ATRIUM
Architecture Traced from RequIrements by applying a Unified Methodology

Computing Systems Department
University of Castilla-La Mancha

Elena María Navarro Martínez

May 2007

PhD. Thesis
Elena María Navarro Martínez, Albacete, Spain
Printed in Spain
Design of the Cover Template: Pedro J. Molina
Design of the Cover: Elena María Navarro Martínez
Picture of the Cover: “Relativity”
Author: © M. C. Escher
Picture of the Back Cover: “Design of a chaple”
Author: © Partially extracted from “Leonardo's Notebooks”

ATRIUM
Architecture Traced from RequIrements by applying a Unified Methodology

Computing Systems Department
University of Castilla-La Mancha

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Presented by:	**Elena María Navarro Martínez**
Supervisors:	**Dr. Isidro Ramos Salavert**
	Dr. Patricio Letelier Torres

May 2007

Dedicatoria/Dedication

A Javier por demostrarme tu amor cada día, por tu apoyo incondicional y ayuda en todo y, sobre todo, por conseguir que cada día sea maravilloso por saber que estás ahí. Gracias vida mía.

A mis padres por todo lo que me han dado.A mi padre, José María, que siempre me ha dado su apoyo y su cariño a pesar de mis innumerables viajes. A mi madre, Elena, tú me has hecho ser quién soy, me has ayudado a crecer en todos los sentidos, a tener un espejo en el que mirarme para saber hacia donde ir. Gracias madre.

A mis hermanos, José Manuel y Roberto, por ser como sois, siempre dispuestos a alguna peleilla que nos hagar reir.

A Mar y Marisa, por hacerme sentir que puedo contar con vosotras, que vais a estar siempre ahí.

A mi sobrina Ruth, que ha traido alegría a mi vida con sus travesuras y su fantasía.

Agradecimientos/Acknowledgment

Esta tesis no habría sido posible sin dos personas que me han ayudado a lo largo de todo el camino. Gracias Isidro por ayudarme a andar al comienzo de esta tesis, sin tu ayuda todavía me encontraría perdida. Tú me tendiste la mano y me ayudaste a salir de la oscuridad. Gracias Patricio por haber sido mi guía, por ayudarme a discutir, por ayudarme a razonar y ver mis errores gracias a esa maravillosa hoja en blanco. Me has ayudado a hacerme creer que era capaz de conseguirlo.

No puedo olvidar a aquellos alumnos que de un modo u otro han colaborado con su esfuerzo e ilusión a hacer realidad las ideas que hay en esta tesis. Gracias David Reolid, Alejandro Rodríguez y Rubén Segura por ayudarme con vuestra energía.

A todos los miembros del Departamento de Sistemas Informáticos, como Gregorio Díaz y Julia Flores, que de un modo u otro me han ayudado en las mil vicisitudes que he vivido a lo largo de esta tesis. Especialmente a Paco Montero y Víctor López que durante estos últimos meses siempre han respondido cuando he tocado la puerta de su despacho.

A todos los miembros del grupo ISSI de la Politécnica de Valencia que me han acogido como a uno más y han hecho que las horas de laboratorio fueran más llevaderas. Gracias Alejandro, José Antonio, Cristóbal, Manolo, Nour, Rafael, María Eugenia y Rogelio por todos los buenos momentos que me habéis hecho pasar. En especial a Jenny e Isabel, por esos buenos ratos de cotilleos en la Vella.

A mi familia de Valencia, que me ha acogido, mimado y querido desde que os conocí. Especialmente a ti Encarna, por toda tu dedicación y tu cariño.

A todos mis amigas y amigos, por todo lo que hemos reído y llorado juntos a lo largo de los años, porque me habéis enseñado el valor de la amistad y por eso me habéis perdonado que os haya robado tantos momentos en estos últimos años. Gracias de corazón por ser como sois, especialmente a Nuria, Rosario, Mari Carmen, Charo y Susana.

A Antonio Quintanilla por ser mi "padre científico". Tú fuiste el primero en creer en mí, el primero que me apoyó cuando empecé a conocer este fascinante mundo de la investigación y el que siempre ha estado ahí haciéndome saber que puedo contar contigo más que como compañero como amigo. Gracias por ser como eres.

ABSTRACT

Requirements Engineering (RE) process must establish how to acquire, analyze and document requirements, i.e., focusing on the customer-defined services and constraints. The Requirements Specification is the foundation on which the system-to-be should be implemented and gives support for requirements validation and evolution over time.

Architectural models have a lower abstraction level than requirements, being closer to the end system, and they must be consistent with defined requirements in order to produce a valid solution. Developing precise Software Architectures from a structural and dynamic point of view and able to meet changeable requirements, is a challenging issue. A well-established process appears as the best approach to achieve this goal, leading to an increased productivity and a quality solution.

Recently, increased attention has been paid to how to establish and strengthen the relationships between requirements and architectural design. In particular, how the process must encompass the definition of requirements over time and their effects upon a system's architecture. This thesis presents our work in this field. It describes the methodology called ATRIUM (Architecture Traced from RequIrements applying a Unified Methodology) to guide the architecture definition that pays special attention to the functional and non-functional requirements that must be met by the system-to-be. In its definition, the Aspect-Oriented approach has been considered as cornerstone helping to specify properly the detected concerns of the system-to-be. In addition, we should mention that this work follows mainly the guidelines of the domain-oriented proposals. For this reason, customization of Models and Process has been considered mandatory in order to facilitate its application to different domains.

ATRIUM has been defined by means of a set of well-defined processes and Models that guides the analyst throughout its application. In addition, automation has also been used in those tasks that could be error-prone or cumbersome. A tool, called MORPHEUS, which allows the analyst the specification of the different Models and its later exploitation, supports ATRIUM. By means of its exploitation, ATRIUM has been put into practice in a real case study, the EFTCoR project, which has facilitated the validation of the proposal.

TABLE OF CONTENTS

"Nothing in life is to be feared. It is only to be understood. ." —
Marie Curie

CHAPTER 1
Introduction

1.1 INTRODUCTION

Nowadays, Software Development process is becoming a high difficult task because systems are more and more complex. This complexity comes from several factors that have a high impact on the outcomes of the process. Among them, it can be considered the new issues that must be taken into account to tackle either environmental or the stakeholders' needs that are evolving over time. The Software Development process must be established in such a way that it facilitates the users not only to describe their needs but also change them whenever they need so. The Lehman Law (Lehman, 1980) of continuous change provides compelling arguments about this need of change

The Requirements Engineering process (RE) establishes the foundation on which the system-to-be should be implemented. Therefore, it has to be able to identify and define this facility for the management of change into its artifacts in such a way that they can be traced to low level abstraction artifacts. It is especially relevant its traceability to the Software Architecture, because the reasoning about the capabilities of the system-to-be is established at this level.

In addition, quality is a critical issue for software development. Quality arises from several key points related to the software development and maintenance processes. Requirements are directly related to the quality in terms of customer satisfaction. This is specially affected by the Non-Functional Requirements, i.e., requirements as modifiability, performance, maintainability and the like. For this reason, satisfying them in the system-to-be, properly encompassed by the system architecture, is a compelling argument about the quality of the system-to-be.

1.2 Motivation

Requirements Engineering establishes the foundations to build the system-to-be. It is in charge of gathering the stakeholders' expectations and needs in order to produce a quality product. Software Architecture typically plays a key role as a bridge between requirements and implementation as (Garlan, 2000) and (Perry & Wolf, 1992) have stated (see Figure 1-1). It provides a system's high-level abstraction helping in its comprehension. Nevertheless, the problem for many software development organisations refers to which alternative, Requirements or Architecture, is the best starting point to proceed with the software development process. As (Nuseibeh, 2001) states, this election inevitably leads to artificially frozen requirements documents (frozen in order to proceed with the next step in the development life cycle) or leads to systems artificially constrained by their architecture that, in turn, constrain their users and handicap their developers by resisting inevitable and desirable changes in requirements. For this reason, one of the keys of the success of any software development process is that both the requirements of the system-to-be and its architecture are developed in an intertwined way and that their development is interleaved.

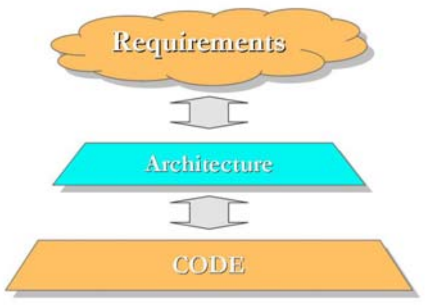


Figure 1-1 Software Architecture as a bridge

Developing Software Architectures enough settled for its use, but also dynamic to address changeable requirements, is not an obvious issue. For this reason, several problems arise, as (Nuseibeh, 2001) state, when requirements and architecture models, which are developed concurrently (see Figure 1-2), have to be changed. A change, which is usually initiated when *new requirements* on existing products or by new products, that needs to be incorporated in the system-to-be. The incorporation of this change typically leads to:

- architectural evolution related to changes to the components that make up the architecture, the relations between them, etc;
- component evolution related to the incorporation of new and changed requirements on the components functionality which can affect to them not only internally but also to their interaction;
- product evolution in terms of both new versions and run-time evolving system, i.e., dynamic architectures.

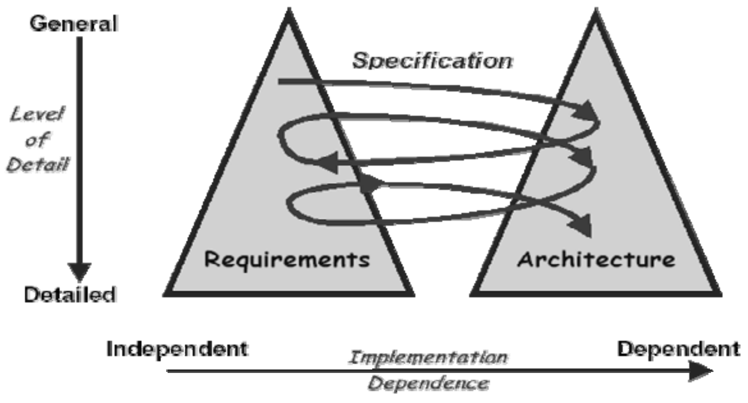


Figure 1-2 An intertwined process to define Requirements and SA (extracted from (Nuseibeh, 2001))

In addition, software requirements could have, implicitly or explicitly, information related to the Software Architecture. How this information can be incorporated in its specification is critical to develop a system meeting the established the needs. Moreover, other limits appear when distinct stakeholders state the same requirement in different ways. As (Bosch, 2000) state:

"since the Software Architecture constrains the quality requirements, the driving quality attributes should have a major influence on the architecture of a software system, in some cases, even more that the functional requirements of the system".

In addition, the selected aforementioned architecture also compels a high constraint on the requirements offered by the system. It shows that an request to provide a feedback from architectural model to the requirements model would help to refine, in an iterative process, both the definition of the problem and the solution provided. This points out that both Requirements and Software Architecture are highly intertwined.

Therefore, the challenge to be faced is the definition of a formalized and flexible process to acquire the requirements to be compiled in an architecture, and manage their concurrent evolution over time is a challenge. This issue is going to constitute the main objective of this thesis, the description of a methodology, which has been called ATRIUM, that provides the analyst with proper guidelines and artifacts for the specification of Requirements and Software Architecture in an intertwined way. It has as a mandatory constraint the establishment of proper traceability between the artifacts all the way down from the requirements to the architecture.

In addition, a key element included in the definition of ATRIUM is the integration of the Aspect Oriented approach as was described by (Elrad et al., 2001a). It is aimed at providing a proper separation of concerns both functional and non-functional (performance, safety, security, etc) of the system. A great deal of work related to this paradigm has been performed from requirements (Rashid et al., 2003) to architecture (Cuesta et al., 2005), design (Suzuki & Yamamoto, 1999) and implementation (Kiczales et al., 1997). In this context, how to identify, specify and trace each concern across the software lifecycle satisfying the closure property (Elrad et al., 2001b) is an open issue that has not been solved up to date, as (Baniassad et al., 2006) state. ATRIUM has been defined to identify, specify and rightly trace this Separation of Concerns (SoC) from the requirements to the architecture.

1.3 METHODOLOGY OF DEVELOPMENT

Software Engineering is frequently criticized because disconnections between theoretic research and its practice emerge as observed in Figure 1-3. In this thesis, special attention has been paid to this aspect. For this reason, all the work presented in this thesis has been performed in a collaborative way in the context of the CICYT project called DYNAMICA. One of the main aims of this project is to develop a framework for dynamic Software Architecture applicable to the development of tele-operated systems. This has facilitated that all the theoretical proposals have been validated by its application in this real case study.

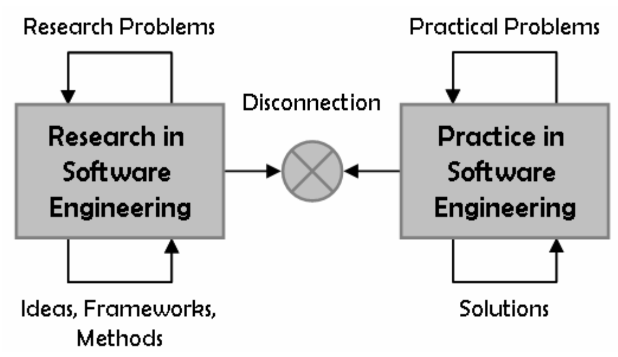

Research Problems
Practical Problems
Disconnection
Research in Software Engineering
Practice in Software Engineering
Ideas, Frameworks, Methods
Solutions

Figure 1-3 Disconnection between Research and Practice in Software Engineering (Moody, 2000)

DYNAMICA exhibits several specific needs in terms of requirements specification, such as, the variability inherent in the family of robots to be handled; the high incidence of non-functional requirements (reliability, performance, safety, etc) which crosscut functional ones; the need to evaluate alternative designs meeting system requirements; and, finally, a large specification where an appropriate organization is unavoidable. In this context of complexity, associated with the requirements specification and the emphasis in achieving reuse through a product family specification, we found favourable conditions for the application of Action-Research, as described by (Baskerville & Wood-Harper, 1996). It allows us to solve a real problem, by means of a continuous refinement of our approach, throughout the 3 years of project duration.

Action-Research has emerged as a proposal to break the disconnection between research and practice. With this aim, it has two clear goals: to generate profits for the customer of the research; and, at the same time, to generate relevant "knowledge of research" (Kock & Lau, 2001). Therefore, Action-Research is an approach to research highly collaborative between research, researchers and practitioners, focused on both theory and practice and carried out by means of a cyclic process. In addition, Action-Research describes a class of methods, which share the following characteristics (Baskerville & Wood-Harper, 1996):

- To be oriented to action and change.
- To be focused on a problem.
- To have an organic model of process that entails systematic stages and some iterative ones.
- To be collaborative between the participants.

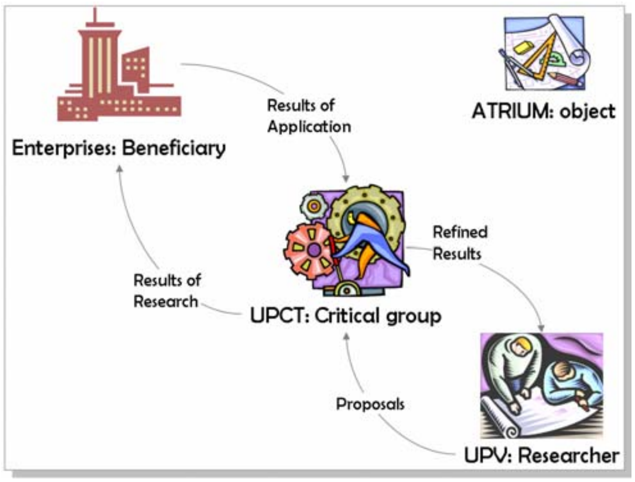


Figure 1-4. Applying Action-Research: actors in DYNAMICA

In a more formal analysis of the participants in Action-Research, (Wadsworth, 1998) identifies the four types of roles (sometimes the same person or team plays more than a role) that are described in the following along the with actors playing that roles in DYNAMICA (see Figure 1-4):

– The *researcher* or team who proactively perform the research process. This role has been played by us (UPV).

– The *object* under research, i.e., the problem to be solved. In our case, the object of research has been the development of a methodology for the description of Software Architecture meeting the established requirements. Despite the fact that this methodology had to be validated in the context of tele-operated systems, it should be appropriate for the development of any kind of systems.

– The *reference critical group* who receives the results of the research and participates in the research process (although less actively than the researcher). This group is integrated for both persons who know they are involved in a research and other who do not know it. This role has been played by the University Polytechnic of Cartagena. They have a deep knowledge of this kind of systems, providing us with the proper background for our work.

– The *beneficiary* (stakeholder) of the research who is expected to exploit the results of the research although it does not take part in the process. In this case, it is going to be any enterprise that has software development, particularly of tele-operated systems, as its main activity.

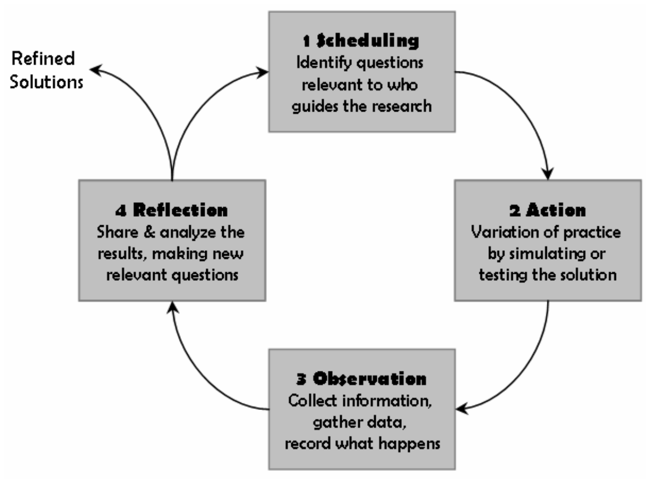

Refined Solutions
1 Scheduling
Identify questions relevant to who guides the research
2 Action
Variation of practice by simulating or testing the solution
3 Observation
Collect information, gather data, record what happens
4 Reflection
Share & analyze the results, making new relevant questions

Figure 1-5. Cyclic character of Action-Research

Considering the above assigned responsibilities, to put into practice the Action-Research methodology was necessary to establish the activities that guide the research. With this aim, (Padak & Padak, 1994) have established the following steps (Figure 1-5):

I. Scheduling: to identify the questions relevant to the guidance of the research, directly related to the object under research and susceptible to obtain the answer. During this activity, we wonder questions as:

- Which kind of requirements specification provides abilities to analyze alternatives?
- Which kind of requirements specification provides mechanisms for traceability between requirements and Software Architecture?

