.de, dutoit@in.tum.de

Abstract

Deriving requirements and architecture in concert implies the joint elicitation and specification of the problem and the structure of the solution. In this paper we argue that such an integrated process should be fundamentally based on experience. We sketch an approach developed in the context of the EMPRESS project that shows how different kinds of experience-based artifacts, such as questionnaires, checklists, architectural patterns, and rationale, can beneficially be applied.

1. Introduction

The last few years have seen a growing awareness of the requirements engineering community for architectural issues and vice versa. Several authors have argued convincingly for the tight interdependencies between functional requirements (FRs), non-functional requirements (NFRs) and architectural options (AOs) that need to be made explicit early, e.g., [1], [2].

The design of an architecture aims at creating a software solution for the problem given in the requirements specification. In the requirements specification, the problem is elicited and documented using concepts from the problem domain. An architecture sketches the solution at a high level of abstraction. This means that the problem must be expressed in terms of concepts from the solution domain (i.e., the programming domain). This is a creative activity that is not well supported by current software development approaches.

In this paper, we propose an approach that supports the elicitation, specification and design activity by providing *experience* in terms of questionnaires, checklists, architectural patterns and rationale that have been collected in earlier successful projects and that are presented to developers to support them in their task.

The approach uses a refinement graph, checklists and questionnaires to capture important NFRs more precisely. In addition, it uses architectural patterns for reusing AOs and for evaluating them against a specific set of requirements. Furthermore, it uses traceability and rationale management to make explicit the decision making involved in a joint specification and design of FRs, NFRs and AOs.

The paper is structured as follows: First, we sketch the fundamental issues to be solved in integrating RE and architecture development, and how these are covered by related work. Second, we discuss the foundation of our approach in terms of a metamodel that describes the basic concepts we are dealing with, such as quality attributes, metrics, and NFRs. Third, the integrated process with input and output documents is described. We conclude with a discussion of how well our approach deals with the fundamental issues identified.

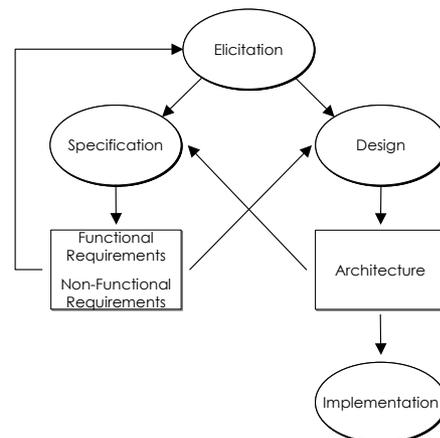


Figure 1: General process of integrating architecture and requirements

2. Fundamental Issues

Figure 1 shows the general process of integrating the architectural decision process into the requirements engineering process.

It comprises the relevant activities of the software engineering process, namely an iteration of requirements elicitation, specification, and design that produces FRs, NFRs and AOs. These are subsequently implemented.

[#]The research for this paper has been partly funded by the EUREKA-ITEA projects “EMPRESS” (ITEA 01003) and “CAFÉ” (ITEA 00004)

As exemplified by the approaches presented at STRAW 2001, there are many different ways to support these activities. They mainly solve the following fundamental issues:

- **Issue 1 – Views of different stakeholders in the elicitation of NFRs, FRs, and AOs:** How to identify the essential NFRs, FRs, and AOs and different views of different stakeholders? How to negotiate conflicts? What is a sufficient level of abstraction for these discussions? One possible support for negotiation is given by the WinWin approach [2][3][4].
- **Issue 2 – Identification of dependencies among FRs, NFRs, and AOs:** How to describe NFRs, FRs, and AOs such that dependencies can easily be identified? In several approaches, goal graphs are used for specifying NFRs and FRs and their dependencies. There is much less agreement on describing AOs, e.g., Use Case Maps [5], agent-oriented goal graphs [6], the CBSP approach [4], or social organizations [7].
- **Issue 3 – Assessment of how well different AOs address a specific set of FRs and NFRs:** How to capture and support the decision making involved in specifying FRs, NFRs and AOs? Typically, concepts from *rationale management* [8] are used to make explicit questions to be solved, options for their solutions, criteria to evaluate the options and assessments of the options against these criteria. For example, goal graphs are used to capture criteria (business goals) and issues (NFRs and FRs), AOs and their assessments [5]. Another example is the Concordance Matrix to capture assessments of the architectural relevance of FRs and NFRs [4]. Also, SEIs Architecture Tradeoff Analysis Method (ATAM) captures criteria (quality attributes, business goals), issues (risks), options (architectural views), and assessments (utility tree). The Cost Benefit Analysis Method (CBAM) is used to refine the ATAM results with cost, benefit (criteria, options) [3].

As argued in the introduction, however, the design of an architecture is a creative task. It involves much judgment and heuristics on the importance of NFRs and FRs and different AOs. Thus, it is error-prone (e.g., guesses about how well an architecture meets a set of NFRs can be wrong) or expensive (e.g., when using a prototype realization of the architecture to experimentally assess the suitability of the architecture). Moreover, it can only be learned through experience and apprenticeship. Hence, leveraging off past experience can help these challenges to be addressed. This raises another issue:

- **Issue 4 – Representation of past experience to facilitate issues 1-3:** How can one capture and use

experience on FR, NFRs, AOs, their dependencies and their assessments? Such representations must not only include the AOs under consideration, but also sufficient knowledge for their selection and application. This includes the context in which they can be used and the trade-offs they entail. Architectural styles, for example, are used to capture typical AOs, a correlation catalogue to capture typical assessments [7].

The last issue is rather implicitly treated in many approaches. In contrast, we have put most emphasis on identifying how experience can support the integrated process.

3. Our Approach

In the following, we present our approach for capturing experience to support the integrated elicitation, specification of NFRs and FRs and design of architecture. First, we explain the fundamental concepts in terms of a metamodel. Second, we sketch the process and the products with a case study dealing with a mobile, interactive application to allow users monitor production activities, manage physical resources and access information. This case study is based on a real system and was provided by Siemens in the context of the Empress project.