Several alternatives were studied and examined to answer all the questions that emerged during the development of this work.

II. Action. The studied alternatives were put into practice to validate the results.

III. Observation. To gather information, data and recording the results of the application of the proposals.

IV. Reflection. To share and analyze the results with other stakeholders in order to improve the proposals and obtain other more refined.

These activities were performed iteratively, in such a way that we were getting more refined solutions by completing cycles. Each cycle supposed that new ideas and proposals were run and validated in the following cycle as is observed in Figure 1-5. The cyclic peculiarity of this process determined to re-evaluate and raise questions and alternatives weighing up diagnosis and reflection.

1.4 SCHEME OF THE WORK

This work has been structure in the following chapters:

- Chapter 1 describes the main questions that have led to the development of this thesis.
- Chapter 2 describes an overview about the current situation in Requirements Engineering and Software Architecture. Special attention has been paid to how both disciplines are seen from the Aspect-Oriented approach.
- Chapter 3 presents an analysis of the current proposals for the intertwining between requirements and Software Architecture, giving special emphasis to that elements related to the Aspect-Oriented approach. This description provides a good insight to present the proposal of this thesis: ATRIUM. It is briefly introduced in this chapter.
- Chapter 4 describes some concepts to improve the comprehension of the rest of this work. Therefore, a brief introduction to the case study employed throughout this thesis is presented. In addition, a sketch of PRISMA, the Aspect-Oriented Architectural Description Language used as target language is presented.
- Chapter 5 explains the *Define Goals* activity of ATRIUM, describing both the artifacts used to describe the requirements of the system-to-be and the process for its description and exploitation.
- Chapter 6 describes a detailed example of how the activity *Define Goals* is applied in the context of a real-life case study.
- Chapter 7 depicts the *Define Scenario* activity of ATRIUM related to the description of the scenarios realizing the established requirements. A model and a process for its description are shown in this chapter.

- Chapter 8 portrays how the *Synthesize and Transform* activity is in charge of transforming automatically the described scenarios into a draft of the architecture. The mechanism used are justified and explained.
- Chapter 9 describes the developed tool called MORFPHEUS that provides support to the description of the different artifacts and their exploitation.
- Chapter 10 includes the main conclusions achieved with this thesis along with the future works.

"It ain't what you don't know that gets you into trouble. It's what you know for sure that just ain't so." —
Mark Twain

CHAPTER 2

Requirements and Software Architecture: considering the Aspect-Oriented Approach

2.1 INTRODUCTION

There are compelling economic arguments why an early understanding of stakeholders' requirements leads to systems that more closely meet stakeholders' expectations. For this reason, the introduction of an appropriate Requirements Engineering (RE) process in the software development is one of the keys to achieve the success of the project. RE must show how to acquire, analyze and document requirements focusing on the customer-defined services and constraints. RE establishes the foundation on which the system-to-be should be implemented and gives support for requirements validation and evolution over time.

Similarly, there are persuasive arguments why an early understanding and construction of the Software Architecture (SA) provides the foundations to gather system requirements and constraints, evaluate a system's technical feasibility, and evaluate alternative design solutions. Architectural models are a bridge between requirements and the system-to-be (Garlan, 2000) providing us with a lower abstraction level than requirements. They are used as intermediate artifacts to analyse whether the requirements are met or not. However, developing precise Software Architectures from a structural and dynamic point of view and being able to meet changeable requirements, is not an obvious issue (Nuseibeh, 2001). A guided process appears as the best approach to achieve this goal, leading to an increased productivity and a quality solution.

This chapter focuses on the state-of-the art of some of the most relevant proposals for Requirements and Software Architecture giving a brief introduction to the Aspect-Oriented approach in both fields. In order to provide a better comprehension it has been structured as follow. Section 2.2 presents a brief resume of the process of Requirements Engineering. In addition, the most relevant approaches in RE, which are exploited for the RE process, are described as well. Section 2.4 gives an overview about the Software Architecture field, describing its main concepts. Section 2.4 introduces Aspect-Oriented Software Development, presenting its main concepts and why this approach has emerged. Finally, the conclusions round up this chapter.

2.2 AN OVERVIEW ABOUT REQUIREMENTS ENGINEERING

A successful software system development is specially affected by how it can satisfy the user needs, providing an appropriate and cost-effective solution. That is why `Requirements Engineering` is the major process applied to anchor development activities to any real-world problem. Broadly speaking, RE is the process of discovering that purpose, by identifying stakeholders (including customers, users and developers) and their needs, and documenting them in an amenable way for analysis, communication, and subsequent implementation. (Zave, 1997) provides one of the most well known definitions for RE:

> *"Requirements Engineering is the branch of Software Engineering concerned with the real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behaviour, and to their evolution over time and across software families".*

This definition faces several key issues. On the one hand, it is related to "real-world goals" that have to be satisfied when a system is developed, i.e., what the system behaviour will be. These goals are named `Requirements` and they describe the activities of the system, such as its reaction to input, its different states before and after any activity, etc. These kinds of requirements are known as `functional requirements` and they usually describe interactions between the system and its environment. However, this description does not fulfil the problem if it does not address that restrictions the system must show, that is, adaptability to different environments, performance on memory usage, etc. These restrictions are called `non-functional requirements`. They are very relevant for the software development because they do not only describe requirements demande by the stakeholders, but they are going to establish the scope of the set of likely solutions as well.

On the other hand, (Zave, 1997) refers to "precise specifications". They can range from an understanding of the problem being solved to its detailed specification. This specification has to be complete, consistent and unambiguous. These provide the basis for analysing requirements in terms of the well known Validation and Verification (V&V) activity:

– `Validation,` according to (Nuseibeh & Easterbrook, 2000), it is the process of establishing that the elicited requirements and models provide an accurate description of stakeholder requirements.

– `Verification` is in charge of determining if the implementation satisfies the specification.

The major problem of these activities arises from their relation with the stakeholders. It is because they found severe difficulties to articulate their needs and to reconcile their different goals. This usually leads to misalignments between their expectations and the final result. For this reason, mechanisms of `trade-off` with and between different stakeholders and `decision support`, which help to achieve agreements and facilitate the negotiation, are always recommendable in any RE approach.

With this scenario in mind, several methods have emerged that assist with the process of Requirement Engineering. `Goal-Oriented Requirements Engineering` (see section 2.2.1) is a well-known approach that helps to define the objectives a system must meet, agents fulfilling them, alternatives to be assessed, etc. On occasions, users find difficult to describe these objectives. In these cases, `Scenario-Based` (see section 2.2.2) approaches are used to provide a better understanding of some aspect of using a system. Among the different proposals, following the scenario-based approach, are the `Use Cases` that describe the system in terms of the user interaction with the system-to-be. `Problem Frames` (see section 2.2.3) are also a well-known approach that supports the RE process by modelling the relation of the system-to-be with its context. `Viewpoints` (see section 2.2.4) is another approach very appropriate to find out conflicts owing to the different views that stakeholders hold. `Features Model` (see section 2.2.5) are gaining more and more attention lately as a mean to simplify the management of the requirements by encapsulating both functional and quality requirements.

But, RE is not only a process of acquiring requirements; it also facilitates communication among stakeholders. How the requirements are documented, it is a crucial point to ensure an appropriate analysis and validation. Several alternatives have appeared related to the specification languages and notations, which range from informal to semi-formal and formal languages:

– `Natural Language` (Ambriola & Gervasi, 1997) proposes the use of an informal and often used language to write requirements specifications that are plain to the user. Nevertheless, it shows misunderstandings between the stakeholders because of the ambiguity and excessive flexibility of the language.

– `Structured Natural Language` is a restricted form of natural language for requirements specification. Its advantage is that it maintains most of the expressiveness and understandability of the Natural Language but ensures a degree of uniformity throughout the specification by limiting the used terminology and, sometimes, using templates (Duran, 2000). However, templates cannot provide by themselves a structured mechanism for requirements. Both this and the previous approach show a poor usefulness to validate and verify the requirements. (Osborne & MacNish, 1996) address this issue providing natural language processing techniques to aid the development of formal descriptions from requirements expressed in a controlled natural language.

– Several `formal approaches` have appeared to address the problems showed by the previous ones. Generally speaking, a formal specification is the expression, in some formal language (Z, VDM, OBJ, etc) and at some level of abstraction, of a collection of properties that the system-to-be should satisfy (Lamsweerde, 2000). These approaches provide higher-quality specifications and the basis for their automated support to produce animations, generate concrete scenarios, etc. However, they also show some problems as restrictions in terms of expressiveness.

Finally, the Zave's definition (Zave, 1997) refers to specifications' "evolution over time and across software families", surfacing the reality of a changing world and the need to reuse partial specifications. It is obvious that a method to read, navigate, query and change requirements documentation is needed, i.e., the `Management of Change` of requirements. This activity has to do with the assessment of the impact of the change requests, and their management trough the software lifecycle. Related to this activity, the establishment of proper traceability mechanisms is a demanding need. The `Traceability` of requirements has been described by (Gotel, 1994) as:

> *"…the ability to describe and follow the life of a requirement, in both a forwards and backwards direction."*

Mainly, it assists to reconcile the changes in user's needs with the software, decrease costs of acquiring critical knowledge, assess consequences and impact of a change, etc. This means that the appropriate introduction of requirements traceability helps to address the problems arisen by the evolution of the requirements over time. Finally, the specifications across software families have become a reality by the introduction of `Variability Management` techniques (see section 2.2.6). They provide facilities to describe the assets that will be shared across a software family.

Some of the most relevant concepts in the Requirements Engineering have been briefly described above. In this sense, either (Nuseibeh & Easterbrook, 2000) or (Lamsweerde, 2000) provide a wider description of these concepts and the field in general. The idea behind this introduction is to make know the reader the main concepts used by the approaches introduced in the following sections.

2.2.1 Goal-Oriented Approach

In the context of Requirements Engineering, the *Goal-Driven Requirements Engineering* approach (Lamsweerde, 2001a) has proven its usefulness to elicit and define requirements. More traditional systems analysis techniques, such as Use Cases, focus only on establishing the features (i.e. activities and entities) that a system will support. Nevertheless, Goal-based proposals, such as (Chung et al., 2000) or (Dardenne et al., 1993), focus on *why* systems are being constructed by providing the motivation and rationale to justify the Software Requirements. They are not only useful for analyzing goals, but also for elaborating and refining them.

There are a wide number of proposals ranging from elicitation to validation activities in the RE process (see (Kavakli & Loucopoulos, 2005) for an exhaustive survey). However, some concepts are common to all of them:

– `Goal` describes why a system is being developed, or has been developed, from the point of view of the business, organization or the system itself. In order to specify it, both functional goals, i.e., expected services of the system, and non-functional goals related to the quality of service, constraints on the design, etc should be determined. These goals can be described using formal languages, as for instance temporal logic, helping in the process of verification of the specification.

– `Agent` is any active component, either from the system itself or from the environment, whose cooperation is needed to define the operationalization of a goal, that is, how the goal is going to be provided by the system-to-be. This operationalization of the goals is exploited to maintain the traceability throughout the process of software development.

– `Refinement Relationships`: AND/OR/XOR relationships allow the construction of the goal model as a directed graph. These relationships are applied by means of a refinement process (from generic goals towards sub-goals) until they have enough granularity to be assigned to a specific operationalization.

It must be pointed out that one of the main advantages exhibited by this approach is that it introduces mechanisms for reasoning about the specification. It facilitates the process of evaluating designs or alternative specifications to the system-to-be.

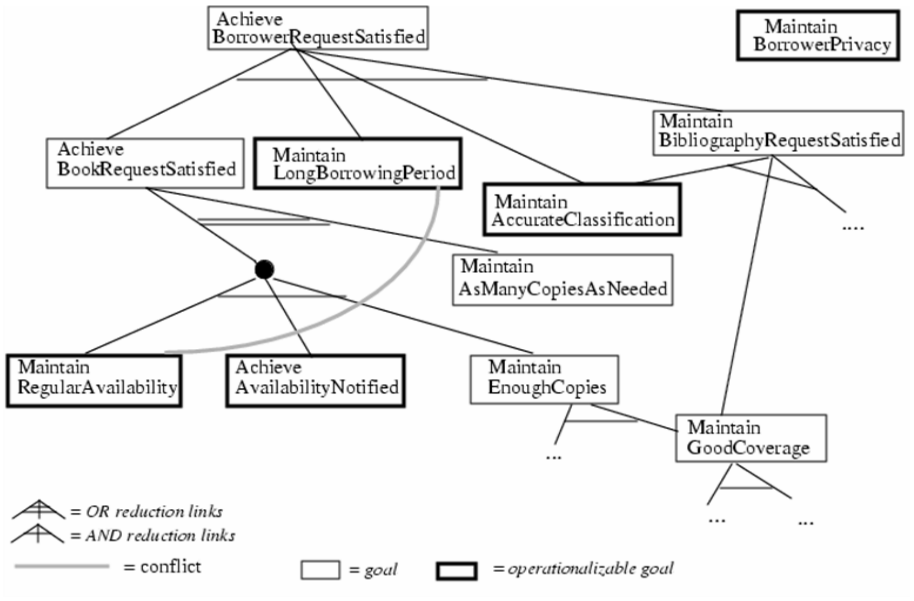

Achieve BorrowerRequestSatisfied
Maintain BorrowerPrivacy
Maintain BibliographyRequestSatisfied
Achieve BookRequestSatisfied
Maintain LongBorrowingPeriod
Maintain AccurateClassification
Maintain AsManyCopiesAsNeeded
Maintain RegularAvailability
Achieve AvailabilityNotified
Maintain EnoughCopies
Maintain GoodCoverage
= OR reduction links
= AND reduction links
= conflict
= goal
= operationalizable goal

Figure 2-1 Partial description of a Goal graph (extracted from Dardenne et al.)

2.2.2 Scenario-based approach

This approach tries to facilitate the discussion of the stakeholders in the RE process. They are based on the description of examples of use of the system-to-be instead of using abstract description so that the stakeholders can criticize and modify them more easily. Each scenario depicts one or more possible interactions, providing a better understanding of some aspect of using a system.

Quite different styles have been used for scenarios description, such as, textual narratives, storyboards, video-mocks up and written prototypes. (Leite et al., 2000) describe a good overview about these alternatives. It must be pointed out that they consider Use Cases as one of the styles for scenarios descriptions. In addition, CREWS project (Co-operative Requirements Engineering with Scenarios) (Maiden, 1998) is also a quite well know project that has detected the wide range of interpretations and uses of the current proposals. Most of the proposals describe a scenario by means of:

– `Goals` that are descriptions of the expected behaviour of the system and the users.

– `Episodes` that are descriptions of the basic flow of events.

– `Exceptions` that are descriptions of the unexpected behaviour and its treatment.

– `Results` that are descriptions of the expected result when the scenario finishes.

Depending on the proposal, different notations are used to describe each concept. For instance, Leite et al. propose the use of a structured natural language to perform the episodes description, whereas the other elements are described by means of natural language. However, one of the most well known approaches is the Use Cases (Cockburn, 2000) that are introduced in the following section.

Use Cases

They are perhaps one of the most popular approaches to requirements specification. They have been widely embraced by the industrial community due to their straightforward notation and application. These properties allow stakeholders to easily understand them, and this contributes to the elicitation and validation of the requirements. Another factor that denotes their popularity is that Use Cases are the only notation included in UML for modelling requirements. In a Use Case diagram (Figure 2-2), we mainly distinguish the following elements:

– `Use Cases` represent an atomic functionality (there is no hierarchical refinement when Uses Cases are specified or identified) that the system offers to the environment for achieving some specific goal. Basically, the detailed specification of a Use Case shows the dialogue between the environment and the system to obtain a desired service. In addition to the communications steps, templates for specifying Use Cases usually include other artifacts, such as preconditions, post-conditions, alternative steps or exceptions, non-functional requirements, etc. They are mainly used to describe the functionality to be provided by the system.

– `Actors` represent the environment of the system-to-be and can be users, devices or any other system that interacts with the system being developed. The *Actor* name describes which role it plays in that interaction.

– `Relationships` which include `Communication` (to represent the interaction between the Actor and the Use Case); `Generalization` (applicable both to Uses Cases and Actors to establish specialization hierarchies); `Include` and `Extend` (to factorize an original Use Case).

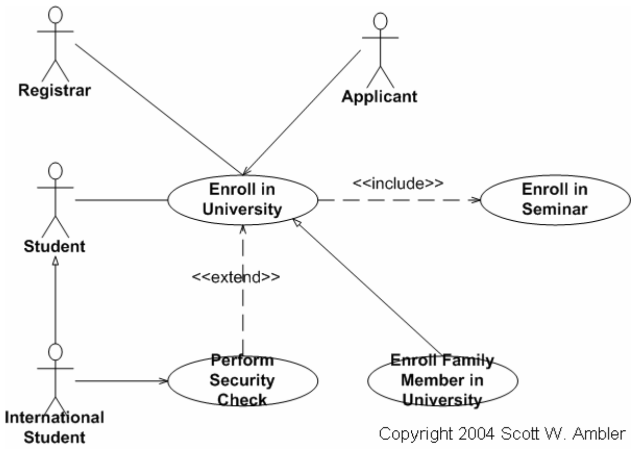


Figure 2-2 Use Case approach

By means of the *Include* and *Extend* unidirectional dependencies, the Use Case model offers expressiveness for specifying relationships between requirements. The *Include* relationship permits a Use Case to be reused and the *Extend* relationship simplifies a Use Case. Thus, this factorization can be for reuse or for simplification purposes depending on whether *Include* or *Extend* relationship are used, respectively. Additionally, as stated above, Uses Cases do not allow for hierarchical refinement, which implies a lack of consensus related to both the proper granularity level of functionality that a Use Case should have and the precise exploitation of *Extend* and *Include* relationships. Consequently, the simple and easy notation of Use Cases is actually one of their major problems. In situations where modelling has to be rigorous and/or precise, Uses Cases usually exhibit problems with regard to their interpretation because of their overloaded semantics and lack of consensus.

In addition, this approach is not appropriate when the system-to-be is highly demanding in terms of non-functional requirements. It is because it is mainly focus on the description of the interaction between the user and the system.

The traceability in this approach depends on the analyst ability to introduce mechanisms that help in the process.

2.2.3 Problem Frames

This approach has been introduced by (Jackson, 2000) in order to change the usual predisposition in software development to think in the solution and not in the problem being solved. For this reason, this approach focuses on the problem analysis and structuring. The main idea behind this proposal is to find out the main characteristics that help to determine a class of problems so that the same type of models can be used.

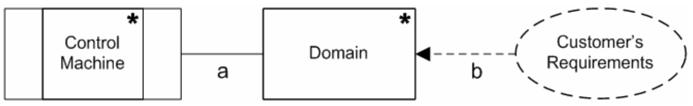


Figure 2-3 Problem Frames approach

Problem Frames approach presumes that some knowledge of the application domain and context has been gathered, as for instance, through a process modelling, so that a Problem Frame can be determined. A Problem Frame is graphically described by means of a context diagram (Figure 2-3) that identifies:

- `Control Machine` is a description of the piece of software, which the customer desires. It illustrates the expected effects of its execution and its interface with the domain.
- `Domain` is a part of the world that is affected by the effects of the Control Machine.
- `Requirements` are the properties in the Problem Domain that the customer wants to observe by means of the shared phenomena *b*. *b* shows the effects of the Control Machine that interacts with the Problem Domain via the shared phenomena *a*.

One of the main advantages of this approach is its ability for requirements reuse, detecting similarities among tasks. They also provide support for traceability because the decomposition strategy, from problems to sub-problems followed by the Problem Frames, allows the analyst to map sub-problems directly to their solutions. However, there are no works, to the best of our knowledge, that introduce support for non-functional requirements.

2.2.4 Viewpoints

Several proposals, such as (Finkelstein et al., 1992) and (Kotonya & Sommerville, 1996), have introduced the use of Viewpoints to both obtain and organize the requirements. In this approach, the system-to-be is defined according to the context where it is going to perform its main computation. With this aim, it is defined considering all the involved stakeholders and assigning a different viewpoint to each party. Each established viewpoint can be used to catalogue some stakeholders and, thus, different sources of information.

Most of the proposals, following this approach, distinguish two different kinds of viewpoints:

- `Direct Viewpoints`: they describe those persons or systems that directly interact with the system-to-be. They are usually customers that are going to receive the service to be provided. They provide detailed requirements, usually about features and interfaces of the system-to-be.
- `Indirect Viewpoints`: they describe those stakeholders that do not interact directly with the system but have some interest in some or all of the service which are going to be provided by the system-to-be. They usually provide organizational requirements and constraints.

There is not a standard notation that can be used to describe Viewpoints but every proposal identify different concepts as relevant for the description of a Viewpoint. However, most of them recognize that it is necessary to identify: the `kind of Viewpoint` and the `specification` following that notation (Petri nets, statecharts, etc.) that is more appropriate to the domain that the viewpoint belongs to. The main advantage this approach provides is the facility to find out conflicts between requirements stated by different stakeholders. In addition, some notations provide support for traceability bottom-up and top-down. For instance, (Nuseibeh et al., 1994) use the *work record* (Figure 2-4) to document every action or process the viewpoint has suffered throughout its history. They also provide specific notation to depict with non-functional requirements.

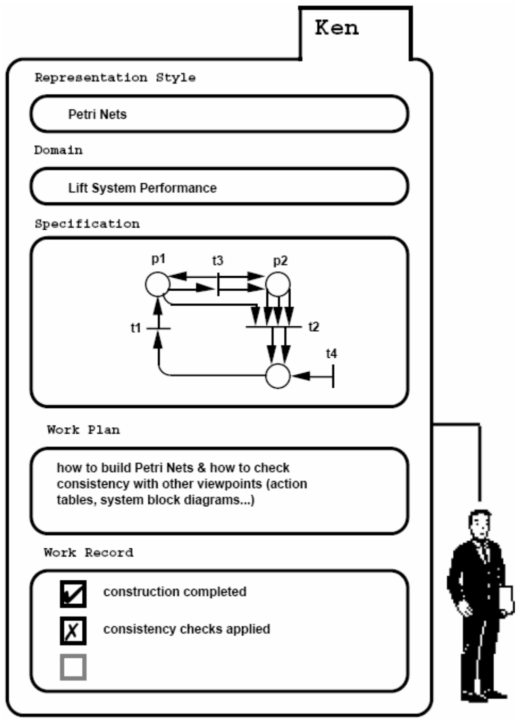

Ken
Representation Style
Petri Nets
Domain
Lift System Performance
Specification
p1
t3
p2
t1
t2
t4
Work Plan
how to build Petri Nets & how to check consistency with other viewpoints (action tables, system block diagrams...)
Work Record
construction completed
consistency checks applied

Figure 2-4 An example of a viewpoint (extracted from (Finkelstein et al., 1992))

2.2.5 Features

Features Diagrams are a popular approach for requirements modelling in two areas: Software Product Lines (SPL) (Clements & Northrop, 2001) and Dynamic Software Architectures (DSA) (Oreizy et al., 1998). SPLs allow the analyst to identify and specify shared product line assets so that the products implemented can be varied. DSA allows the system to evolve in both composition and configuration, by supporting ad-hoc features at run-time. Features Diagrams have been massively used for modelling the variability, i.e., the commonalities and differences existing in a SPL or a DSA (see section 2.2.6).

A wide diversity of proposals have emerged during the last year that describe different languages for modelling Features Diagrams. (Schobbens et al., 2006) describe a survey where the most relevant proposals are described and analysed. However, despite this diversity, most of the proposals describe a Features Diagram as a tree composed of nodes and directed edges. The nodes are called

`Features` and the edges determine the relationships among the *features*. In order to describe what a *feature* is, the definition provided by (Kang et al., 1990) can be used:

"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems."

The edges are going to determine if a feature is `mandatory`, `optional` or `alternative`. *Mandatory features* are always included in every product. *Optional features* can be included or not, depending on the product. *Alternative features* describe when only some features from a set can be included in a product. Figure 2-5 shows an example of a Feature diagram where each kind of feature is illustrated.

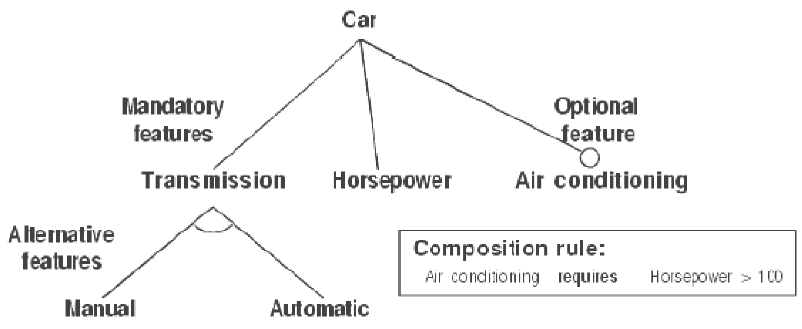


Figure 2-5 An illustrative example of a feature diagram (extracted from Kang et al.)

One of the main advantages of this approach is that it provides a specific notation for the description of the variability. In addition, it also simplifies the management of the requirements because they can be used as groups described from the point of view of one or several stakeholders. This approach has the advantage of dealing with both functional and non-functional features of the system-to-be. The traceability towards others artifacts in the software development can also be established by means of the *implementation* links.

2.2.6 Variability Management

This approach helps the analyst to delay the decision of what functionality or quality aspects will be incorporated in the final system as long as possible. This approach has been successfully applied in two areas: Software Product Lines and Dynamic Software Architectures.

Traditionally, most of the proposals supporting this approach have dealt with the management of variability at the architectural level by establishing a central architecture, along with a set of components that can be evolved or integrated according to the system needs. When variability identification is delayed at the architectural level, a problem related to product lines and dynamic architectures arises because the number of potential systems and the capability of adaptation, respectively, are more limited (Clements & Northrop, 2001). Thus, the early identification of the variability which is performed in the requirements phase, which has been called `early variability`, is a great advantage as (Gurp et al., 2001) have stated (and described in Figure 1-5).

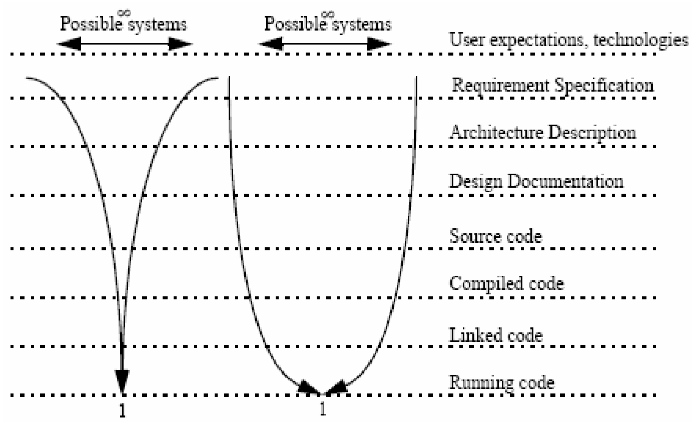


Figure 2-6 Bottleneck with early and delayed variability (extracted from (Gurp et al., 2001))

There is no a widely accepted notation to specify the variability in the requirement phase. However, other approaches such as Use Cases have been extended to support the variability management. Instead of describing the different notations available for modelling variability, we present below the necessary concepts as designated by (Trigaux & Heymans, 2003) as follows:

– `Representation of common and variable parts`. The notation should allow the analyst to express those assets that are shared among different products of a product line, or among different instances of Software Architecture and those that are specific to a specific product or instance. The notation should be able to represent both variation points and variants. A `variation point` indicates a specific variability in the specification. A `variant` is a specific realization of variability in a specific variation point. Each variation point has a `time link`, i.e., when the variability is removed during the development process: design, analysis, running, etc. It is also important to define the `multiplicity` (both maximum and minimum) in each variation. The multiplicity determines how many variants must exist at the same time in a product or architecture when the variability is removed.

- Distinction between `types of variability`. The notation must allow the analyst to express the different types of variability in the following way: a) `optional` - when some specific variants can be selected in the instantiation process, b) `alternative` - when only a single variant can be selected and c) `optional alternative` - when either cero or one alternative can be selected from those available.
- Representation of `dependencies` between variable parts. The variants frequently have dependencies among them that have to be represented. For instance, some the dependencies that are used are `Require` (when a variant must be selected if another variant is present) or `Exclude` (when the selection of a variant implies another variant can not be selected). A more extended set of relationships in the requirements stage is presented in (Bühne et al., 2003).

2.3 An overview about Software Architecture

There is not a standard and recognized definition of Software Architecture. On the contrary, a wide set of definitions can be found in the bibliography. We introduce those we consider the most relevant:

> *"Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design." (Perry & Wolf, 1992)*

> *"Beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives." (Garlan & Shaw, 1993)*

"The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time. (Garlan & Perry, 1995)."

"Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. This definition is intended to encompass a variety of uses of the term architecture by recognizing their underlying common elements. Principal among these is the need to understand and control those elements of system design that capture the system's utility, cost, and risk. In some cases, these elements are the physical components of the system and their relationships. In other cases, these elements are not physical, but instead, logical components. In still other cases, these elements are enduring principles or patterns that create enduring organizational structures. The definition is intended to encompass these distinct, but related uses, while encouraging more rigorous definition of what constitutes the fundamental organization of a system within particular domains." (ANSI, 2000)

The core of these definitions is the notion that the architecture of a system describes its gross structure. This structure surfaces the top-level design decisions, including issues such as how the system is composed of interacting parts, where the main pathways of interaction are, and what the key properties of each part is. The Software Architecture allows designers to reason about the ability of a system to satisfy certain requirement. For this reason, the Software Architecture typically plays a key role as a bridge between Requirements and Implementation. It is because an architectural description should include enough information by providing an abstract description of a system. It allows that a high-level analysis and critical assessment can be performed. In addition, they also suggest that it is a first draft for system construction and composition so that the implementation of the system can be properly planed.

However, one the main issue is how the architecture is represented. The informality of most box-and-line depictions of architectural designs leads to a number of problems. The meaning of the design may not be clear. Informal diagrams cannot be formally analyzed for consistency, completeness, or correctness. In addition, the architectural constraints assumed in the initial design cannot be enforced as a system evolves if a precise description is not established.

In response to these problems, a number of researchers in industry and academia have proposed formal notations for representing and analyzing architectural designs. Generically referred to as `Architecture Description Languages` (ADLs) [1], these notations usually provide both a conceptual framework and a concrete syntax for characterizing Software Architectures. They also typically provide tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions. A wide diversity of languages have emerged during the last decades, each one having distinctive capabilities. For instance:

- *C2* (Medvidovic, 1996), supports the description of user interface systems using an event-based style;
- *Darwin* (Magee et al., 1995) supports the analysis of distributed message-passing systems;
- *Rapide* allows architectural designs to be simulated, and has tools for analyzing the results of those simulations;
- *Wright* (Allen & Garlan, 1994) supports the formal specification and analysis of interactions between architectural elements.

Independently of the specific properties these ADLs have, they should be expressive enough to facilitate as well-defined Software Architectures as to be used for ensuring the requirements are properly meet. Nevertheless, they also play an interacting role with requirements. It has been described that they play a critical role to determine the feasibility of the requirements, as (Andrade & Fiadeiro, 2003) suggest, and to support the decision-making process at this level, as (Miller & Madhavji, 2001) have analysed.

[1] Interested readers are referred to (Medvidovic & Taylor, 1997) and (Cuesta, 2002) for a detailed analysis of these ADLs

2.3.1 Concepts for Software Architecture Descriptions

Despite the wide set of ADLs defined up to date, there are some concepts that are transverse to all of them. In order to facilitate a better comprehension of this work, the most relevant ones are presented in the following sections. In order to obtain a deep insight into Software Architecture (Garlan, 2001) and (Shaw & Clements, 2006) are recommended. The former shows the emerged tendencies over the last decade and the latter set out the new challenges appearing for the next one.

Components

Normally, the `components` of a system are treated as "black boxes" about which nearly nothing is known, except for the way they connect with other architectural elements. They are the base to modularize the functionality of the system with a high level of encapsulation. It is because their interface (or interfaces) are well defined describing clearly the service they require and/or provide. They have been defined by (Szyperski, 1998) as:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

This concept is applied in other contexts, specially for Components-Based Software Development (CBSD) and the use of components Commercial off-the-shelf (COTS). Exploiting CBSD, the core computation is separated from connectivity between the elements to provide such a computation (Szyperski, 1998). In this way, the system-to-be is built by assembling pre-built units. The introduction of COTS helps to speed-up the process of software development because code can be reused. That is why this concept has been widely used at the implementation level, describing a component as a package of code (Souza & Wills, 1999). However, while defining the Software Architecture, its abstraction level is higher facilitating its reuse and, over all, the comprehension of the architecture.

Connectors

`Connectors` are defined in terms of the interaction among components. They are in charge of coordinating the process of the components they connect. This means they facilitate the separation of two concerns: the main computation performed by the components and the coordination provided by the connectors. For this reason, they supply the components with a loose coupling and enhance their reuse in different systems. (Shaw, 1994) describe them as:

> *"…the locus of relations among components. They mediate interactions but are not "things" to be hooked up (they are, rather, the hookers-up). Each connector has a protocol specification that defines its properties. These properties include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction such as ordering, performance, etc."*

In a similar way to components, they interact with the other parts of the system by means of the interfaces that describe the services they require and/or provide.

Some ADLs do not incorporate this concept explicitly. For instance, *Rapide* describes connection among components that cannot be named and, thus, they cannot be reused. However, several works, as (Allen & Garlan, 1994) and (Shaw, 1996), state compelling arguments for its definition as "first class citizens". Among them, it is specially relevant the expressive power and analysis properties that connectors provide to the architectural description.

Systems

It is frequently the case that different abstraction level must be provided to facilitate the understandability and the specification of the architectural description. For this reason, mechanisms to describe architectural elements with different granularity level are always desirable. Most ADLs have introduced the notion of `System` as a complex component, i.e., a component that is made up of other architectural elements. This facilitates that the system-to-be can be described in a hierarchical way, as (Andrade & Fiadeiro, 2003) set out. In such a way, the software composition can be defined in a straightforward way facilitating the reuse and the modularity.

Ports

Every architectural element usually has `interfaces`. An interface specifies the service, or set of services, that they provide and/or require. Interfaces are often typing the `ports` of the architectural element. They are the interaction points between them and the rest of the Software Architecture. They are in charge of preserving the black-box view every architectural element should have.

Connections

They are used to constrain when an interaction is allowed between which architectural elements. It is because they establish the communication channel between the architectural elements, connecting components ports and connector ports. However, if connectors are not considered as first-citizens in the specification then connections are only established between components. Usually, these connections are called `attachments`.

Compositions

This type of relationship is established to allow the communication between systems and the architectural elements that made them up. In this way, architectural elements with different granularity level are connected, providing a compositional semantics, as (Garlan, 2001) set out. This is why attachments are not used for this aim. These relationships usually receive the name of `Binding`.

Configuration

Additionally, the interconnection between these elements has to be embodied to describe the structure of the system. This structure is usually named `configuration`. Its definition is key to determine how the system-to-be will be built. Some assistance in order to define this topology, and select the involved elements, is provided by the `Architectural Style`. (Garlan & Shaw, 1993) describe them as:

> *"An Architectural Style defines a family of systems in terms of a pattern of structural organization. More specifically, an Architectural Style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints, having to do with execution semantics, might also be part of the style definition."*

As can be observed, they mainly sketch the main structure that every Software Architecture, compliant with an Architectural Style, should have. The Architectural Styles basically describe a set of constraints that must be satisfied if it is applied. Both (Buschmann et al., 1996) and (Shaw & Garlan, 1996) have proposed a set of styles to be reused for facing different problems that share a common solution space, as for instance the pipe and filters, event based, etc. More details about Architectural Styles are presented in chapter 7.

Other concepts

There are other concepts related to the Software Architecture that can be defined. For instance, the concept of `View`, firstly introduced by (Perry & Wolf, 1992), offers the analyst the facility of analyzing a Software Architecture from different points of view. It also relevant the concepts of `Property` (Garlan, 2001) and `Constraint` (Andrade & Fiadeiro, 2003) used to describe the semantics associated to the architectural elements or the restriction of the design, respectively. However, they have not been exploited in this work. Reader is referred to that works to obtain more details about them.

2.4 Aspect-Oriented Software Development

(Parnas, 1972) introduced the concept of `Separation of Concerns (SoC)` as a way to manage the complexity of software development by decomposing it into simpler units. Applying this concept, a loose coupling is achieved so that change on a concern has not any or reduced effect on the other concerns. Parnas demonstrated that it means a meaningful advantage in terms of maintenance and reuse. (Sutton & Tarr, 2002) have also highlighted its support for analysis and understanding, evolution and reuse. Most of the proposals following this approach consider that a concern is not an artifact produced during the software development but a conceptual entity that can affect throughout several artifacts. (Sutton & Tarr, 2002) describe a concern as:

"…any matter of interest in a software system"

As (Hursch & Lopes, 1995) state, several approaches have emerged that offer different support to this concept. For instance, `Component-Based Software Development` (CBSD) (Szyperski, 1998) promotes the separation of the functionality from the coordination concern by exploiting an assembly process to produce a final product. However, the CBSD approach is not a silver bullet for the software development. As (Jacobson, 2003) set out, components are developed to satisfy several requirements, which lead to a development where concerns of the systems are `tangled` and `scattered` across the components of the system. (Tarr et al., 1999) have described these concepts as:

"scattering - a single requirement affects multiple design and code modules - and tangling - material pertaining to multiple requirements is interleaved within a single module".