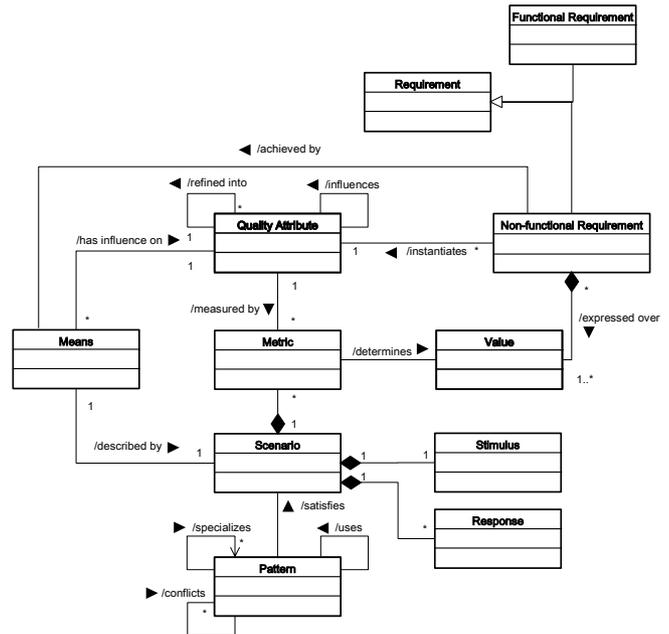


Figure 2: The metamodel

3.1 Foundation

Our integrating approach is based on a metamodel that describes the main concepts we are dealing with (see Figure 2).

- *quality attribute (QA)* is a non-functional characteristic of a software product or process. We distinguish between high-level QAs (i.e., efficiency, maintainability, reliability, usability, and portability) and refining QAs of these attributes. The high-level QA “efficiency” can, for example, be refined into “time behavior” and “resource utilization”, “time behavior” can be refined into “workload” and “response time“. In addition, QAs can have positive or negative *influences* on each other, e.g., if the “workload” is higher, the “response time” will increase (negative influence).
- To make explicit the distinction between knowledge about QAs gained in experience and the quality to be achieved in a specific project, we use the term *NFR* to describe the latter. A NFR is an *instantiation* of a QA that is created by determining a *value (range)* for a *metric* associated with the QA. For example, the NFR “The database of our new system shall handle 1000 queries per second.” instantiates the QA “workload of database”. The value is determined based on an associated metric “Number of jobs per time unit”.

The distinctive feature of this metamodel is that we distinguish problem-oriented refinement from solution-oriented refinement of QAs. The latter is made explicit in terms of means which mediate between QAs and patterns.

- *Means* are principles, techniques, or mechanisms that facilitate the achievement of certain qualities in a software architecture. They are abstract patterns that capture a way to achieve a certain quality requirement, but are not concrete enough to be used directly (i.e., they have to be instantiated as patterns). Means are described by *scenarios*, which consist of stimulus and response, and a metric. For example, a scenario for the NFR mentioned above is “object creation throughput must be fast”, where the stimulus is “object creation”, the response is “throughput” and the metric is “number of objects created per second”.
- A *pattern* is used to document AOs. Pattern help designers in creating architectures by providing solutions for recurring problems in the design of software architectures. The pre-defined solutions have proven to be beneficial in certain situations. As they have been applied repeatedly, their impact on a software architecture is known. Patterns are chosen to satisfy the scenarios. They can be refined through *specializations*. For example, the pattern

“layered architecture” can be specialized into “strictly layered architecture” and “loosely layered architecture”. Furthermore, if a pattern *uses* another pattern, the used pattern is applied to create the using pattern. With this mechanism, collaborating patterns can be used to form higher-level patterns. Two patterns can also be in *conflict*, e.g., the “client server” and “layered architecture” patterns cannot be applied at the same time.

The following sections describe how these concepts are used within our approach.

3.2 Experience-Based Process

Figure 3 gives an overview on our experience-based process of integrating architectural decision making into the requirements engineering process.

In the following, the different activities of our process are listed. The overall process is iterative, that means within each activity and between the activities iterations are probable and necessary. Products consumed and produced by the activities of the process are explained in more detail and illustrated with examples in the following sections.

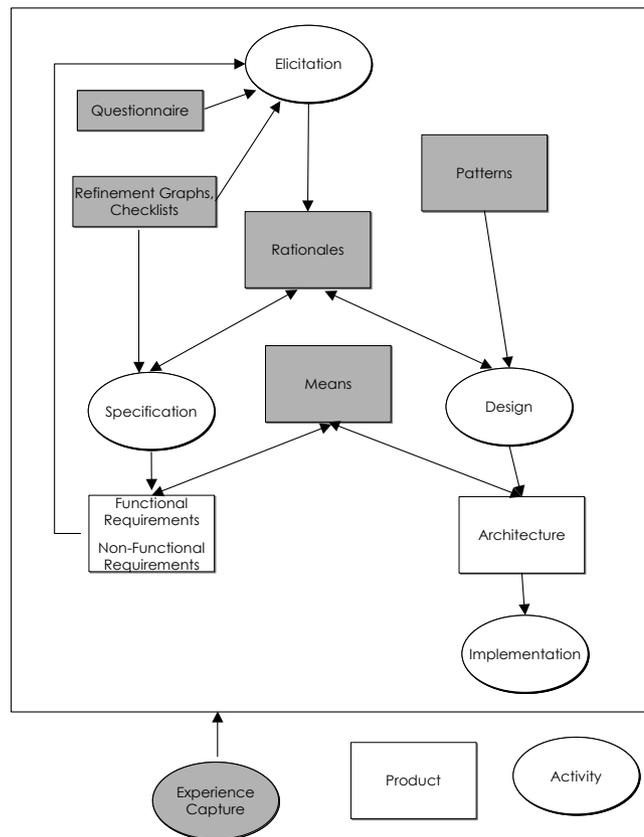


Figure 3: The experience-based process

- **Elicitation:** During the elicitation, the customer has to prioritize the QAs at the highest level of abstraction for the system to be developed. A questionnaire is used for this purpose. Then, QAs with the highest priorities are refined with the help of checklists. Refinement graphs for the high level QAs are the foundation of all checklists. We distinguish different types of checklists. Each checklist focuses on a certain refinement aspect (e.g., problem-refinement, solution-refinement, dependencies between QAs). The rationale for specific estimates for the NFR (e.g. maximal load) is captured.
- **Specification:** During the specification, measurable NFRs will be documented in a requirements document. Checklists guide this activity. We use a requirements template that allows different NFRs to be described at different places in the document. NFRs, for example, that are expressed over FRs are explicitly stated together with the FR. We use Use Cases and Use Case descriptions to describe FRs (our approach for describing FRs for embedded systems has been developed in the QUASAR project [9]). NFRs (e.g., response time requirements) are explicitly stated in the Use Case descriptions. Furthermore, concrete means to achieve the NFRs are identified by using the assessments of their suitability documented in a refinement graph. The rationale for a chosen means is captured.
- **Design:** During the design, requirements that have an effect on the architecture are selected. In addition, the principal structure of the system is refined based on the requirements and the means and pattern catalogue. In the following, the existing architecture is iteratively refined based on requirements and the catalogue. After each refinement step, the architecture is assessed concerning their non-functional properties. The rationale for chosen means and patterns is captured.
- **Experience Capture:** During the performance of a project, experiences are collected and consolidated to improve the questionnaire, refinement graphs and checklists and the patterns and means catalogue.