The `Aspect-Oriented Programming` (AOP), (Kiczales et al., 1997), (Elrad et al., 2001b), emerged as another approach for realizing this concept, but showing a key difference: it focuses on those concerns that crosscut a software system facilitating that each concern can be separately specified. For this reason, its notion of concern is close to that provided by the standard (IEEE, 2000):

> *"…those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders"*

The problem that AOP tries to solve is how to manage properly those catalogued as `crosscutting-concerns`, i.e., concerns that are scattered and tangled. The software development without considering separation of concerns techniques can leads to problems compromising its maintainability and performance. For this reason, AOP introduce mechanisms to factorize these crosscutting-concerns into units called `aspects` that can be reused throughout the system by weaving them wherever it is necessary, managing properly the tangled and scattered code. It means advantages in terms of understandability of the code, maintainability and reusability, as (Kiczales et al., 1997) demonstrated. AOP introduce a set of concepts necessary to understand this paradigm[2]:

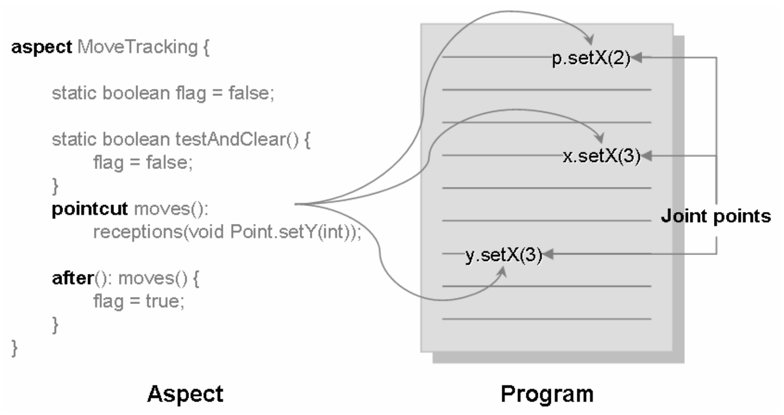
– `Base code` is that code that describes the core functionality of a program or domain, where the aspects are woven. Every aspect encapsulates a crosscutting concern for the specific program or domain. The `aspect code` collects the set of defined aspects.

– `Join points` determine the coordination structure between the base code and the aspect code. With this aim, each *joint point* identifies a point in the base code that determines where an aspect will be hooked.

– `Pointcut` is a set of join points. When the execution reaches one of them, the advice (a piece of code) is executed to determine the sequence of execution at that point. Usually, an `advice` determines if the aspect must be executed before, after or instead of the base code.

[2] (Dounce & Le Botlan, 2005) offer a more exhaustive definition of these concepts

– `Weaving` is the process allocated to control this change of context between the base code and the aspectual code, that is, when and how the injection of the aspectual code is performed.



```
aspect MoveTracking {

    static boolean flag = false;

    static boolean testAndClear() {
        flag = false;
    }
    pointcut moves():
        receptions(void Point.setY(int));

    after(): moves() {
        flag = true;
    }
}
```

Figure 2-7 Main concepts in AOP (code extracted from (Kickzales et al., 2001))

Figure 2-7 shows where each one of these concepts is established. The code has been (partially) extracted from (Kickzales et al., 2001) using as reference language AspectJ, the most widely accepted Aspect-Oriented language defined as an extension to Java[3]. It can be observed, that the specified *pointcut*, whose name is "moves", establishes a set of reference points, that is, each time the method "setY" of the class "Point" is referred, directly or indirectly, a joint point is established. It can be observed that there is not a specific notation for the join points but they are implicitly defined. Their semantics depends on the `Model of Joint Points` established in the Aspect-Oriented language. In the example of the figure, a model based on method call has been used. Several models of joint points have been established up to now following this approach. For instance, (Kickzales et al., 2001) establish a set of eleven kinds of *join points* that determines they can be established on a method call, method call reception, method execution, etc. Alternatively, other *Models of Join Points*

[3] See (Brichau & Haupt, 2005) for an extensive survey of the current Aspect-Oriented language and Execution Models

propose to specify explicitly such join points, usually by means of labels (Walker et al., 2003).

Figure 2-7 also illustrates how an advice has been established determining that *after* the execution of the *pointcut* "moves", the variable "flag" is set to "true". Therefore, it is responsible for providing the crosscutting behaviour. However, it must be taken into account that there are other alternatives to support this behaviour as, for instance, by means of `inter-type declarations`. Some languages, such as XAspects proposed by (Shonle et al., 2003), use them to modify the static structure of a program introducing methods, constructors, etc.

However, this paradigm has not only focus on the implementation level but a wide range of proposals has emerged that promote the detection and description of aspects at early stages of development, such as the design (Suzuki & Yamamoto, 1999), architecture (Pérez, 2006) or requirements (Rashid et al., 2002), in order to satisfy the closure property (Elrad et al., 2001b). This is how the term `Aspect-Oriented Software Development` (AOSD) has emerged exploiting the advantages this paradigm can provide in each stage of the software development. Because of the main motivation of this work is the establishment of a process that support the SoC from the requirements to the Software Architecture definition, in sections 2.4.1 and 2.4.2 an introduction to how the AO approach has been incorporated in both fields is presented. Special attention is devoted to the proposals in the Requirements Engineering arena because its relevance for the later definition of the architecture, as (Ferrari & Madhavji, 2007) state.

2.4.1 Aspect-Oriented Requirements Engineering

The attention to AOSD has burst onto the Requirements Engineering with the definition of `Aspect-Oriented Requirements Engineering` (AORE). This approach has been used with several objectives in the RE process. One of them is obviously the detection of `early-aspects`, that is, concerns detected in the requirements stage that are candidates to be realized as aspects in later stages of the development (Rashid et al., 2002). However, other alternatives try to exploit this approach to improve the requirements specification, such as (Alencar et al., 2006). This is because, no care the requirements specification employed, when dealing with complex and/or large systems, the crosscutting usually appears in the specification. This crosscutting manifests itself by affecting negatively the readiness and maintainability of the specification. AORE identifies and manages this crosscutting in an elegant and effective way, based on the SoC.

AORE does not have its own notation or expressiveness. Current proposals have developed their own notations based on other techniques in order to accomplish the identification of early aspects. Nonetheless, we can enumerate a some concepts that are common to all of them:

– `Concern` refers to the interests of the system, which can be either functional or non-functional.

– `Crosscuting` is a relationship among concerns that arises whenever a given concern interacts with other ones (either by constraining, extending, etc.). A more detailed description of potential crosscutting relationships can be found in (Rashid et al., 2003).

Some requirements models that are based on the Aspect-Oriented approach include the concept of *early aspect* as a constructor of the model. However, this is not mandatory because, when applying AORE, it is more important to detect and manage the composition relationships between early aspects (along with their traceability later) than to explicitly specify them as early aspects. The main purpose of AORE is to improve the crosscutting management and to establish the composition relationships between specifications. This is done by encouraging the separation of concerns in a similar way to the role played by weaving in Aspect-Oriented Programming.

It is worthy of note that the proposals presented up to date can be classified as `symmetric` or `asymmetric models`. Most of the current proposals are asymmetric because of the great influence AspectJ has had. It was the first proposal, and currently, the most widely used. This kind of models assumes that there is a dominant decomposition, which is usually the functional one. Aspects specify the crosscutting concerns, typically the non-functional ones, of the system-to-be for thei later woven to the functional ones. However, the symmetric model do not care whether they are crosscutting concerns or not. Each concern is independently specified and can be woven or not depending on the expected behaviour of the system.

In the following sections, an example of different proposals in the field, according to the RE notation extended, is presented (a deeper analysis can be found in (Chitchyan et al., 2005)).

Scenario Approach and AORE

Several proposals have been presented that employ scenarios as the base approach for AORE. Most of them employ the Use Cases Models as the base notation to describe the functional requirements, and another notation to identify the non-functional requirements and its possible crosscutting relationships.

(Brito & Moreira, 2003) have presented one of these proposals. They have defined an extension to Use Case Model to identify and specify the crosscutting concerns, considering quality attributes as the first candidates to be crosscutting-concerns. They define the functional requirements by means of a Use Case Model. The non-functional ones are described by using a template describing their *names*, *description*, etc. It is specially relevant the section *where* and *contribution* to detect the candidate aspects. It is because they list the models, model elements and concerns that are affected by the non-functional requirement being specified. Both models are composed by describing the matchpoints in the Use Case diagrams. The composition stereotypes used are *overlap*, *override*, and *wrap* depending on the candidate aspect is applied before (or after), or superposing or encapsulating the concerns it transverse, respectively. One of the main advantages exhibited by this approach is that it allows the modelling of aspects (elicited at the requirements phase) at the design phase. It is also relevant the identification of some mechanisms to facilitate the trade-off among conflicting candidate aspects. However, the use of different models to describe functional and non-functional requirements can lead to problems in terms of the maintainability and traceability between them. In addition, they neither provide a tool to support their proposal.

Goal-Oriented and AORE

The goal-oriented approach is by definition quite appropriate to be used for detection and specification of crosscutting concerns. It is due to the fact that it introduces a notation to specify contributions between goals described in the model. However, not many works have been presented up to date. (Yu et al., 2004) is one of these proposals. They have defined an analysis process that helps to identify aspects by using a Goal Model that they have called a *V-Graph*. This process is mainly based on the relations of functional goals (called *goals* in their proposal) and non-functional goals (called *softgoals*). These goals are refined into sub-goals until operationalizations can be described. These operationalizations have a contribution relationship towards the satisfaction of goals. An aspect is detected whenever an operationalization is contributing to several goals. One of the advantages of this proposal is that it offers an automatic support to detect early aspects. In addition, they propose a well-defined process to specify the V-Graph by means of refinement. However, its main drawback emerges when dealing with negative contributions. It is because some goals, necessary to achieve the satisfaction of the model, could be not satisified because of the negative contributions they have. It means that these requirements could not be properly mapped to the operationalizations level and they would not meet by the system-to-be.

Viewpoints and AORE

Some proposals have exploited the Viewpoint approach for the detection of early aspects. The main advantage they show is its capability for managing conflicts, so relevant when different stakeholders are involved in the process.

(Rashid et al., 2003) have defined the *Early AORE model*. They describe a process that systematically entails all the activities of SoC: identification and specification of concerns, detection of candidate aspects, composition of concerns and handling of conflicts. Different Viewpoints are used to identify the concerns, and their related requirements, of the system-to-be. Both elements are specified in an association matrix that facilitates the identification of the candidate aspects whenever a concern is related to several Viewpoints. They have also defined a set of composition rules in order to facilitate their later composition.

In addition, they provide support for trade-off when conflicts emerge. These conflicts are defined in an association matrix, describing when a concern contributes negatively to other/s along with a weight that indicates its importance. A negotiation process is carried out to resolve such situations. Once this process is resolved, the elicited aspects are classified according to their importance and mapped on the artifacts defined at later stages of development.

This proposal exhibits several advantages. One of them is that a tool called Aspectual Requirement Composition and Decision support tool (ARCADE) supports the whole process. In addition, it is the proposal with a more detailed description of the composition rules to be applied in the early stages of development. However, authors recognize that the composition rules have not validated beyond the proposed case study. They recognize that these rules could be highly dependent on the application domain.

Features and AORE

The exploitation of Features models is gaining more and more adepts because of its simplicity. (Pang & Blair, 2004) have exploited it by extending the Agile process of *Feature Driven Development* (FDD) proposed by (Coad et al., 1999). The FDD is analysed by using a feature extraction template in order to identify the crosscutting. The analysis determines that a crosscutting exists if more than one class is included in a feature. They also describe a specific process for managing conflicts. This crosscutting is identified and resolved by an algorithm that is based on *Boundary Condition Exploration* that uses the priority assigned to the features. They have defined a specific notation, similar to AspectJ, that is in charge of providing the crosscutting behaviour. However, it is only a theoretical proposal that has not been validated in a case study nor has a supporting tool.

Multi-dimensional AORE

(Tarr et al., 1999) proposed the Multidimensional Separation of Concerns to break the called "*tyranny of the dominant decomposition*". The main idea is that artifacts are defined, by default, by multiple and overlapping concerns. For this reason, if they were decomposed according to several concerns simultaneously then advantages in terms of traceability and impact of change would be obtained.

Several proposal have emerged that describe the main ideas provided by this approach, such as (Sutton & Rouvellou, 2004), and (Moreira et al., 2005). The latter have proposed an extension to the (Rashid et al., 2003)'s work called CORE (Concern Oriented Requirements Engineering). However, in this case they decompose the requirements uniformly without taking into account if they are functional or not. If facilitates that any concern can be mapped to any other one, without caring its nature and providing an enhanced flexibility.

CORE proposes to use a *meta concern space* that can be reused from system to system. It is defined as a catalogue of concerns that is used to classify the requirements in the specific system. In order to reduce the potential number of concerns to be analysed as possible conflicting ones, they have introduced the notion of *compositional intersection*. It provides a reduced set of concerns that can be used as a base for concern projection and the trade-off analysis that is performed latter on. CORE identifies a matrix to determine the relations between concerns that is the base to apply the composition rules at the requirements level. A conflict resolution is applied by assigning priorities and its later discussion with the stakeholders. For every concern, its influence on decisions at the architectural level is identified. The proposal exhibits several advantages related to the way the decomposition of the system proceeds and the support it provides for conflict resolution. However, one of the main drawbacks is that there is not a catalogue of concerns to help in the process, neither any guidance to determine the compositional intersection. In addition, there is not a supporting tool facilitating its applicability.

Other approaches to AORE

There are some proposals that have been defined without following any traditional approach to RE. Among them, it is worth noting that proposed by (Baniassad & Clarke, 2004). The proposal is based on a concept they have introduced: *Theme*. It is a meaningful unit of cohesive functionality that is composed with other themes. This facilitates that the system-to-be description follows a multidimensional approach. There are two kinds of themes: *base themes*, which may share some structure and behaviour with other base themes; and, *crosscutting themes* that have behaviour overlapping the behaviour described by the base themes. The crosscutting themes are identified as *aspects*.

The main idea of the proposal is to describe the different *Themes* of the system-to-be by means of a graph-based representation, which is amenable to perform the subsequent analysis to determine the crosscutting. With this aim, four elements are used throughout the process:

– *Action View* that describes non-hierarchical links between the *action words* (verbs) identified from the requirements document and requirements sentences. These actions must be previously classified as *Themes* if they are major enough or just behaviour within themes following a described process. If a requirement sentence is linked to several themes, and it can not be rewritten to break such link 1-N, then tangling behaviour is identified, and, thus, an aspect is identified.

– *Clipped Action View* that is generated from the previous View by clipping the secondary actions from every requirements sentence. This secondary behaviour will crosscut the base behaviour and the primary actions will be classified as *base*.

– *Theme View* is in charge of identifying entities from the requirements document.

This proposal provides the advantage of helping to identify aspects by using action words. In addition, there is a tool called Theme/Doc that helps to automate to some extent the process. Despite their well-behaviour with functional requirements, it does not seem so appropriate with the non-functional ones. It is because they are not usually written using action words, and, thus, it is required to rewrite them for their identification. That means that finally the analyst must do the work that is the advantage of the proposal.

Main discussion

As can be observed, there are a wide diversity of proposals that provide support for the identification, specification and composition of concerns at the early stages of development. They are not only theoretic proposal but most of them have been applied and have a supporting tool. However, the major problem is related to traceability throughout the software development process. As far as we know and according to (Baniassad et al., 2006), there is none proposal that provides traceability from requirements to architecture maintaining the so desired SoC, that is, a proposal where AORE and Aspect-Oriented Software Architecture (AOSA) are combined.

2.4.2 Aspect Oriented Software Architectures

Recently, a deal of work has been performed that exploits the benefits of the integration of AOSD and Software Architecture. This integration must tackle a wide diversity of issues but two of them are mandatory for any proposal. The first one is related to how the concept of aspect is introduced at the architectural level. However, their definition at the architectural level is not standardized but a wide set of definitions have emerged. The second one is related to the composition, i.e., the description of mechanisms that integrate both architectural aspects and architectural elements in a suitable way. Both issues are more detailed in the following.

The incorporation of aspects at the architectural level implies considering what an aspect is at this level. However, as was stated above, there is not a standard definition of aspect at the architectural level but its meaning is very dependent of the proposal[4]. In order to facilitate its comprehension, (Cuesta et al., 2005) have elaborated a taxonomy that can be used to understand the different meanings this concept can have at the architectural level. They have identified the following approaches:

a) *Non-aspects*. The proposals in this category, such as JIAZZI (McDirmid et al., 2001), consider that aspects do not have to be defined because either they are not necessary or they can be provided by some existing composition mechanism.

[4] (Pérez, 2006) and (Chitchyan et al., 2005) offers a deeper analysis of the most well know proposals in the field

b) *Architectural aspects.* The proposals following this approach extend or use the architectural elements to specify architectural aspects, that is, they are considered as a special type of component and/or connector that support aspectual capabilities. Some proposals in this category are FUSEJ (Suvée et al., 2005), AspectLeda (Navasa et al., 2005), etc.

c) *Aspectual binding.* The proposals in this category state that there is no need of describing aspects but only extending the binding mechanisms of the components to consider the aspectual ones. They use some kind of architectural abstraction to encapsulate the aspectual interaction. Aspectual components (Lieberherr et al., 1999), CAESAR (Mezini & Ostermann, 2003), or Composition filters (Bergmans & Aksit, 2001) are examples of this kind.

d) *Concern model.* This approach considers those ADL that provide support for the description of an internal model of concerns. Perspectival Concern-Space (Kande, 2003) and PRISMA (Pérez, 2006) are clear examples of this kind.

e) *Multiple dimensions.* It is similar to the previous one but concerns are made explicit in their definition. The description of the system is structured according to the identified concerns, and later on, some mechanism is provide to facilitate their composition. Architectural Views of Aspects (Katara & Katz, 2003) is a proposal following this approach.

Taking into account the taxonomy described above, it is described in the following sections how the composition with the architectural elements has been defined by different proposals.