3.3 Questionnaire for Prioritization

For the prioritization of QAs at the highest level of abstraction, a standardized questionnaire is used. The questionnaire elicits wishes and facts concerning the development context of the customer and relates them to a selection of the QAs defined by ISO9126 [10]: we selected maintainability, efficiency, usability, and reliability in our case study.

In the following, we describe at first how the questionnaire was developed and then how it can be applied.

To develop the scales of the questionnaire, in a first step, potential scale items were generated. For this purpose, we phrased a set of 120 statements containing wishes and facts, which a person involved in a system development project would express. The statements covered the complete set of second level QAs (ISO 9126) of the high level QAs mentioned above.

Once the statements were generated, they were presented to eight software quality experts. These experts judged, whether a customer that needs a certain QA would agree to each statement. A 1-5 rating scale was used for the judgment. The experts were – as usual in scale development [11] – asked not to rate their own project context, but rather to judge based on their personal experience, how favorable each item is with respect to the QA of interest.

In a next step, items with the highest mean and lowest variance (high interrater reliability) were selected and assembled to a 30 items questionnaire. As response scale, a 1-5 Likert scale was chosen, of which 17 statements covered facts of the current project (strongly agree – strongly disagree), and 13 statements covered wishes for future conditions (very important – very unimportant).

The determination of the mean value of the statements affecting one QA enables to build a rank order of these. The one with the highest ranking is the most important attribute for the current system development project and should receive the greatest deal of attention. The prioritization is of special interest in case of limited requirements engineering resources and allows focusing the requirements engineers' energies on the most important high-level QAs. This prioritization questionnaire was applied in the case study. It did not confirm the prior expressed expectations of the customers. A closer analysis showed, that the customers tended to rate such quality aspects as most important, that were difficult for them to handle, namely "efficiency", instead of naming the most important aspects for the success of the project in scope. The results of the prioritization questionnaire ranked "maintainability" as most important. The customer confirmed the correctness of this result.

3.4 Refinement Graph

A refinement graph (also called quality model) instantiates parts of our metamodel. It describes typical refinements of high-level QAs into more detailed QAs, metrics, and means. In addition, it describes relationships between different QAs. Therefore, it captures experience of previous projects. Our refinement graph is similar to the goal graphs of e.g. [6], but emphasizes dependencies. Figure 4 gives an example for such a refinement graph for the QA "efficiency". White rectangles represent QAs at

different levels of detail. Ovals represent metrics that measure certain QAs. Grey rectangles represent means to achieve certain QAs.

Four types of relationships can be found in such a refinement graph. The metamodel in Figure 2 describes the general types of relationships.

- A QA, such as “efficiency” is *refined into* more detailed QAs, such as “time behaviour” and “resource utilization”.
- A means *has influence on* a QA, i.e., it is used to achieve the QA, e.g., “load balancing” is used to achieve “workload distribution”.
- A QA is *measured by* a metric. For example the “workload” can be measured by the metric “number of jobs per time unit”.
- A QA can be positively or negatively *influenced by* another QA. If the “workload”, for example, is higher, the “response time” will increase (negative influence).

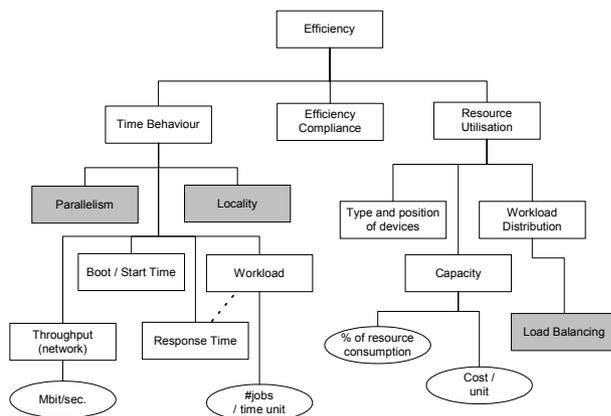


Figure 4: Refinement graph for efficiency

Our approach provides a default refinement graph that can be used without adaptations by a company. Reasons for this can be a lack of time or money. We recommend tailoring the refinement graph to the context of each company and project. Alternatively, a company might have an own refinement graph that shall be used. In this case, it is very important to agree on the meaning of the different QAs in this graph. Our recommendation is to build a refinement graph together with the company in a workshop. By doing so, the refinement graph benefits from the already integrated experience of our default refinement graph and it is tailored to the project and company. So far, we defined default refinement graphs for the QAs “efficiency”, “reliability”, and “maintainability”. NFRs are elicited for each QA and relationships between NFRs and FRs are established via the checklists.

A mechanism to capture the experience of multiple projects and store the various refinement graphs is also

developed as part of the ITEA EMPRESS project. This so-called *Prometheus* approach (Probabilistic Method for early evaluation of NFRs) is described in [12].

3.5 Checklists

Based on the information included in the refinement graph, we developed checklists that focus on different aspects of a high-level QA. We distinguish for each high-level QA between: (1) initialization checklists, (2) refinement checklists, and (3) dependency checklists. All checklists are described in more detail in the following. Again, in the other approaches for integrating RE and architecture we have not found something similar to checklists. They help to make the experience captured in the refinement graph directly applicable in workshops.

Initialization checklists are defined that capture everything that has to be decided before NFRs are refined. There are two types of initialization checklists that are used in our process: a general initialization checklist and specific high-level QA checklists.

The general initialization checklist includes aspects of the following categories:

- Organizational aspects (e.g., domain knowledge required)
- Technical issues (e.g., notations required)

Figure 5 depicts an extract of such a general initialization checklist.

1. **Supplier** (*project issues->process stakeholders*)
 - Do you expect a certain kind of organization? (e.g., number of employees, size of company, lifetime of company)
 - Do you expect a certain domain experience? If yes, specify the level of experience! (e.g., 2 years experience / number of developed systems in the domain)
2. **Process and documents**
 - Do you have certain constraints towards the system development process?
 - a. Conformance to standards (e.g., ISO 9001, IEEE ...) (*project issues->process requirements || documentation requirements*)
 - b. Special activities (e.g., audits/reviews) (*project issues->process requirements*)
 - c. Documentation (e.g., specification documents) (*project issues->documentation requirements*)
 - d. Notation (e.g., statecharts) (*project issues->documentation requirements*)

Figure 5: Extract of general initialization checklist

Initialization checklists include a set of questions. To support answering the questions, examples are given in brackets. Italic formatted comments describe at which place in the requirements document, the information should be stated. Examples for NFRs concerning organizational experience are:

- “At least 3 years of experience in maintenance is required (the longer the better).”