Non-aspects

One of the proposals following this approach is *Jiazzi*. It is an aspect-oriented proposal that extends Java in a non-invasive way because it does not change the core of the language. The construction of systems is performed by defining java classes, which constitutes the base code, and components, which are called *units*. These *units* can be thought as generalized Java packages that are described by importing *packages* of java classes. Figure 2-8 illustrates the unit "a:applet" which imports the package "ui" and exports the package "applet". *Units* are compiled and typed-checked independently of the base code improving the composition of concerns later. Each *unit* can import and export several classes so that it can modularize a given concern that crosscut multiple classes.

A separate language of Jiazzi, which acts as an aspect-configuration language, is in charge of composing base code and units. The links are defined externally eliminating hard-coded dependencies and making more flexible the definition of the components. The linking creates the *compounds* that are built from other units and compounds. As can be observed in Figure 2-8, they establish the connections between the *units* by matching source and destination *packages* in each *unit*, so that direct connection between classes can be eliminated. The compound can be used to create new compounds because they can export packages as it is shown in the Figure 2-8 by "ui_out" and "applet_out". *Compounds*, *units* and java classes of the base code are linked all together by the *jiazzi unit linker* and compiled to produce new builts.

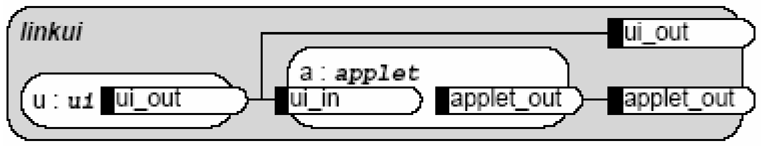


Figure 2-8 Describing the *compound* linkui as a link between the *units* applet and ui by matching the packages ui_out and ui_in (extracted from (McDirmid et al., 2001))

As can be observed, this proposal does not introduce the concept of aspect in its definition but a *unit* whose granularity regarding other proposals is higher. In addition, it has been defined in a non-invasive way for the Java language. It cannot deal with concerns whose implementation is deeply tangled with other code so that it only provides an aspect-interaction similar to the "around" advice defined previously (see section 2.4).

Architectural Aspects

Most of the proposals following the *Architectural aspects* approach employ or extends the mechanisms provided to connect architectural elements. One of these proposals, *FUSEJ*, uses the mechanism of composition described in Figure 2-9. *Components* are used to describe both regular components and aspect-oriented components in the *Component Layer*. Each *Component* describes a set of services, which it provides, called *features*.

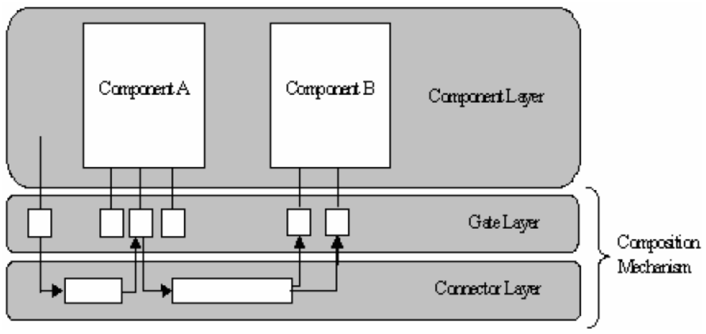


Figure 2-9 Describing the composition in FUSEJ (extracted from (Suvée et al., 2005))

These features cannot be directly accessed. It is by means of the *gates*, described in the *Gate Layer*, how it is specified the *features* that are provided for a *Component*. The coordination between the *Components* is established specifying *connectors* in the *Connector Layer*. They are specified in a different way depending on whether they are coordinating two regular components or a regular component and an aspect-oriented one. In the latter case, the connector is specified using the primitives of the Aspect-Oriented approach. It can be observed, that the specification of the *connector* is specialized to deal with the interaction mechanisms of the Aspect-Oriented approach.

Aspectual binding

Regarding the *Aspectual binding* approach, (Lieberherr et al., 1999)'s proposal considers an *aspectual component* as a module of functionality that is structured by means of a graph called *Participant Graph*. Their participants are object-oriented classes. Each *aspectual component* must describe its expected and provided interfaces achieving their higher reusability. These aspectual components are composed with the base application that is just a special kind of aspectual component but without interfaces. Both the base application and the *aspectual components* are composed by means of *connectors* that coordinate them by encapsulating the pointcuts.

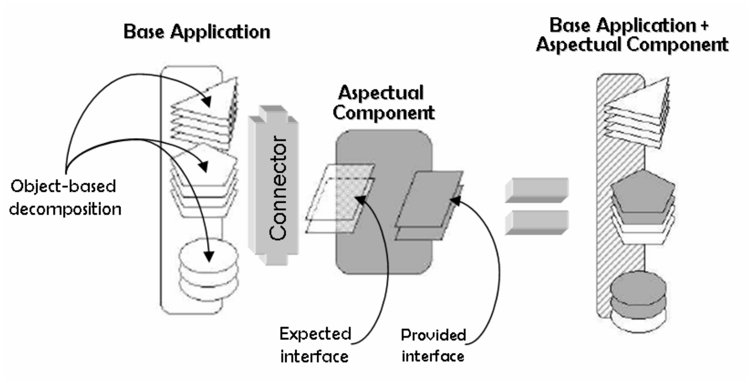


Figure 2-10 Composing an Aspectual Component and the Base Application (partially extracted from (Lieberherr et al., 1999))

The *connectors* are in charge of performing the pattern matching between the base application and the interfaces of the *aspectual components*. Once this composition is established, the result is weaved code at the deployment stage as is described in the Figure 2-10. It is worthy of note that this *connector* can also be used to describe the composition between *aspectual components* in order to produce a new composed *aspectual component*.

Concern model

As was stated above, the *PRISMA* model is an example of *Concern model*. PRISMA defines the architectural elements, both components and connectors, by means of a gluing of aspects, where the weaving relationships are described internally to these elements achieving more reusable aspects, as is described in Figure 2-11. This means that there is not any mechanism to establish the composition between the architectural elements and the aspects, but the composition of aspects are described internally to the architectural elements. This model is more widely described in section 4.3.

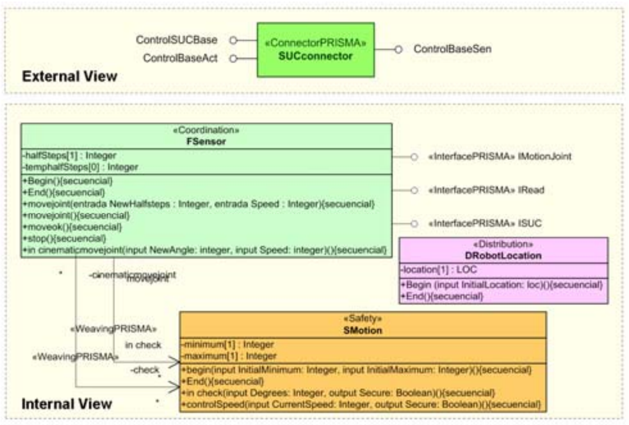

ControlSUCBase
ControlBaseAct
«ConnectorPRISMA» SUCconnector
ControlBaseSen
External View
«Coordination» FSensor
-halfSteps[1] : Integer
-temphalfSteps[0] : Integer
+Begin(){secuencial}
+End(){secuencial}
+movejoint(entrada NewHalfsteps : Integer, entrada Speed : Integer){secuencial}
+movejoint(){secuencial}
+moveok(){secuencial}
+stop(){secuencial}
+in cinematicmovejoint(input NewAngle: integer, input Speed: integer)(){secuencial}
«InterfacePRISMA» IMotionJoint
«InterfacePRISMA» IRead
«InterfacePRISMA» ISUC
«Distribution» DRobotLocation
-location[1] : LOC
+Begin (input InitialLocation: loc)(){secuencial}
+End(){secuencial}
-cinematicmovejoint
movejoint
«WeavingPRISMA»
in check
«WeavingPRISMA»
-check
«Safety» SMotion
-minimum[1] : Integer
-maximum[1] : Integer
+begin(input InitialMinimum: Integer, input InitialMaximum: Integer)(){secuencial}
+End(){secuencial}
+in check(input Degrees: Integer, output Secure: Boolean)(){secuencial}
+controlSpeed(input CurrentSpeed: Integer, output Secure: Boolean)(){secuencial}
Internal View

Figure 2-11 Describing a PRISMA connector

Multiple dimensions

Finally, the *Architectural Views of Aspects* is a proposal following the *Multiple dimensions* approach. This proposal uses *Views* as a means to specify and analyse the system-to-be from the point of view of a specific concern, such as security, or functionality. Each *View* can be treated by more than one *aspect,* where an *aspect* is a module that encapsulates a set of components along with the connections between them. As can be observed in the Figure 2-12, an aspect can be shared by several concerns to describe a behaviour that is addressed by them. In addition, they can be composed when it is necessary to form a composite aspect for a single concern. For instance, "(C, S)" are composed for the "Security" concern in Figure 2-12.

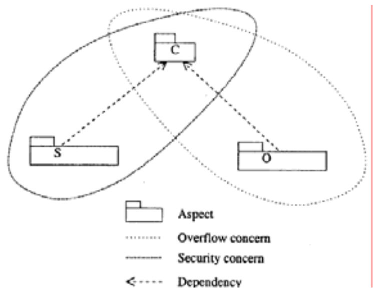


Figure 2-12 Concern diagram (extracted from (Katara & Katz, 2003))

The composition of aspects is based on the *superimposition principle*, an asymmetric operation that establishes the composition order of the aspects to establish which aspects are applied on top (before, after or instead) of others.

The *dependencies* between modules are described when given a module it has elements that are bound to elements belonging to other module. These relationships are used to perform the composition. Therefore, in the example of the Figure 2-12, a specification that deals with both "overflow" and "security" concerns would compose the aspect in any order, that is, "S/O/C" or "O/S/C".

Some conclusions

Taking into account the above presented proposals, it can be concluded that the definition of aspect at the architectural level is highly dependent on the proposal. Most of them consider that `Architectural Aspects` provide a new means of modularization and encapsulation part of the system computation and/or its interaction that is used in the definition of the system-to-be. In addition, it can be notice that some of them use symmetric model where the crosscutting do not have to exist to encapsulate that behaviour but it is intended as a way of managing the complexity of the specification.

As was expected, the mechanisms used for the composition between the architectural aspects and the architectural elements depend totally on how they have incorporated in the proposal. It was observed that some proposals do not deal with such composition because they are not directly connected.

2.5 Conclusions

In this chapter, a brief overview about the Requirements and Software Architecture fields has been presented. The most well known proposals in RE has been introduced in order to make know the reader the concepts they introduce. Similarly, the most important concepts in SA have been described as well. It will facilitate the comprehension of the remaining chapters.

In addition, the Aspect-Oriented Software Development approach has been introduced. This approach means a step forward the achievement of maintainability and reusability, two quality factors key in every software development process. Its use facilitates an efficient management of change because their impact can be better understood and evaluated. It facilitates the analysis of the system because it reduces the complexity of its specification. How this approach has been addressed at the Requirements and Software Architecture stages has been introduced as well.

"There was a wish to get something exceptional…
I also wanted to deliver something technically
unique" —
Santiago Calatrava

CHAPTER 3

Intertwining Requirements and Software Architecture: a Context for ATRIUM

3.1 INTRODUCTION

Nowadays, software has quality as a goal throughout its lifecycle, from its inception to its completion. Several factors, methods and/or processes can be used to cope with this issue. Some of them assessed the quality in terms of customer satisfaction. Such approach highlights how needed the elaboration of high quality requirements specifications is. Therefore, they should produce systems that are more likely to perform according to the stakeholder's expectation.

Architectural specifications allow the analyst to reason whether a system satisfies its requirements and, therefore, to determine the quality shown by the system. However, it is clear that the transition from requirements to architecture is not a straightforward task but a complex one, because different languages are used when dealing with both kinds of artifacts. In addition, once a system is built, new and changed requirements may arise and the system needs to evolve. Therefore, traceability among them becomes a critical issue for the development. For this reason, some workshops and conferences, as for instance STRAW'01 (Castro & Kramer, 2001) and STRAW'03 (Berry et al., 2003), as emerged lately to cope with this issue.

An additional topic has also emerged: AOSD. As was presented in the previous chapter, there is a need of providing traceability from the early-aspects detected and specified in the early stages of development to the architecture and, finally, to the code. This also constitutes another topic that must be dealt with by any proposal following this approach.

In order to describe properly these issues the chapter is structured as follows. Section 3.2 describes the most relevant works that deal with the intertwining between Requirements and Software Architecture along with a comparative framework. Section 3.3 presents the process that has been defined in this work having into account the previous proposals. The chapter finishes with the obtained conclusions.

3.2 PROPOSALS INTERTWINING REQUIREMENTS AND SA

In the next sections, some of the most relevant approaches in the field are introduced. It can be observed that they are catalogued according to the RE approach they use. This is because it has been detected, as (Ferrari & Madhavji, 2007) set out, that a proper knowledge in RE, from the point of view of techniques and their exploitation, has a high impact on the quality of the architecture specification.

3.2.1 Goal-Oriented for defining SA

By putting emphasis on goal analysis, goal-oriented proposals explicitly link business needs and objectives to system functional or non-functional components. According to (Kavakli & Loucopoulos, 2005) there are four proposals that concern on this topic: KAOS (Lamsweerde, 2003), GBRAM (Antón, 1996), the NFR framework (Chung et al., 2000), and the CREWS-L'Ecritoire (Rolland et al., 1999). However, we can focus on *i** (Castro et al., 2002) due to the fact that this has been an extension of the initial NFR Framework in order to deal with functional requirements too. These works are based on the premise that systems components satisfy some higher goal in the larger environment. In addition, GRL, proposed by (Liu & Yu, 2004) is also introduced because it has also commonalities with this thesis.

KAOS

(Lamsweerde, 2003) has proposed the use of the KAOS framework (Dardenne et al., 1993) to guide the process of elaborating the Software Architecture from requirements. It defines an iterative refinement process, from functional specifications to an abstract architectural, draft in order to meet domain-specific architectural constraints. The KAOS methodology is aimed at supporting the process of requirements elaboration – from the high level goals that should be achieved by the composite system to the operations, objects and constraints to be implemented by the software. KAOS use four models, depicted in Figure 3-1:

– *Goal Model*. The various *goals* the system should meet are defined in this model and interrelated by means of AND/OR refinement links. Whenever a goal is assignable to an *agent* of either the environment or the system is specified as a *constraint*.

– *Object Model*. A KAOS `Object` is a thing of interest in the domain whose instances may evolve from state to state. In KAOS, the object model describes the set of *entities* along with the possible *relationships* among them and the *events* that can arise through the objects life.

– *Operational Model* describes the set of KAOS Operations, i.e., the set of input-output relations over objects. Whenever an operation is applied, a state transition for the involved objects is performed.

– *Responsibility Model*. An `agent` is an object acting as a processor for some actions. Agents can be humans, devices, programs, etc. The Operational Model and the Responsibility Model are directly related because the former describes the services to be provided by the *agents* of the latter. It is because each *operation* is assigned to be *performed* by a specific *agent*. The Responsibility Model is also related to the Object Model because the agents are in charge of *monitoring* and *controlling* the objects state. Finally, this model is related to the Goal Model as well thanks to the *responsibility* relationships established between *constraints* and *agents*.

Software Agents, identified in the Responsibility Model, are the main elements used to generate the architecture. That means that every software agent responsible for performing some constraint will be specified as a component in the final architecture. The connections among these components are established thanks to the monitors and controls relationships that agents have. Every time an agent monitors an object that another agent controls, then a connection is established between them.

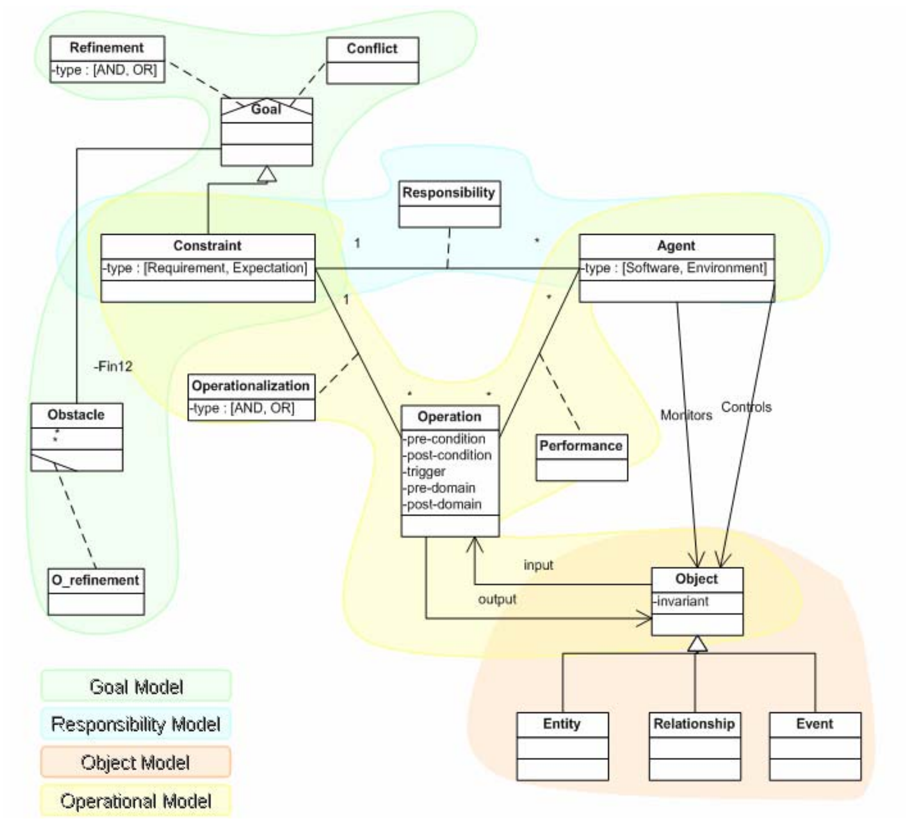


Figure 3-1 Four inter-related models for KAOS

One of the main advantages exhibited by this proposal is its capability to analyse both alternative designs and systems. With this aim, the proposal provides the following facilities:

i. definition of optional goals by means of the OR relationships. The introduction of optional goals means that different systems can be defined depending on which are finally selected;

ii. definition of optional operations. They describe alternative ways to operationalize a constraint, and, thus, different designs of the system because the responsibilities assigned to the agents will depends on the selected operations.

The main restriction shown by this approach is related to the use of non-functional requirements. They are only used to make decision among several alternatives but they are not as relevant as the functional ones. This is quite controversial because their relevance has been widely recognized, as (Bass et al., 2001) and (Bosch & Molin, 1999) set out.