- “Project experience with wireless networks is required.”

An excerpt of a specific initialization checklist for the high level QA “efficiency” is given in Figure 6. This structure of the checklist corresponds to the structure of the other initialization checklist. In our case study, there were no specific NFRs concerning the organizational experience regarding efficiency.

1. **Supplier** (*project issues->process stakeholders*)
 - Do you expect a certain experience in building time critical systems? If yes, specify the level of experience! (e.g., had a similar project before, has special qualification (training, certification)).
 - Do you expect a certain experience in building systems with resource limitations? If yes, specify the level of experience! (e.g., had a similar project before, has special qualification (training, certification))
2. **Process and documents** (*project issues->process requirements || documentation requirements*)
 - Are there any laws or standards regarding efficiency your system will have to adhere to? If yes, specify the requirement!

Figure 6: Excerpt of efficiency initialization checklist

NFRs, refinement checklists are used to elicit specific measurable NFRs. Refinement checklists are specific for high-level QAs (e.g., efficiency). In case of efficiency and reliability requirements, we recommend creating Use Cases to identify concrete NFRs. An excerpt for the refinement checklist for throughput NFRs is given in Figure 7. Again, text in italics indicates the place to document the NFR in a given document structure.

6. **Elicitation of Throughput NFRs** (*FRs-> UC description -> NFRs -> throughput || NFRs->efficiency ->throughput*)
For each network element in the system architecture:
 - Go through each Use Case: Identify the UC-steps and exceptions involving data transportation on this component. Think of an **average** scenario of the Use Case
 - How much data has to be transported by this component?
 Specify the throughput NFRs on the network element.
 - Go through each Use Case: Identify the UC-steps and exceptions involving data transportation on this component. Think of a **maximum** usage of the Use Case
 - How much data has to be transported by this component?
 Specify the throughput NFRs on the network element.

Figure 7: Excerpt of refinement checklist for throughput

Measurable efficiency NFRs that were elicited by using the refinement checklists in the case study are for example:

- In a maximum usage, 8 people must be able to download a document (about 1 MB) within 10 sec. via the WLAN (6.4 Mbit/s).
- The PDA must be able to handle 60 alarms (coming from machines) at the same time.
- The memory of the database server must at least have a capacity of 512 MB.

While eliciting the NFRs, dependencies to other NFRs and architectural decisions are checked by using a dependency checklist. Figure 8 depicts an excerpt of the efficiency dependency checklist.

After applying these checklists, conflicts between NFRs and solution alternatives are documented. If concrete solutions were specified, also the rationale for the decision is documented. In our case study, a conflict appeared between the following two NFRs:

- “In a maximum usage, 8 people must be able to download a document (about 1 MB) within 10 sec. via the WLAN (6.4 Mbit/s).”
- “The WLAN supports 10 Mbit/sec.”

In this case, the net throughput of the WLAN might be not sufficient for the first requirement. This conflict was documented.

The NFRs have been expressed separately. Now check for the NFRs affecting the same architecture component and instantiating dependent quality attributes:

- Consider first dependencies between refinements of efficiency: Is it possible to fulfill all NFR at the same time (e.g. response time and workload)? If not, high-light the conflict.
- Consider then all dependencies: Is it possible to fulfill all NFR at the same time (e.g. response time and maintainability)? If not, high-light the conflict.

Figure 8: Excerpt from efficiency dependency checklist

3.6 Means and Architectural Patterns

The general dependencies between means and patterns are captured in a separate catalogue. This catalogue is used as follows. A designer working on a certain component or (sub-) system chooses the architectural relevant FRs, as well as the NFRs. The NFRs are then used to select appropriate means. This is done by comparing the scenarios associated with the means with the requirements. Once the means are selected, the patterns that specialize the respective means are selected from the catalogue. This is again is done by comparing the scenarios related to the patterns with the requirements. The selected patterns are instantiated to support the design.

3.7 Rationale

The refinement graph and the catalogue capture general relationships between QAs, means and patterns. The choice of a specific pattern requires detailed evaluation of the means and the patterns against the relevant requirements. We capture this evaluation in terms of rationale that can then be used to refine the refinement graphs and the catalogue.

The designer documents the selection of means with an assessment matrix for each subsystem under consideration (see Table 1). The rows of the matrix

represent the selected means. The columns of the matrix represent the requirements that are relevant to the subsystem under consideration. Each cell denotes whether a specific means makes it easier or more difficult to realize the corresponding requirement with the symbols “+” and “-” and a reference to the scenario that was used to generate the value. If the means has no impact on the requirement, the cell is left empty. Once the matrix has been filled out, the designer identifies potential conflicts between selected means. While the designer can select alternate means in order to reduce the number of conflicts, in general, however, the potential conflicts cannot be completely eliminated. The remaining conflicts are documented by annotating the cells (i.e., means x requirement x scenario) that are involved in the conflict for further consideration during the next step.

	FR1	FRn	Efficiency	Maintainability
Locality			-	+
Load balancing			+	-
Caching			+	-
Concurrency			+	-
Sharing			+	-

Table 1. High-level assessment matrix for detecting conflicts among means

The patterns are selected by comparing the scenarios related to the patterns with the requirements. For each means, the designer builds a new assessment matrix. The rows represent the candidate patterns selected with the scenarios. The columns include the requirements addressed by the means. When the means under consideration is involved in a conflict, the columns in the higher-level matrix that are negatively affected by the means are reported into the lower-level matrices. The designer uses the scenarios that result in negative assessments in the higher-level matrix to select a set of architectural patterns, hence addressing the relevant requirements and resolving the potential conflict.

This two-level approach for documenting trade-offs between options is similar to the rationale capture of designing services from user tasks described in [13]. The use of an assessment matrix enables the designer to summarize the rationale behind the selection of means and patterns and their evaluation with scenarios. Using a two level selection process reduces the size of the matrices that the designer has to work with and the total number of cells that need to be considered. By identifying conflicts in the higher-level matrix and reporting conflicting columns in the lower-level matrices, the designers focuses only on the relevant interactions between means and attempts to address those during the pattern selection and instantiation. Thus, the distinctive

feature of our rationale capture is the detailed guidance we give for decision making.

4. Conclusion

We have presented a comprehensive approach covering the issues identified in section 2.