We should mention that there is another proposal that exploits the KAOS approach to generate architectural specification that has been suggested by (Brandozzi & Perry, 2001). They promote the use of intermediate descriptions between requirements and architectures known as `Architectural Prescriptions`. They are a draft of the description of the architecture using an Architectural Prescription Language (APL). They have established mappings between the from KAOS *entities* and *relationships* to *data components*, and from KAOS *agents* to *processing components*. These mappings are established using the goals described in an intermediate level of the KAOS Goal Model. In successive steps, these architectural elements are refined, modified or deleted until every *component* has some constraint to satisfy, that is, some goals to achieve. The main advantage this proposal exhibits is that an architectural prescription is generated without being oriented to the implementations. However, it does not give a full guidance throughout the process. For instance, it does not describe how to detect which level is more appropriate to start the process, why to describe non-functional requirements as additional constraints when they are already described in the KAOS Goal Model, etc. In addition, it is especially relevant that it is not clear how the connections among the components are established.

GBRAM

(Antón, 1996) has focuses its efforts on the definition a well-established process for identifying, elaborating, refining and organizing Goal Models. With this aim, GBRAM extends the exiting approaches by means of the definition of heuristics, which help to identify goals, and guidelines and recurring questions, which help to refine them.

The main activities are depicted in Figure 3-2. It can be observed that *Explore* is in charge of studying the available information to obtain a proper knowledge of the needs of the system-to-be. This information is used by the activity *Identify* that analysis the documentation, by applying a set of defined questions, to extract goals and responsible agents. The identified goals are *organized* in the next activity by cataloguing them as achievement or maintenance goals. *Refine* activity is devoted to establish the precedence relationships, that is, when a goal must be fulfilled before another. A set of questions is also provided to help in this activity. *Elaborate* is the process of analyzing the goal set by identifying obstacles, which can prevent the completion of the goal, identifying scenarios, which help to uncover hidden goals, and identifying constraints, which must be met for the goal completion. Finally, the *Operationalization* activity focus on defining goals with enough detail by specifying the relationships between goals and agents in terms of events that cause a change of state. The outcome of GBRAM is a Software Requirements Document (SRD) that contains a complete description of what the system will do.

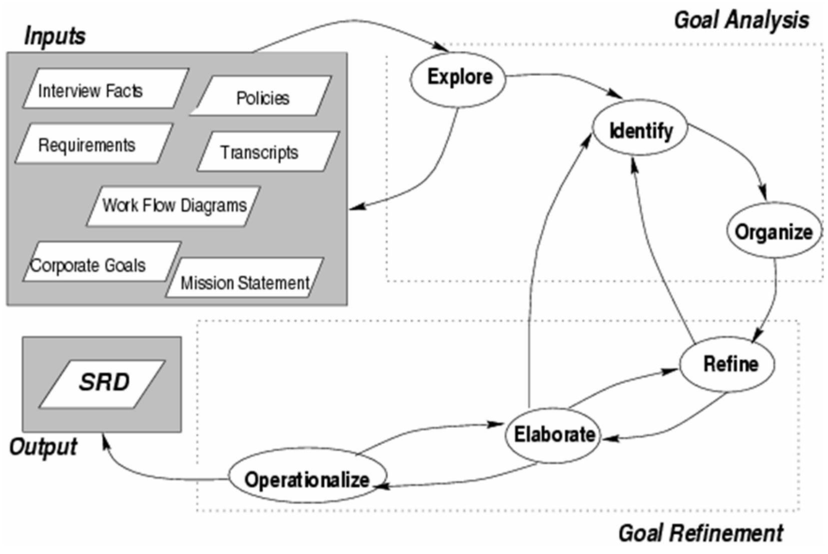


Figure 3-2 Main activities of GBRAM (extracted from (Antón, 1996))

As was stated, the main advantage of this proposal is its guidance in the process of Goal Model definition. However, this proposal does not deal with the description of a Software Architecture but an initial assignment of responsibilities to agents. In addition, this method produces a software specification of the functional requirements in the form of goal schemas without considering the non-functional requirements and the constraints they mean.

TROPOS

(Castro et al., 2002) have established this methodology for guiding the process of system specification from early requirements to a detailed analysis. This methodology includes a set of techniques to generate code executable in the platform JACK (an agent-oriented platform). The requirements are gathered and elicited using the framework called *i**. This framework has been specially used to analyse business goals. *i** includes two models:

– *Strategic Dependency Model* is a graph, where each node represents an actor, and each link between two actors indicates that one actor (*depender*) depends on another (*dependee*) for something (*dependum*) so that the former achieves some goal. It is defined by means of *goals, softgoal* (not precisely defined goals), and *agents*.

– *Strategic Rationale Model* is employed for reasoning how each actor expects to fulfil its dependencies. Using a means-ends analysis, it is determined how the goals can actually be fulfilled thanks to the contributions of other actors. Figure 3-3 illustrates an example of this Model. It can be observed that it has four kinds of nodes: *goals*, *tasks* (steps to accomplish a goal), *resources*, and *softgoals*. It illustrates an example of the goals of an e-business shop called "Mediashop" whose goals are marked by means of a boundary. It can be observed that their dependencies with other agents are established by means of *dependencies*.

Once both models have been defined, they are lately refined in an analysis stage where both functional and non-functional requirements emerge. In this phase, the number of requirements increases because of the decomposition of the system.

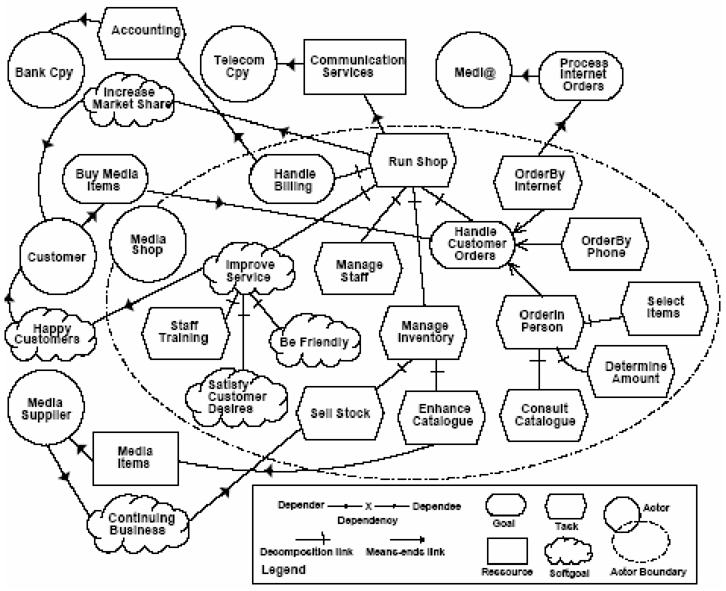


Figure 3-3 Strategic Dependency Model (extracted from (Castro et al., 2002))

Another Goal Model is defined in order to select the Architectural Style applicable to the system-to-be. The NFR Framework is used to perform an analysis of the Architectural Styles, represented as operationalizations, according to their contribution to the identified quality criteria. The selection of the Style implies a refinement of the Strategic Dependency Model by introducing new agents and re-assigning responsibilities. Once these agents have been identified, they are refined into a detailed analysis by using the extension of UML proposed by the FIPA (Foundation for Intelligent Agents, (Odell et al., 2000)).

One of the advantages exhibited by this proposal is that it is a driven-requirements proposal, that is, authors state that the same concepts are used throughout the process, avoiding misconceptions. In addition, they generate an executable which facilitates the validation tasks.

However, some problems emerge related to the analysis. In this proposal, the mean-ends links are in charge of determining the alternatives to meet the requirements. In a similar way to KAOS, these alternatives describe different *tasks* in the system (operations in KAOS) to achieve a specific goal or softgoal. Once these tasks have been assigned, a set of actors are identified as responsible to perform such tasks. This means that there is not analysis of alternatives in terms of the composing element of the system-to-be. There is not any help to guide the analyst in this process either. In addition, there is no mention about how the architecture is generated. We should point out that Software Architecture is not only described by the identification of its constituents but by the relations they have as well. This topic has not been addressed in the published works. Finally, the use of a network approach for representing the models, not a hierarchical one as KAOS uses, can offer problems of legibility to the model when the system is complex.

CREWS-L'Ecritoire

(Rolland et al., 1999) have defined a proposal where both goal-driven and scenario-based approaches are integrated. They are highly coupled allowing the analyst to move from goals to scenarios and vice versa. It is because when a goal is established a scenario is described for it, but also, once a scenario is specified it is analysed to discover new goals. This means that both processes are coupled, with an incremental discovering of goals and description of scenarios.

The model they have proposed has as building blocks what they have called *Requirement Chunks* (RC). It is a pair <G, Sc> where G is a goal and Sc is a scenario. For the description of goals, they use a structured natural language to facilitate their later analysis. A scenario is described as "*a possible behaviour limited to a set of purposeful interactions taking place among several agents*". It is described by means of a set of *agents*, which interact by means of *actions*, a *pre-condition* and a *post-condition.*

As can be observed in Figure 3-4, the *RCs* are related by means of AND/OR composition relationships. But, they have also introduced AND/OR refinement relationships to relate *RCs* described into different abstraction levels. They have defined three levels of abstraction, which are called *contextual*, *functional* and *physical*, depending on if they describe services for the organization, for the users, or their actual performance, respectively.

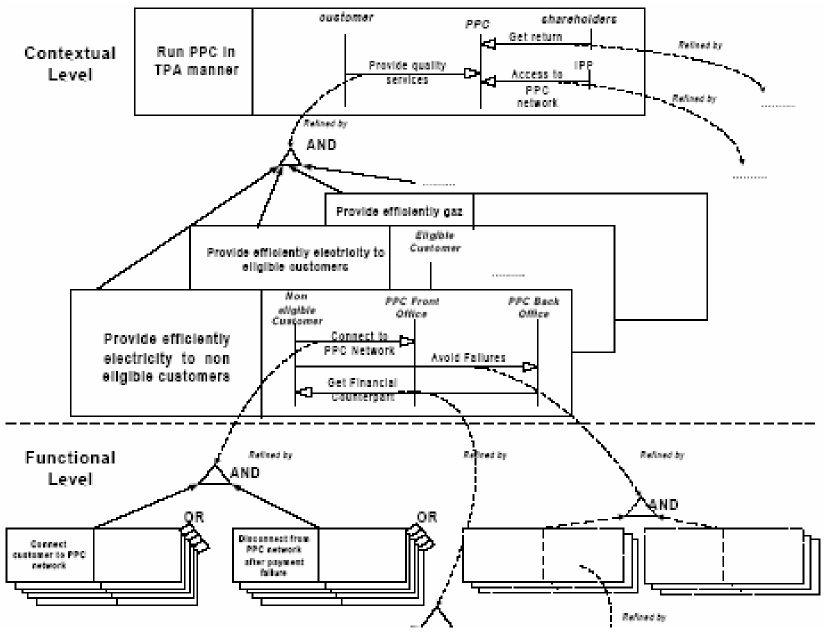


Figure 3-4 Requirements chunks at different levels of abstraction

This proposal exhibits several advantages, over all in terms of analysis. The exploitation of a textual template to describe the goals have been used to determine alternative designs. It is because every parameter they introduce is used to determine the most likely alternatives for the abstraction level where the RC is being described.

However, this proposal does not provide more guidance for the generation of the proto-architecture. It is mainly because scenarios are described by means textual descriptions. In addition, they do not provide specific identification of non-functional requirements, but they are tangled with the functional ones.

GRL

(Liu & Yu, 2004), similarly to l'Escritorie, propose to combine scenarios and goals-based models during architectural design. The `Use Case Maps`

(UCM) notation, proposed initially by (Buhr & Casselman, 1996), is used for the former and the *GRL* language is used to define the latter.

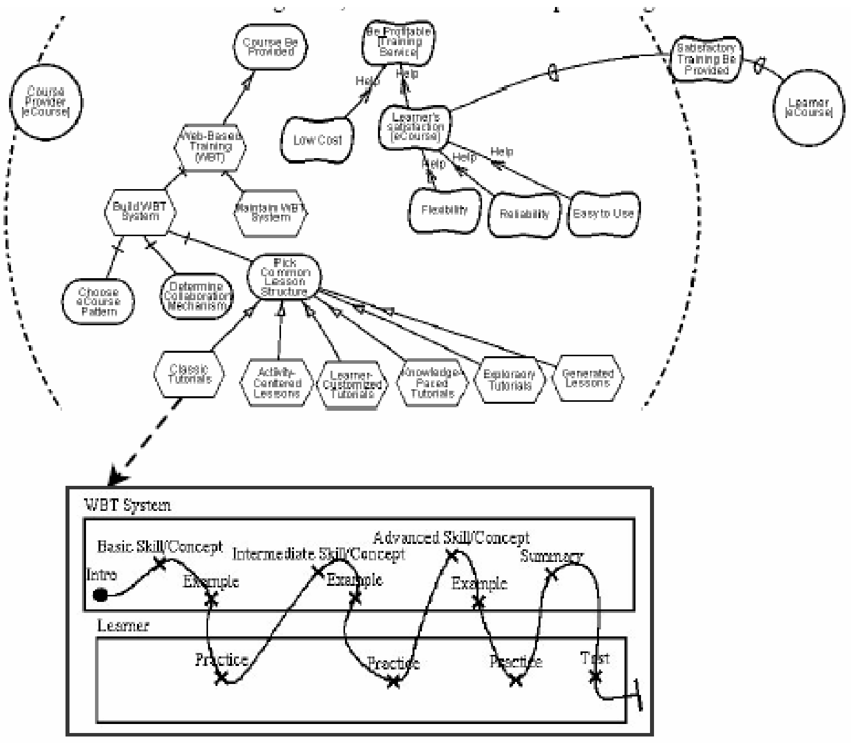


Figure 3-5 Alternative Designs and their UCM (extracted from (Liu & Yu, 2004))

By means of GRL, a goal model is established similar to that presented for *i**. The main difference is related to this proposal does describes an ordered refinement of the model, as can be observed on top of the Figure 3-5. In addition, the analysis process is performed on the Goal Model by means of tasks once the different alternatives have been established. This process determines if the functional goals are achieved by evaluating the tasks respect to the non-functional requirements in order to obtain the optimum alternative. The optimum task is lately refined to facilitate the process of scenario identification.

The UCM are scenarios employed to describe causal relations between the interacting elements. The elements used in their definition are: *starting point* (trigger) that is marked by means of a point; *responsibility* (actions, tasks or functions) is specified by means of a cross; *end point* (post-conditions) specified by means of a line; and *components* (entities or objects of the system). The execution is established as path going from the starting point to the end point through the responsibilities. A responsibility point represents a place where some component of the system is changed or enquired. The UCMs are described using the refined tasks as can be observed in Figure 3-5, where a trace is established from a task "classic tutorial" to a UCM for the WBT System.

One of the main advantages of this proposal is that UCM can represent system designs in a high-level way. However, the tradeoffs between alternatives and the intentional reasoning behind design decisions cannot be explicitly shown, because both models are decoupled.

3.2.2 Scenarios and AOSD

Some proposals have emerged during the last years to face the problem of defining Requirements and Software Architecture from the AOSD perspective using the scenario approach. Three of them are described in the following.

AOSD/UC

AOSD/UC has been proposed by (Jacobson, 2003) focusing on AORE. However, the main idea in its proposal is that Use Cases are, by definition, crosscutting concerns. This is because several classes are usually necessary to realize a Use Case, and the same class can collaborate to realize several Use Cases. This idea is illustrated in Figure 3-6. We can observe that "Interface" is realizing the three Use Cases established. Jacobson has proposed the introduction of AOSD techniques has a way to maintain the separation of concerns from the requirements to code.

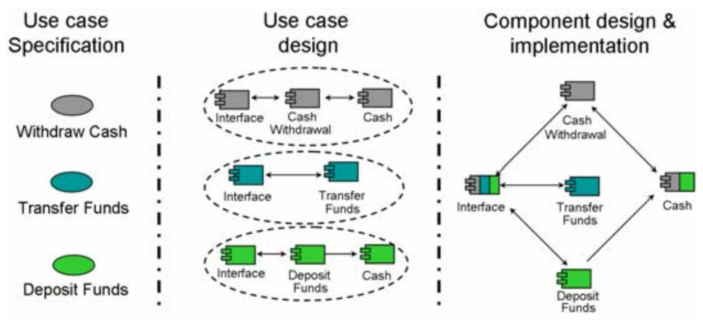


Figure 3-6 The scattering and tangling while working with a use-case driven approach (extracted from (Jacobson, 2003))

In order to maintain this SoC, Jacobson has defined an extension to UML. The main idea is to facilitate the description of the *use case slices*. They are in charge of describing the *pointcut* of a use case at each stage of the software development (use-case model, analysis model, design model, implementation model, etc) so that each use case remains separate all the way down of the lifecycle. The set of slices of a UC composes a *use case module*.

He also distinguishes between *peer* and *extension* UCs. The first depict those UC that are independent of any other described for the system-to-be, they are the basic requirements. The second represent those UC that define additional features that are hooked on the basic requirements. Both kinds of UC could be separately defined and, later on, composed by means of the *Extends* relationship. They are already provided by UML, and extended by Jacobson to support the concept of *join points*. It facilitates that an extension can be associated to a list of extension points.

According to Jacobson, AOP is the magic wand that facilitates this separation of the peer and extension UCs can be translated to design and code in a straightforward way. For this reason, he proposes that detected extensions and extension points correspond to aspects and joint points in design and code, respectively. It can be easily supported by (HyperJ, 2000) by realizing each UC as a crosscutting concern.

One of the problems this proposal shows is related to non-functional requirements. They introduce the notion *infrastructure Use Case* to model them. However, a UC depicts by definition an interaction between the system and external actors. There are some non-functional requirements that can be defined in such a way, as for instance performance, by converting them into a use case. However, it is not the case of others such as maintainability; the actors do not interact with the system in any way to get it.

In addition, there is no concept for crosscutting concerns at the requirements stage. Every UC is identified as a crosscutting concern because it crosscuts, at the design and implementation level, several classes and/or components. Nevertheless, it does not mean it is a crosscutting concern, that is, a requirement that crosscuts other requirements.