- Issue 1: The different views of the stakeholders are elicited and negotiated through the prioritization questionnaire, different view-oriented checklists and the rationale-based discussion. The distinction between QAs and means helps to keep the discussion on an adequate level of abstraction. This is achieved by separating problem refinement from solution refinement.
- Issue 2: Typical dependencies between QAs are captured in the refinement graphs. Concrete dependencies are elicited with the help of checklists and are captured in the rationale matrices. We use patterns to document AOs and Use Cases to document FRs. We use a requirements template that allows different NFRs to be described at different places in the document. NFRs, for example, that are expressed over FRs are explicitly stated together with the FRs. However, we have not yet worked on an intuitive representation of the dependencies between patterns and Use Cases.
- Issue 3: The relationships between AOs, FRs and NFRs are covered by our rationale matrices.
- Issue 4: As described in detail, we capture and use experience in terms of the questionnaire, the refinement graphs, the checklists, the patterns, and the rationale.

Of course, there are still many issues to be solved, in particular a full-scale case study. So far, we have used this approach together with our cooperation partners from Siemens to elicit and specify the FRs, NFRs, means and metrics. In a 2 day workshop its was possible to define a measurable and a more complete set of NFRs in comparison to ad-hoc approaches. In addition, the relationships between FRs and NFRs were clear. The choice of the patterns will be performed in the near future.

Till the end of the year, we plan to address the following questions:

- Package experience for different QAs from literature, in particular the catalogues for means and patterns.
- Find suitable architecture descriptions that facilitate the assessment of the dependencies between requirements and AOs.

So far, we have not investigated the utilization of problem frames (as a further instance of documented experiences). That would correspond to capturing typical FRs in the refinement graph. This would generalize our work from the domain of embedded systems – which is the focus of EMPRESS – to other domains.

Acknowledgements

We acknowledge the ITEA project EMPRESS for partly funding our research. Furthermore, we want to thank all partners in the ITEA project EMPRESS that contributed to our research. In particular, we want to thank Ricardo Jimenez Serrano (Siemens) for providing a case study to validate our approach.

References

- [1] B. Paech, A. Dutoit, D. Kerkow, A. von Knethen: „Functional requirements, non-functional requirements and architecture specification cannot be separated – A position paper”, REFSQ 2002
- [2] In, H., Boehm, B.W., Rodgers, T., Deutsch, W., "Applying WinWin to Quality Requirements: A Case Study", ICSE 2001, pp. 555-564, 2001
- [3] In, H., Kazman, R., Olson, D., “From requirements negotiation to software architectural decisions”, STRAW 2001
- [4] Egyed, A., Grünbacher, P., Medvidovic, N., “Refinement and evolution issues in bridging requirements and architecture – the CBSP approach”, STRAW 2001
- [5] Liu, L., Yu, E., “From requirements to architectural design – using goals and scenarios”, STRAW 2001
- [6] Gross, F., Yu, E., “Evolving system architecture to meet changing business goals: an agent and goal-oriented approach”, STRAW 2001
- [7] Kolp, M., Castro, J., Mylopoulos, J., “A social organization perspective to software architectures”, STRAW 2001
- [8] Dutoit, A., Paech, B., “Rationale management”, in Handbook of Software and Knowledge Engineering, World Scientific Publishing, 2002
- [9] von Knethen, A., Paech, B., Houdek, F., Kiediasch, F., “Systematic Requirements Recycling through Abstraction and Traceability”, Int. Conf. RE 2002
- [10] ISO/IEC 9126-1:2001(E), “Software Engineering - Product Quality - Part 1: Quality Model”, 2001
- [11] Trochim, W. M. K., “The Research Methods Knowledge Base”, Atomic Dog Pub Inc., Cincinnati, 2001
- [12] Punter, T., Trendowicz, A., Kaiser, P., “Evaluating evolutionary software systems”, PROFES 2002
- [13] Dutoit, A., Paech, B., “Rationale-based Use Case specification”, Requirements Engineering Journal, special issue REFSQ 2001, 2002
- [14] E. Dincel, N. Medvidovic, A. van der Hoek, “Measuring Product Line Architectures”, in: Int. Workshop on Product Family Engineering, Bilbao (S), 2001

A Framework for Managing Traceability Relationships between Requirements and Architectures

Susanne A. Sherba and Kenneth M. Anderson
University of Colorado
Department of Computer Science
Boulder, CO 80309-0430 USA
{sherba, kena}@cs.colorado.edu

Abstract

Traceability helps stakeholders to understand the relationships that exist between software artifacts created during a software development project. For example, the evolution of the relationships between requirements and the components to which they are allocated can provide insight into the maintainability of a system. Unfortunately, due to the heterogeneous nature of these artifacts, creating, maintaining, and viewing these relationships is extremely difficult.

We propose a new approach to traceability based on techniques from open hypermedia and information integration. Open hypermedia and information integration provide generic techniques for establishing, maintaining, and viewing relationships between software artifacts. Our approach allows the automated creation, maintenance, and viewing of traceability relationships in tools that software professionals are accustomed to using on a daily basis.

1. Introduction

Traceability can provide important insight into system development and evolution. Antoniol *et al.* maintain that traceability assists in both top-down and bottom-up program comprehension [4]. According to Jacobson, Booch, and Rumbaugh, “[t]raceability facilitates understanding and change [9, page 10].” Palmer asserts that “[t]raceability gives essential assistance in understanding the relationships that exist within and across software requirements, design, and implementation [13, page 412].”

Even if we limit our discussion to relationships between requirements and architectural artifacts, we are confronted by a large number of potentially useful relationships. The models of Ramesh and Jarke [16] and Pohl [14, 15] suggest possible relationship types between various elements and artifacts. Han [7] lists three categories of structural

relationships: coarse-grained inter-document, fine-grained inter-document, and fine-grained intra-document. We divide inter-document relationships into two subcategories: relationships between different versions of the same artifact and relationships between different artifacts. Furthermore, we consider relationships between both consecutive and non-consecutive versions of the same artifact as well as relationships between relationships. Figure 1 depicts these five relationship types.

Relationships can exist between elements of a single artifact (relationship type 1 in Figure 1). For example, in a requirements specification, one requirement might *elaborate* or *depend_on* another. In an architectural diagram, a component might be *part_of* another component or *depend_on* another component. Across versions of the same artifact (relationship types 2 and 3) we may observe relationships such as *refines*, *replaces*, *based_on*, and *formalizes*. Between requirements and architectural components, relationships such as *satisfies* and *allocated_to* might be useful (relationship type 4). Furthermore, software engineers may be interested in how a particular type of relationship between artifacts evolves over time (relationship type 5). For example, a *number_of_instances* relationship might provide insight into component cohesion. If the number of *allocated_to* relationships between a requirements document and a component diagram explodes after an iteration in the design phase, this may indicate that one or more components have lost cohesion.