Aspectual Scenarios

(Araujo et al., 2004) have defined an approach for the exploitation of aspectual scenarios. Initially they describe the requirements of the system-to-be in a similar way to that described by (Brito & Moreira, 2003), i.e., using Use Cases for the functional requirements and templates for the non-functional ones (see chapter 2, section

Scenario Approach and AORE). Using that description, non-crosscutting scenarios are described by means of UML Sequence Diagrams and aspectual scenarios are described using Interaction Pattern Specification (IPS). The IPS, proposed by (Kim et al., 2004), is a specialization of the UML Sequence Diagrams that allow one to describe patterns by establishing the roles of the involved elements, and the expected behaviour. Once these scenarios have been described, they are translated into a set of aspectual and non-aspectual state machines for each entity specified in the scenarios. The state machines of each entity are composed in order to meet the whole set of requirements. The set of composed state machines are executable so that they can be used for validation purposes. The main advantage of the proposal is the introduction of aspectual scenarios, that is, they describe the crosscutting behaviour by means of a well-established notation. In addition, the use of a widely accepted approach for the generation of state-machines facilitate the automation of the process. However, the authors do not provide any help or guidance about how to identify the aspectual scenarios. In addition, the process cannot be performed in a fully automatic way because bindings between state machines must be introduced manually.

AO-MDSD

(Sánchez et al., 2006) have proposed a process that combines Model Driven Software Development (MDSD) and AOSD to derive aspect-oriented architectures from aspect-oriented requirements models. The first step of this process entails the identification of aspectual requirements using some of the current proposals of AORE. In the next step, the functional requirements are modelled as UML scenarios using a profile they have defined. Non-functional requirements are modelled independently as parameterized UML diagrams. Figure 3-7 shows an example of what these scenarios look like. It depicts a functional scenario called "Identification" and two non-functional scenarios "ResponseTime" and "Authenticity". The crosscutting relation between the former and the latter are specified by means of «bind» relationships that specify when and how a non-functional concern is woven to the functional one. In Figure 3-7, "who", "i_am" and "time" are specified as the actual parameters in the bind relation for the formal parameters of "ResponseTime".

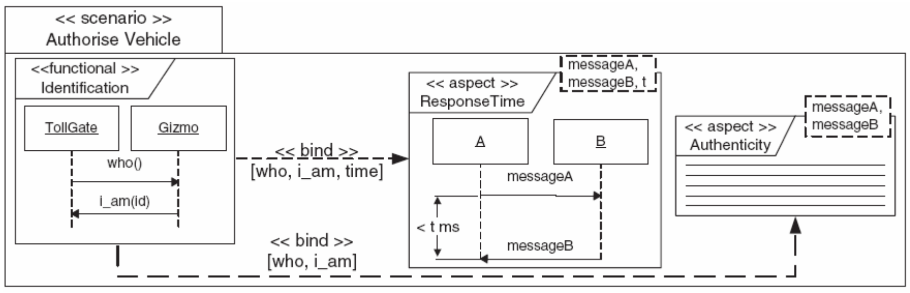


Figure 3-7 UML representation of a scenario (extracted from (Sánchez et al., 2006))

Once the set of scenarios, both aspectual and functional, have been defined, they are transformed into an aspect-oriented architecture by means of a set of transformation rules. These rules have been specified using (QVT, 2005), an standard model transformation language proposed by OMG[5]. Specifically, the AO-ADL they have used as target language is CAM (Pinto et al., 2005). This AO-ADL identifies aspects as a special kind of components that are applied outside of the components in order to obtain components reusability and composition.

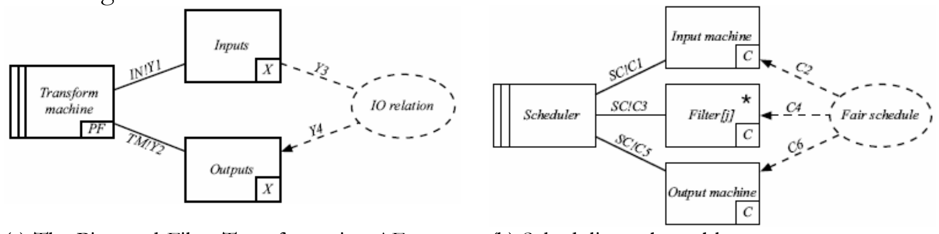
One of the main advantages shown by the proposal is the definition of aspectual scenarios. As they are parameterized, they can be woven whenever they are necessary, obtaining a proper management of the crosscutting. However, this proposal does not provide an architectural description coping with non-functional requirements. Instead, they are introduced, at the architectural level, as constraints. These constraints must be resolved at the architectural level to discuss what architectural mechanisms should be used to address them.

[5] Object Management Group, http://www.omg.org

3.2.3 Problem Frames

The combination of Problem Frames and Software Architecture has not received much attention during last years. One of the most well known approaches in this sense has been that proposed by (Rapanotti et al., 2004). They have described an extension to Problem Frames by introducing what they have called *Architectural Frames (AFrames)*. Problem Frames define requirements as dependencies among the system and the world outside of the computer. They help the developer to focus on the problem domain, establishing a clear separation between the world and the machine. However, it does not provide any guidance neither help to relate that description of the problem to the structure of the solution.

The *AFrame* has been defined to address such a problem by combining a problem class and an architecture class (Architectural Styles). The main idea is that the latter can guide the analysis and the decomposition of the former. An AFrame is defined by means of three elements: a *Problem Frame diagram* (see section 2.2.3) that describes the Architectural Style to use; a collection of *decomposition templates* that describe which problems must be faced if the selected Architectural Style is applied; and a *correctness argument* to determine the correct re-composition of every fragment by means of the application of the templates. Figure 3-8 shows an example of an *AFrame* describing the *Problem Frame diagram* for a pipe-and filter style and one of its related sub-problems: the scheduling of the filter transformation. The process proceeds by applying the same transformation as many times as needed. The templates can also be applied according to the needs.



(a) The Pipe-and-Filter Transformation AFrame (b) Scheduling sub-problem

Figure 3-8 Describing an AFrame: (a) the pipe-and-filter transformation and (b) the scheduling sub-problemn

This solution is amenable to perform iterative development due to the interaction among requirements, architecture, and design: each one informing about the others and vice versa. Both requirements and architectural specification are encoded using the *AFrame*, what facilitates that interaction between them. However, there is no mention of how the synthesis is performed when the number of iteration is greater than one.

3.2.4 Features

Most works have exploited Feature Diagrams as a support for managing properly the variability but not for defining concurrently requirements and architecture. As far as we know, (Bruin & Vliet, 2003) is one of the proposal that does address this issue. They have proposed a process for the generation of Software Architecture taking as inputs both a rich Feature-Solution graph and UCM. The former is in charge of describing the knowledge of both the functionality of the system and the quality requirements, that they have called *Feature Space* (FS). However, it is quite relevant because they also describe solution fragments at the architectural level, called the *Solution Space* (SS). They are not Feature Diagrams in the traditional sense, which was presented in section 2.2.5, but authors have extended these diagrams borrowing concepts from the goal-oriented approach. For this reason, they introduce links between the feature and the solution space to establish which influence, positive or negative, is on the solution space if a feature is selected. Figure 3-9 illustrates an example of a Feature graph, where the feature space is shown on the left, and the solution space on the right. As can be observed, both functional and non-functional features are described by refining Functional and Non-Functional in the graph. Negative links are described by means of broken lines.

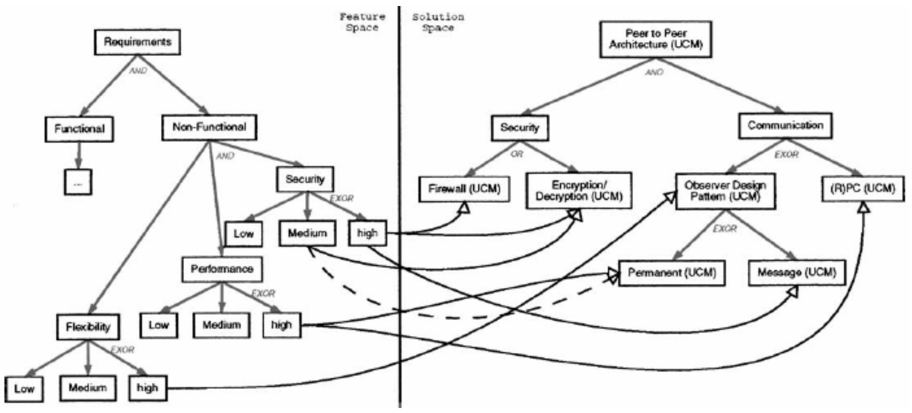


Figure 3-9 An example of a FS graph for peer-to-peer communication (extracted from (Bruin & Vliet, 2003))

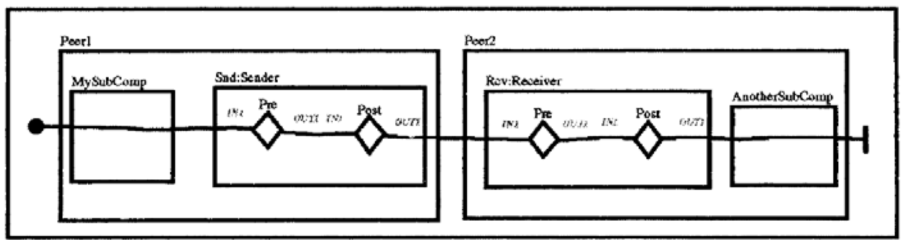


Figure 3-10 Describing a UCM for a peer-to-peer architecture

The UCMs are defined by means of a refinement process, which entails the composition of new UCMs as it progresses. Figure 3-9 describes that for the system being developed a peer-to-peer architecture has been selected. This is described by means of a UCM where *sockets* are specified. A *socket* is described as a placeholder where at most two UCMs can be plugged. For instance, in the Figure 3-10 a UCM is described that contains two components "Peerl1", that provides a socket for sending data, and "Peerl2", which provides a socket for receiving data. The solution space is built by means of a composition process. For instance, if the analyst selects the feature "High (Security)" then the feature "Firewall (UCM)" could be selected to be in the final description of the system. It would mean that its UCM would have to be plugged into the UCM described by the "Peer to Peer Architecture". This process continues all the way down until the architecture is fully defined according to the selected features of the solution space.

The main characteristic of this proposal is its ability to establish traces between the feature space and the solution space. This idea could be applied to the SPL approach where products could be built using the FS graph. It facilitates that both descriptions of the system can be maintained up to date. However, one of the major problems to apply this proposal is that the analyst needs a clear wisdom of where the sockets, in the initial description of the system, must be placed. No guidance or help is provided in this sense. This means that anticipated decisions must be made in the system. Authors do not determine how the process of selection of the initial architecture is performed either.

3.2.5 Other Proposals

Other proposals have emerged during last years that address this coupling between requirements and architecture without using any traditional approach from the RE field. In the following, some of most well known proposals in this field are introduced.

The work presented by (Westhuizen & Hoek , 2002) on software product lines and system families has also examined the relationship between architectures and requirements. The work has focused on identifying `core requirements` (identified through a process of requirements prioritising) and linking them to core architectures (identified by examining the stability of various architectural attributes over time). However, their main focus is on the architecture without a special treatment of requirements, when they are normally the source of change.

(Grünbacher et al., 2001) explore the relationships between software requirements and architectures, and propose an intermediate model called *CBSP* (Component - System – Bus - Property), to depict the dependencies among the key architectural elements and the stated system requirements. These dependencies are established in a polling process where the involved architects classify each requirement with respect to its architectural relevance. Based on the CBSP model elements a proto-architecture can be derived with the selection of the appropriate Architectural Style. Therefore, it describes a systematic process to go from requirements to architecture. However, there is not an explicit trace to the final architecture description. Neither an automatic support helps in the process but the architects must summarize the results of the trade-offs to establish manually the description.

(Wile, 2001) has examined the relationship between specific classes of requirements and their equivalent dynamic architectures. It aims at enabling requirements engineers to monitor running systems and their compliance with these requirements. However, the focus of this work is *runtime monitoring*, not more traditional development activity.

The method REVEAL, proposed by (Hammond et al., 2001), is based on a clear separation between the world and the machine in order to provide a practical approach to developing systems, taking a wide background from the Problem Frames approach (section 2.2.3). For this reason, they understand the system according to their interaction with the real word where it works. REVEAL assumes a traditional development process where system is refined progressively from the structuring of functional areas to the identification of subsystems, as requirements are more detailed. The outcome of the process is the identification and documentation of the subsystems along with the interfaces between them. The process also introduces the concept of *rich traceability*, that is quite similar to the use of AND/OR relationships in the Goal-Oriented approach (section 2.2.1). However, there is not any information about how these relationships are exploited to reason about the satisfiability of the specification.

AOCE

(Grundy, 1999) has proposed Aspect-Oriented Component Engineering (AOCE), in order to define and develop software components from component requirements. The author states that traditional approaches of RE do not provide the specific expressiveness to describe generic interfaces for individual components. With this aim, the author introduces the notion of *systemic aspect*, i.e., a system concern (e.g. user interface, persistence, etc) for which components provide and/or require services. These systemic aspects catalogue the services a component requires or provides related to the functional and non functional specified requirements.

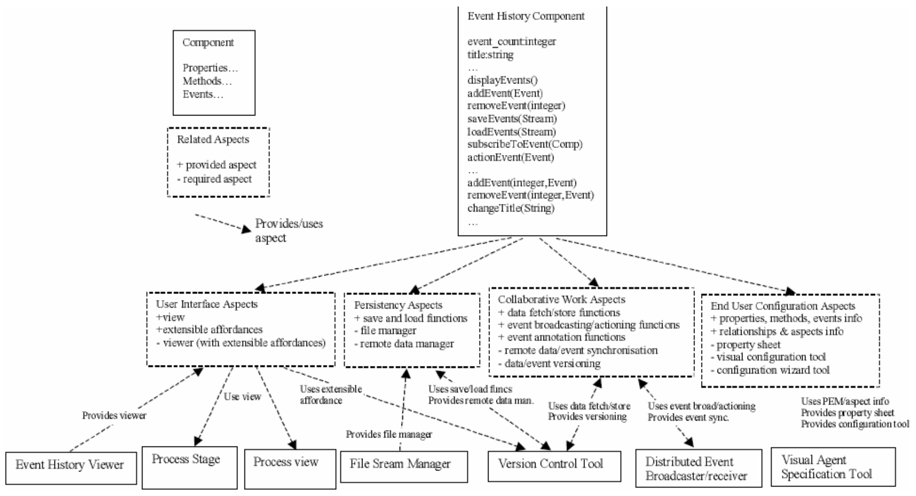


Figure 3-11 Example of components and their relationships to systemic aspects (extracted from (Grundy, 1999))

In order to identify and specify the systemic aspects, a set of components are pre-selected using the system requirements. This set is analysed to determine the systemic aspects and the provides/requires relationships between them and the components. Figure 3-11 depicts an example. It can be observed, in the bottom level, the set of components that were identified using the system requirements described in the top level. Every component has provides/requires relations with the systemic aspects that are identified in the medium level by means of dotted rectangles. These relationships are useful to determine the relationships inter-components. Once systemic aspects and their relations to components are identified, the *aggregate aspects* are identified. They are requirement aspects that are related to a set of component or even the whole system. These aggregate aspects are used to reason about the established relationships, specified aspects or, even the selected components. This reasoning could lead to determine that new components could be added, some components could be modified, and thus, new or modified requirements aspect could be determined. Once this reasoning is done, an evaluation is performed to determine whether the system requirements are satisfied. Otherwise, an iteration would be performed to select or add new components and/or requirement aspects.

Therefore, AOCE reveals as an engineering approach that covers the lifecycle of component engineering, from component requirements and specification, to implementation. However, the proposal does not address one of the main characteristics of AOSD: the management of the crosscutting. In addition, the identification of aspects is quite directed by the set of the selected components, what means that the outcome is not directed by the system requirements. It also exhibits some problems related to the management of change because if the system requirements change there is not explicit mechanisms to deal with the impact in the systemic aspects and the identified components.

3.2.6 Main discussion

To the best of our knowledge, there is not a comparative framework that can be used to evaluate the proposals presented above. In this sense, only the work of (Chitchyan et al., 2005) identifies some features that can be used to analyze AORE but without paying attention to the whole process, that is, from requirements to architectures. They have identified four criteria for the analysis: `traceability` through software lifecycle; `composability` of the different artifacts to improve the understanding of the system as a whole; `evolvability` that is related to the ease of change; and `scalability` according to the size of project. Reader is referred to that work where some of the works presented in section 2.4 have been analyzed using such features. However, due to the different motivation of this analysis, we have established our own comparison framework including a set of features that should be present in any proposal having as goal the generation of proto-architecture from requirements having the AOSD as cornerstone of the process. In the following, these features are introduced along with the reasons that have motivated their selection:

– `Requirements Model for the process` (RE Model). Requirements are the foundation to build the system. A well-established Requirements Model is mandatory if a right specification of the system must be provided. It is also especially relevant in terms of the traceability to be provided by the process.

- `Functional and Non-Functional requirements` (F & NF). Non-functional requirements present the main constraints to be satisfied when the Software Architecture of the system is being defined. For this reason, any process should include specific mechanisms for both specifying and dealing with such kind of requirements.
- `Separation of Concerns throughout the lifecycle` (SoC). As was stated above, the SoC introduces clear benefits in terms of maintainability and reuse along with a way to manage the complexity of software development. For this reason, its appropriate management throughout the lifecycle means a clear benefit for the final product.
- `Traceability throughout the lifecycle` (Traceability). The introduction of proper mechanisms of traceability is one of the cornerstones necessary to evaluate the impact of change, follow properly the realization of requirements, etc. This means that the introduction of mechanisms to manage this traceability from requirements to code means a clear benefit for the software development. As can be observed in Table 3-1, not all proposals provide support for traceability throughout the full lifecycle but up to a specific stage of software development.
- `Alternative analysis` (Alternatives). It is frequent the case that an alternative must be selected among a set of potential solutions when an architectural decisions is being made to satisfy a specific requirement. This means that a trade-off must be made among them in order to select the most proper one. For this reason, the exploitation of mechanisms to perform such analysis means a meaningful advantage for any proposal.
- `Architectural Styles and/or Patterns` (Arch. Style). Both Architectural Styles and Patterns help the analysts in the process of describing Software Architecture because their use means the reuse of quality solution. In addition to this guidance, they also convey much information about the decisions that are made. This leads to consider that its use can provide a significant improvement for the solution described.
- `Automatic or semi-automatic support for the generation of the proto-architecture` (Automation). The generation of a proto-architecture can be an error-prone and/or cumbersome task as the system-to-be become more and more complex. In this case, the number of requirements, the traceability links to the made decisions, etc, can be quite significant. This means that the automation of this task as much as possible can be another step towards the generation of quality solutions.

– `Tool support` (Tool). Any kind of process shows a high demand in terms of support to speed up and guide the stakeholders throughout its application. Several persuasive reasons can be stated for the process of intertwining the definition of Requirements and Architectures. First, it is especially relevant when most of the artifacts to be produced, such as Requirements Models and Architectural Models, have a graphical representation to improve their legibility. Second, there are tasks susceptible of being automated. Third, the establishment and maintenance of the traceability can be quite difficult if there is not support for it. They are compelling arguments for requiring the support of tools.