All of these relationship types might be useful to one or more stakeholders at some point in the software development project; however, the vast number of possible relationships makes the task of manually creating and maintaining these relationships daunting. Furthermore, different stakeholders have diverse information needs. Not all relationships will be of interest to all stakeholders. For example, a customer might only be interested in knowing that all requirements have been allocated to components whereas a

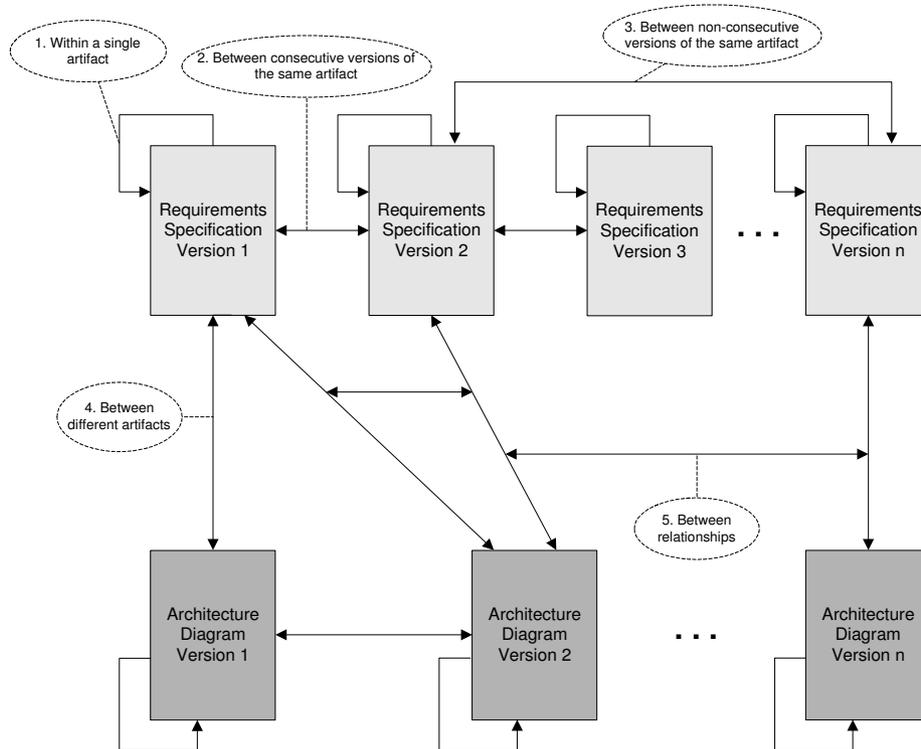


Figure 1. Implicit and Explicit Relationships in Software Artifacts.

software developer might need to understand the requirements in light of their allocated components and sub-components as well as the dependencies between these components. Stakeholders need to be able to filter relationships to provide a view of the information space that conforms to their information needs.

Finally, users should not be required to use a specialized tool to view traceability information. Different stakeholders use different tools to produce requirements and architectural artifacts. It is more likely that stakeholders will make use of these relationships if they are able to view the relationships in the tools that originally created the artifacts. However, these tools are often not designed to interact. Thus, we must also overcome the problem of heterogeneous artifacts produced by different tools [3].

These problems lead to three requirements for creating, maintaining, and viewing traceability relationships:

- The creation and maintenance of traceability relationships must be automated.
- Stakeholders must be able to create a view of traceability relationships based on their information needs.
- Users should be able to create and view traceability relationships within common, familiar software tools.

In addition, we suggest that a traceability tool should provide support for evaluating the evolution of relationships between artifacts. This analysis can provide insight into the entire project.

2. Approach

We hypothesize that open hypermedia [12] and information integration [2] enable an approach to traceability that allows:

1. automation of the discovery, creation, and maintenance of traceability relationships.
2. customized views of these relationships based on the information needs of the stakeholders.
3. creation and viewing of these relationships in the tools that originally created the artifacts.

In this section, we provide a brief overview of open hypermedia and information integration and then present our conceptual framework. Next, we offer a motivating scenario and then describe our proposed prototype.

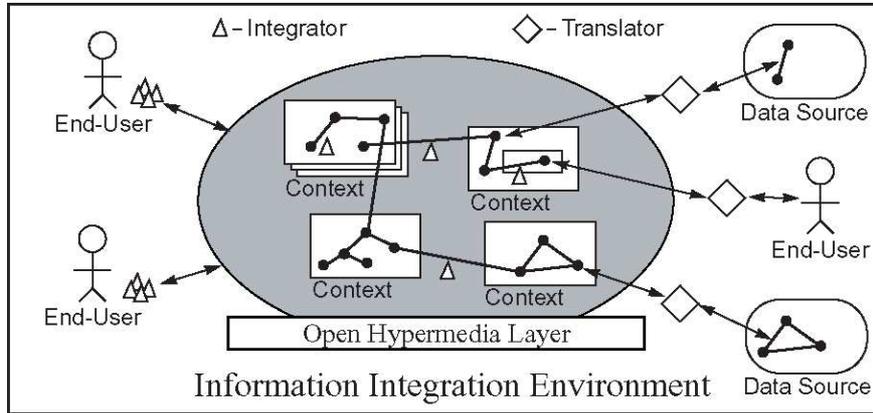


Figure 2. The InfiniTe Information Integration Environment [2].

2.1. Open Hypermedia

Open hypermedia systems [12] enable the creation and viewing of relationships in heterogeneous applications as well as the traversal of those relationships within and between applications. Open hypermedia services allow links (relationships) to be stored separately from an artifact. The open hypermedia data model supports complex relationships such as bi-directional relationships, relationships with multiple anchors (n -ary relationships), and relationships between relationships [3]. Open hypermedia systems also provide services to filter relationships based on type and to create collections of relationships (hyperwebs or, more commonly, linkbases).

2.2. Information Integration

Information integration [2] provides services to automate the discovery, creation, maintenance, and evolution of relationships between heterogeneous artifacts. Information integration uses the concept of a *data source* to model information outside the information integration environment. *Translators* are responsible for importing information from data sources into the information integration environment as well as exporting information from the information integration environment to data sources. *Integrators* work within the environment to automate the discovery and creation of relationships. Specific integrators can be developed to find different relationships within the environment. These relationships can be between artifacts, other relationships, or collections of artifacts and relationships. In addition, information integration uses *contexts* to model different views of the information space. A conceptual model of the InfiniTe information integration environment [2] is shown in Figure 2.

2.3. Conceptual Framework

Our conceptual framework builds on techniques from open hypermedia and information integration. The main elements in our framework are *tool*, *artifact*, *relationship*, and *metadata*. These concepts are illustrated in Figure 3. A *tool* is something that a stakeholder uses to perform a task. Examples of tools include word processors, UML diagramming tools, integrated development environments (IDEs), mail programs, version control systems, and issue tracking systems. An *artifact* is produced by a tool. A *relationship* is a semantic association between artifacts, portions of artifacts, or relationships. *Metadata* allows a method engineer (or someone familiar with the software project) to describe the artifacts and relationships that are created and used during the project.

As can be seen in Figure 3, stakeholders are able to use their original tools. These tools produce heterogeneous artifacts. Artifact and relationship metadata provides information about artifact and relationship types. The metadata includes information such as which translators can be applied to specific artifact types or which integrators can be used to find particular relationship types. The artifacts are translated into the information integration environment where traceability relationships can be automatically discovered and created by appropriate integrators as determined from the metadata. These relationships are forwarded to the open hypermedia system, which provides services to display the relationships in the tools that originally created the artifacts. The traceability system provides services to schedule integrators and translators, to “chain” relationships together to form new relationships, to register new artifact and relationship types, to create customized views by filtering both artifacts and relationships, and to provide insight into system evolution based on the traceability relationships that the system has discovered.

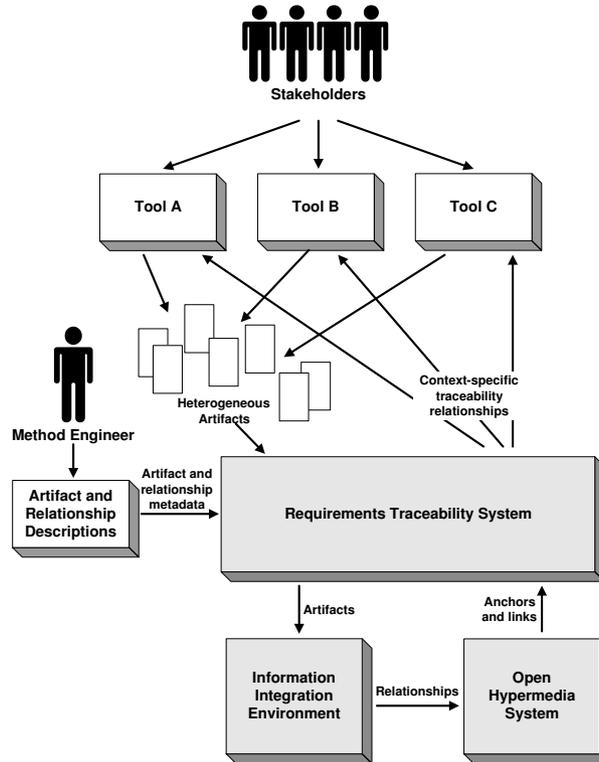


Figure 3. Conceptual Traceability Framework.

2.4. Motivating Scenario

This section provides an example of the use of relationships to provide insight into the evolution of requirements and a system architecture. The hypothetical product is an income tax program. An initial analysis determined that much of the functionality of the system would remain stable (e.g. user interface, data import and export). However, the tax calculation component of the software would need to be frequently updated to reflect current state and federal tax laws. Thus, maintainability has been identified as one of the important quality requirements of the software architecture.

In the initial architecture, the system architect chose to allocate all requirements related to the calculation of taxes to one component. The product is now in its fifth release and the project manager has observed that it is taking software developers more time to update the software to reflect the current tax laws. In addition, these changes are adversely affecting other functionality in the system.

The system architect is called in to review the evolution of the architecture. To analyze the problem, he requests that he be given a summary of all *allocated_to* relationships between tax computation requirements and the components that satisfy those requirements. From this information, he is able to discern that in the first three product releases, all tax

computation requirements were indeed satisfied by the tax component. However, in the fourth and fifth releases, the relationships show that some of the tax requirements are being satisfied by a component that also satisfies data import requirements. For more details, the architect opens the requirements specification associated with the fourth product release. He requests that the system display all *allocated_to* relationships. By traversing these relationships, he is able to view the component that currently satisfies the requirement. Thus, the system architect is able to identify the point at which the architecture began to lose its conceptual integrity [5] and to suggest changes to restore the cohesion of the tax component.

This scenario shows the need to be able to create and view relationships between different versions of different artifacts in a customized context. In addition, it suggests that the evolution of relationships over various versions and product releases can provide valuable insight into system evolution.

2.5. Prototype

To evaluate the feasibility of our approach, we plan to build a traceability system that implements the conceptual framework described in Section 2.3. The traceability system will provide an infrastructure for the automated

discovery, creation, and maintenance of traceability relationships between heterogeneous artifacts. The system will be built using services from both open hypermedia and information integration as well as our own traceability-specific services. To provide open hypermedia services, we have chosen the Chimera open hypermedia system [3]. We will use the InfiniTe information integration environment [2] for information integration services.

InfiniTe will provide services for the automated creation of typed traceability relationships. As described in Section 2.2, translators will be used to import information to and export information from InfiniTe; integrators will be used to discover and create relationships. Chimera will provide services for viewing and traversing relationships created by InfiniTe in the tools that originally created the artifacts. To utilize all open hypermedia services, a tool will need to be integrated with Chimera (a Chimera “client”).¹ If a tool is not integrated with Chimera, the user will be able to request an HTML summary of the artifact’s relationships. The relationships created by InfiniTe will be typed to facilitate filtering these relationships to create customized contexts. Chimera allows the selective viewing of relationships based on type.

Our customized traceability services will be based on information provided in the artifact and relationship metadata. A metadata definition tool will allow a method engineer to define the artifacts and relationships that are required for an organization’s software development process. Medvidovic *et al.* [10] describe strategies for modeling architectures in UML. Assuming that a project creates its requirements specification in Microsoft Word [11] and creates architecture diagrams in a UML diagramming tool, we can create specific translators to translate these artifacts into InfiniTe. We can then write one or more integrators to find relationships of interest, for example the *allocated_to* relationship between requirements in the requirements specification and components in the architecture diagram. In addition, we can create one or more integrators to analyze the evolution of the relationships between these artifacts. These translators and integrators can be invoked automatically or as needed, depending on the definitions in the metadata.

Since it is impossible to anticipate the needs of every software development team, we do not propose to build a comprehensive set of translators and integrators for every artifact and relationship type. We do, however, propose to implement a system that will manage artifact and relationship metadata, inform users of available options for integrators and translators, and invoke translators and integrators when appropriate. The system will also provide information about explicit and implicit relationships in the system as well as provide filtering based on both artifact and rela-

tionship types. Users of the traceability system will be able to customize the system to their traceability needs by defining artifact and relationship metadata; they then can create and register translators and integrators that translate the defined artifacts and create the required relationships.

To evaluate the utility of our approach, we plan to apply it to artifacts from an existing research project. We will develop translators and integrators and then use the traceability system to automatically generate traceability links between representative artifacts. Whenever possible, we will make use of techniques described in the literature to find these relationships.

The prototype will also allow us to perform a preliminary evaluation of the user interface. We will engage 2–5 computer science graduate students to perform several tasks (e.g., invoking a translator and an integrator, displaying different types of relationships). This preliminary evaluation will help us to detect useability problems so that they will not influence later evaluation.

In the next phase of our evaluation, a developer familiar with the research project will manually create traceability links in a commercial tool. We will record and compare the time involved in creating these relationships in the commercial tool and in our tool; we will also compare the number and types of links created by each of these approaches.

For the final phase of our evaluation, we will seed the research system with several defects. We will engage 5–10 software developers of similar programming experience who have no experience with the research project. The test subjects will receive training on our traceability system and the commercial tool as well as an overview of the research project and development environment. We will then ask them to locate and correct defects using three different sets of information:

1. No explicit traceability links represented—developers are free to use any tools with which they are familiar (e.g., *grep* and *find*).
2. Traceability links created and represented in the commercial tool.
3. Traceability links generated by our traceability system and represented in the tools that originally created the artifacts.

We will collect data on the number of defects found using each approach, the time required to find and fix each defect, and the ability to correct a defect without introducing new problems. We will use post-evaluation surveys to record the programmers’ experiences in using the different tools and to solicit their opinions on the various approaches. We will then analyze this data to determine the utility of our system in locating and fixing defects as compared with the other two approaches.

¹Fortunately, the open hypermedia community has developed techniques to facilitate application integration [6, 17, 18].

3. Related Work

Han [8] describes an information model for requirements and architecture management. The model provides the structure for two templates, a *System Requirements Document* and a *System Architecture Document*; a tool, TRAM, facilitates the creation of documents based on these two templates. TRAM's use of templates differs from our approach in that with a template approach the data must conform to a prescribed format. Our approach allows a user to create custom translators and integrators to handle different artifact formats and relationship types.

The Unified Software Development Process [9] defines trace dependencies between elements of its various models. For example, “[a] use-case realization [in the analysis model] . . . provides a straightforward trace to a specific use case in the use-case model [9, page 186].” The Unified Software Development Process differs from our approach in that it prescribes specific artifacts and relationships. In our approach, we allow users to create and maintain artifacts and relationships of interest to the project (by locating or creating appropriate translators and integrators). Thus, with appropriate metadata definitions, our approach can be adapted to various software development processes.

Pohl *et al.* [15] describe six meta-models for requirements and architectural artifacts. They then define dependencies between the meta-models. The introduction of explicit dependencies between use case and architecture scenarios allows dependencies between other requirements and architectural models to be “derived”. The derivation of dependencies is the same as “chaining” of relationships in our system. Thus, these meta-models can be realized in our system. A user would need to create or locate appropriate translators for the requirements and architectural artifacts. Integrators to create the explicit relationships would need to be developed as well. Our system could then manage the automatic generation and representation of Pohl's derived relationships.

4. Conclusion

We believe that the services of open hypermedia and information integration can be leveraged to provide an approach to traceability that facilitates the automated discovery, creation, maintenance, and viewing of relationships in tools that originally created the artifacts. Furthermore, these services can provide a customized view of the information space.

This research is still in its early stages. To evaluate our hypothesis, we plan to build a prototype traceability system along with representative translators and integrators to find relationships between requirements and architectural artifacts. We have demonstrated the feasibility of the cycle

represented in Figure 3 [1] and have already built several translators and integrators for text, HTML, and source code artifacts [2].

Although we believe that our approach can be successfully applied to requirements and architectural artifacts and relationships, our approach is not limited to these artifacts and relationships. We envision that, by developing an appropriate set of translators and integrators, the approach can address traceability concerns throughout a software development project.

References

- [1] K. M. Anderson and S. A. Sherba. Using Open Hypermedia to Support Information Integration. In *OHS-7/SC-3/AH-3*, pages 8–16, 2001.
- [2] K. M. Anderson, S. A. Sherba, and W. V. Lepthien. Towards Large-Scale Information Integration. In *Proceedings of the 24th International Conference on Software Engineering*, pages 524–535, Orlando, FL, USA, May 2002.
- [3] K. M. Anderson, R. N. Taylor, and E. J. Whitehead, Jr. Chimera: Hypermedia for Heterogeneous Software Development Environments. *ACM Transactions on Information Systems*, 18(3):211–245, July 2000.
- [4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, October 2002.
- [5] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.
- [6] H. C. Davis, S. Knight, and W. Hall. Light Hypermedia Link Services: A Study of Third Party Application Integration. In *Proceedings of the Sixth ACM Conference on Hypertext*, pages 41–50, Edinburgh, Scotland, September 1994.
- [7] J. Han. Specifying the Structural Properties of Software Documents. In *Proceedings of the 1994 International Conference on Computing and Information*, pages 1333–1351, 1994.
- [8] J. Han. TRAM: A Tool for Requirements and Architecture Management. In *Proceedings of the Australasian Computer Science Conference*, Gold Coast, Queensland, Australia, 2001.
- [9] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [10] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, January 2002.
- [11] Microsoft® Word®. <http://www.microsoft.com/office>.
- [12] K. Østerbye and U. K. Wiil. The Flag Taxonomy of Open Hypermedia Systems. In *Proceeding of the Seventh ACM Conference on Hypertext*, pages 129–139, Washington, DC, USA, 1996.
- [13] J. D. Palmer. Traceability. In R. H. Thayer and M. Dorfman, editors, *Software Requirements Engineering, Second Edition*, pages 412–422. IEEE Computer Society Press, 2000.
- [14] K. Pohl. *Process-Centered Requirements Engineering*. Research Studies Press Ltd., 1996.

- [15] K. Pohl, M. Brandenburg, and A. Gülich. Integrating Requirements and Architecture Information: A Scenario and Meta-Model Based Approach. In *Proceedings of the Seventh International Workshop on Requirements: Foundation for Software Quality (REFSQ'01)*, Interlaken, Switzerland, 2001.
- [16] B. Ramesh and M. Jarke. Toward Reference Models of Requirements Traceability. *IEEE Transactions on Software Engineering*, 27(1):58–93, January 2001.
- [17] E. J. Whitehead, Jr. An Architectural Model for Application Integration In Open Hypermedia Environments. In *Proceedings of the Eighth ACM Conference on Hypertext*, pages 1–12, Southampton, UK, April 1997.
- [18] U. K. Wiil and J. J. Leggett. The HyperDisco Approach to Open Hypermedia Systems. In *Proceedings of the Seventh ACM Conference on Hypertext*, pages 140–148, Washington DC, USA, March 1996.