A comparison among the presented proposals was performed using the features described above. Table 3-1 and Table 3-2 present the results obtained, where each row describes the comparison for a proposal in the same order as it was described. Some cells have been left blank because the needed information was not available.

Table 3-1 Determining the satisfaction of the established features (first part)

Approaches		RE Model	F & NF	SoC	Traceability
Goal-Oriented	KAOS	Defined	Yes	No	Up to design
	Architectural Prescriptions	Defined	Yes	No	Up to architecture specification
	GBRAM	Defined	Yes	No	Up to architectural decissions
	TROPOS	Defined	Yes	No	Up to code
	L'Escritorie	Defined	Yes	No	Up to requirements
	GRL	Defined	Yes	No	Up to architectural decisions
Scenarios	AOSD/UC	Defined	Infrastructure use cases	Yes	Up to design
	Aspectual Scenarios	Not specific model	Not defined	Up to design	Up to design
	AO-MDSD	Not selected	Yes	Yes	Partially, from scenarios to architecture

Approaches		RE Model	F & NF	SoC	Traceability
PF	AFrames	Defined	Yes	No	Up to architectural decisions
Features	Features & UCM	Defined	Yes	It is recommended but not addressed specifically	Up to architecture
Others	Core Requirements	Not defined	Not defined	No	
	CBSP	Not defined (intermediate model CBSP)	Yes	No	Up to architecture specification
	Runtime monitoring	Textual description	Not specifically	No	No
	REVEAL	Similar to Goal-Oriented	Not specifically	No	No
	AOCE	Textual description	Not specifically	Concept of aspect is not explicit	No

Table 3-2 Determining the satisfaction of the established features (second part)

Approaches		Alternatives	Styles & Patterns	Automation	Tool
Goal-Oriented	KAOS	Yes	It is recommended but not dealt with	No	Objectiver: (requirements & design specification)
	Architectural Prescriptions	Yes	No	No	GBRAT (requirements specification and analysis)
	GBRAM	Yes	Organizational and technological styles evaluated for use but not automatically managed	Set of rules identified but it is not documented its use	TAOM4E (generation of the architectural design is not documented)

Approaches		**Alternatives**	**Styles & Patterns**	**Automation**	**Tool**
Goal-Oriented	TROPOS	Yes	No	No	CREWS-L'Ecritoire tool(requirement specification)
	L'Escritorie	Yes	No	No	OME (under development)
	GRL	No	No	No	
Scenario	AOSD/UC	No	No	No	No
	Aspectual Scenarios	No	No	Partially, up to the generation of state machines, but weaving is manually established	Not specific Tool
	AO-MDSD	No	No	QVT as definition language but it has not been tested	No
PF	AFrames	Yes	Used but not evaluated nor automatically managed	No	No
Feat.	Features & UCM	Yes	No	No	No
Others	Core Requirements	No	No	No	Integration:Arch Edit+ xArchADT+ Apigen+ XML Spy+ Data Binding+ Libraries + Apache Xerces
	CBSP	Yes	No	No	Integration: GSS
	Runtime monitoring	No	No	No	No
	REVEAL	Yes	No	No	DOORS
	AOCE	Yes	No	No	JComposer

As can be observed, there is no proposal that satisfies the whole set of features established above. This has motivated the definition of ATRIUM, presented in the following section.

3.3 Our Proposal: ATRIUM

ATRIUM is a methodology intended to define concurrently Requirements and Architectural artifacts by exploiting the Separation of Concerns as key to improve the maintainability and adaptability of software artifacts. It focuses its efforts on detecting and specifying this separation from the very beginning of the software lifecycle until the architectural stage.

Using AOSD, functional and non-functional needs, such as performance or compatibility, of the system's behaviour can be separately acquired and specified across the development lifecycle improving the maintainability and reusability. This traceability across the software lifecycle is necessary to satisfy the demanded closure property (Elrad et al., 2001b).

It is because ATRIUM deals with functional and non-functional requirements, it exhibits a clear difference from other approaches which propose a functionality-based design of the architecture. In this sense, this approach tries to address the drawback, identified by (Bosch, 2000), of those architectural designs not fulfilling the quality requirements of the system-to-be. In addition, (Bosch, 2000) also states that a bottom-up reuse does not work in practice. For this reason, ATRIUM provides a refinement from requirements to architecture.

ATRIUM has been described as an iterative and incremental process. This is because the generation of the Software Architecture from Requirements is not as straightforward as to define all the artifacts at once. Therefore, it seems natural to provide the analyst with an iterative process in order to develop incrementally the Software Architecture. It allows him/her to reason on `partial views` of the architecture, that is, only on partial descriptions of the architecture. In addition, mechanisms for traceability throughout the lifecycle have been incorporated so that the analyst can evaluate the impact of the changes, reason about the realization of the requirements, etc.

In the following sections, the models and the processes that have been identified and specified for the description of ATRIUM are briefly presented. It has been described how they have been more widely detailed in the following chapters.

3.3.1 Models for ATRIUM

Currently, the Model-Driven Development (MDD) (Selic, 2003) is not only becoming more and more embraced by the researchers but by practitioners as well. It has demonstrated positive influences for the reliability and productivity of software development process due to several reasons. It allows one to focus on the ideas and not on the supporting technology. It facilitates not only the analyst get an improved comprehension of the problem to be solved but also the stakeholders obtain a better cooperation of the software development. With the aim of reliability and productivity, MDD exploits the models both to properly document the system and generate automatic or semi-automatically the final system.

This paradigm has been followed for the definition of ATRIUM, in such a way that three models have been defined (Figure 3-12), from the requirements to the architectural stage, that are briefly described in the following. In addition, whenever automatic transformation can be defined, they have been introduced to speed up the process and avoid errors as much as possible.

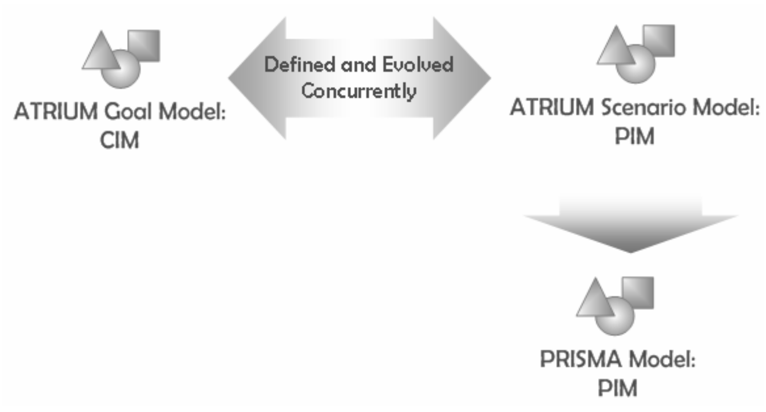


Figure 3-12 Models for ATRIUM

ATRIUM Goal Model

The Goal-Oriented approach has become highly relevant in the Requirements Engineering arena mainly due to two advantages:

- Its ability to specify and manage positive and negative interactions among goals, as state (Chung et al., 2000) and (Lamsweerde et al., 1998a), allows the analyst to reason about design alternatives.

– Its capability to trace low-level details back to high-level concerns, described by (Dardenne et al., 1993), is very appropriate to bridge the gap between architectural models and requirements.

These are the main reasons why the Goal Model is interesting as a Requirements Model to identify and describe the users' needs and expectations, their relationships and how these can be met by the target system. Therefore, it is going to play the role of the Computation-Independent Model (CIM) because it is in charge of gathering the requirements of the system-to-be.

Furthermore, the Goal-Oriented approach allows for a proper separation of system concerns, in such a way that evolution, adaptability, comprehensibility, etc, can be achieved. Those concerns can be goals/requirements that are either functional, i.e., services that the system-to-be has to provide; or non-functional, such as security or fault-tolerance.

These advantages have been also detected by the industrial community. According to (Lamsweerde, 2004) and his lessons learnt from the evaluation of a wide set of real projects:

> *"Decision makers looked at goal models carefully, paying special attention to alternative goal refinements, operationalizations and responsibility assignments; they did not care too much about UML object models; annotated goal diagrams were found to be more helpful for focussed brainstorming, validation, negotiation, and decision making than fairly vague use case diagrams".*

These are the main reasons why the *Goal Model* has been introduced as an ATRIUM artifact to identify and describe the users' needs and expectations, their relationships and how these can be met by the target system. The ATRIUM Goal Model is widely described in chapter 5. How this Model has been exploited in a real case study is presented in 6.

ATRIUM Scenario Model

One of the main outcomes of ATRIUM is the generation of a `proto-architecture`, (Brandozzi & Perry, 2001), i.e., an initial description of the architecture to be refined in a later stage of the software development. In order to facilitate this task, it is necessary to provide some partial description of how the requirements established in the Goal Model can be operationalized. These partial descriptions act a Platform Independent Model (PIM) because it does not include any detail from the point of view of the platform.

In order to obtain that description, a possible alternative could be to identify the architectural elements, which collaborate in the operationalization of a requirement, identifying a partial structural description of the architecture.

However, when the Software Architecture is defined, it is mandatory not only to identify these elements but how they collaborate and how their behaviour is going to meet the requirements.

Therefore, in the ATRIUM Scenario Model, a scenario is used to describe the system behaviour associated to some requirements and under certain operationalization decisions. A scenario is described by the sequence of interaction messages in accordance with a specific choreography. Unlike classic scenarios proposals, the ATRIUM scenarios specify architectural elements interaction instead of objects along with the environmental agents that played a role in that scenario. Sequence diagrams (UML, 2005) have been selected as the language to specify the ATRIUM Scenario Model. Mainly because it is a wide extended language with a well-known notation. It is also flexible enough to be adapted to the specific needs of ATRIUM.

It has to be emphasized that the Scenario Model provides us with partial views of the architecture, that is, partial descriptions of the architecture. These partial views only identify shallow-components, shallow-connectors and shallow-systems along with their behaviour expressed through interaction. They are called shallow because we do not need their complete definition but an initial one that can be refined for their later compilation to code. This model is detailed in chapter 7.

PRISMA Model

The architectural artifacts to be produced must integrate two approaches: AOSD and CBSD. Both approaches present a high potential when used in an integrated way because of their modularization criteria are orthogonal (Atkinson & Kuhne, 2003). For this reason, an AO-Architectural model was selected as target of ATRIUM. Specifically, PRISMA (Pérez, 2006) has been chosen to describe the proto-architecture. It is because its use allows us to establish the separation of concerns detected and specified in the previous models because it integrates both CBSD and AOSD. Thanks to its definition as an extension of a formal language called OASIS (Letelier et al., 1998) it provides facilities for verification purposes.

It must be highlighted that, similarly to the ATRIUM Scenarios Model, it provides a PIM view of the system because it is not dependent of the platform. In addition, one of the main advantages of PRISMA is that it also provides support to a Platform Specific Model (PSM). It is because PRISMA has been developed to support for the automatic generation of C# code and its execution by means of a middleware.

A more detailed description of PRISMA is presented in chapter 4, section 4.3, in order to introduce briefly the concepts that have been used in this work. Reader is referred to (Pérez, 2006) to obtain a whole description of this Architectural Model, its AO-ADL and platform.

3.3.2 A process for ATRIUM

ATRIUM is a methodology oriented to the concurrent definition of Software Architecture and Requirements. With this aim, ATRIUM provides the analyst with guidance, along an iterative process, from an initial set of user/system needs until the instantiation of the architecture. This process entails three activities, to be iterated over, in order to define and refine the different artifacts and allow the analyst to reason about partial views, of both requirements and architecture. Figure 3-13 illustrates the three main ATRIUM activities, which are described bellow, to be iterated over.

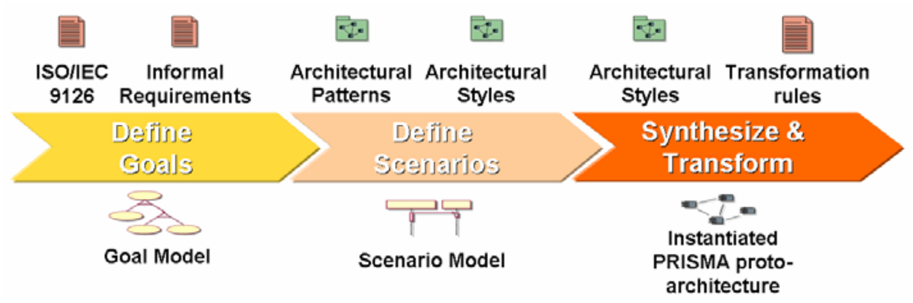


Figure 3-13. ATRIUM: activities and artifacts

Activity 1. Define Goals.

Two inputs are used to start on this activity. One of them is an informal set of user/system needs, usually stated in natural language. The other one is the (ISO/IEC 9126) quality model that is used as an instantiable framework, providing the analyst with an initial set of concerns of the system that him/her can select.

The main aim of this activity is to specify the set of goals to be satisfied by the system-to-be. Considering an initial selection of concerns they are refined until functional and non-functional requirements are established. During this refinement, the detected crosscutting among goals and requirements is also established. This is a first step for the identification of concerns used in the detection of candidate aspects in the architecture specification, and their realization through aspects integrated into components and connectors.

More details about how this process is performed are presented in chapter 5. An example of its exploitation is presented in chapter 6.

Activity 2. Define Scenarios.

The Goal Model obtained in the previous step is the main input for the elaboration of the Scenario Model. In this sense, this activity is focused on the identification of the set of scenarios that describes the system's behaviour under certain operationalization decisions. Each scenario depicts the elements that interact to satisfy a specific requirement of the Goal Model and their level of responsibility for achieving it. It facilitates that the analyst can focus on partial views of the system description.

In order to describe these scenarios, two additional inputs are used during this activity: patterns and Architectural Styles. The former are going to help us to reuse partial solutions to describe a scenario. The latter will help us to identify and select the suitable patterns to be used and it will give us some initial sketch about how the scenarios should be defined.

Using scenarios an advantage can be taken. We are not only identifying the elements, which can appear into the description, but also the coordination structure through the temporal sequences of interaction events. This means that an initial description of the system behaviour can be described.

The elaboration of the Goal and Scenario Models are two intertwined processes. The Goal Model is operationalized at the next activity by means of the Scenarios Model. Furthermore, the analysis information provided by the Scenarios Model helps us to refine and identify new goals at the next iteration. Therefore, both models are coupled, as in (Rolland et al., 1999), with a meaningful advantage in terms of traceability. This activity is detailed in chapter 7.

Activity 3. Synthesize and transform.

It aims at obtaining a proto-architecture for the concrete system thorough scenarios synthesizes and using related Architectural Styles and interaction patterns. The transformation from scenarios to proto-architecture is achieved by means of a set of transformation rules. They are described by means of a language that provides with meaningful advantages in terms of the generated Architectural Model, the incorporation of Architectural Styles, etc. This activity is detailed in chapter 8.

As ATRIUM is intended to be iterative and incremental, a feedback is provided from Activity 3 to 1. In this way, all the models are up-to-date all over the process. This process is supported by a tool called MORPHEUS that is presented in chapter 9.

3.4 CONCLUSIONS

In this chapter, a summary of the most well known proposals focused on the definition of Requirements and Architectures has been introduced. It can be observed in the evaluation performed, that a reduced number of proposal have oriented its efforts to provide support to the traceability of the SoC from the early stages of development to the SA definition, taking into account both functional and non-functional requirements. It has been shown as well, that nearly none of them introduces automation in this process, what can difficult its acceptance and deployment.

Finally, we have introduced ATRIUM as a methodology to address the iterative development of requirements and architectures during the development of software systems. ATRIUM guides the analyst, from an initial set of requirements to an instantiated proto-architecture. It uses the strength provided by the coupling of scenarios and goals systematically to guide through the iterative process. Moreover, it supports the traceability between both artifacts to avoid lacks of consistency providing support to the SoC. Finally, it also introduces mechanisms for the automatic generation of the proto-architecture. It must be also highlighted that a tool, which helps throughout the process of its application, supports this proposal. All these topics constitute the main proposal of this work and are developed in the following chapters.

The work related to the motivation and initial description of ATRIUM has been presented in the following publications:

- E. Navarro and I. Ramos, "Requirements and Architecture: a marriage for Quality Assurance", VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante, Novembre 12-14, 2003, ISBN 84-688-3836-5, pp. 69-78.
- E. Navarro, I. Ramos, J. Pérez Benedí, "Software Requirements for Architectured Systems", Proceedings of 11th IEEE International Requirements Engineering Conference (RE'03), Monterey, California, USA, September 8-12, 2003, IEEE Computer Society 2003, ISBN 0-7695-1980-6, pp. 365-366 (short paper).

"If knowledge can create problems, it is not through ignorance that we can solve them."
Isaac Asimov

CHAPTER 4
Preliminaries

4.1 INTRODUCTION

Despite presenting in the previous chapter several proposals and approaches for Requirements Engineering and Software Architecture, some more detailed information related to the field have to be presented before introducing how ATRIUM proceeds. For this reason, this chapter contains some introductory material that will facilitate the comprehension of ATRIUM.

Because always an explanation through examples facilitates the legibility and comprehensibility of the work, the case study that was used for validation purposes is introduced in section 4.2. Specifically, the EFTCoR system and the Teachmover are introduced. The former was the seed for the work presented in chapter 5 because it presents a complex system with demanding needs in terms of requirements specifications. The latter is a reduced case study that will facilitate the introduction of examples for the remainder chapters.

As was described in the previous chapter, PRISMA is the AO-ADL used for the description of the proto-architecture. In order to make it clear what the meaning of each term related to PRISMA is, a brief description of its Metamodel is presented in section 4.3. This is especially useful for the comprehension of chapter 8.

4.2 TELE-OPERATED SYSTEMS

It is more and more frequently the case that Robots are used to perform tasks in critical domains such as rescue, military battles, mining and bomb detection, scientific exploration, etc. The behaviour of these robots must be controlled according to the requirements of the domain and by the practitioners of the system.

It is in this context where tele-operated systems have appeared. They try to provide the practitioners with an effective communication and interaction medium between robots and humans. Robots work in environments and/or perform tasks that are highly dangerous for humans. For this reason, humans are taken away from the areas where robots work.

These systems are characterized by having always two basic components in their construction: `sensors` and `actuators`. Both components are the actual interface between the control software and the hardware elements of the systems. Actuators are responsible for sending operations to the joints of the robot. Sensors are in charge of reading the results of such operations to check whether they have been properly executed. Depending on the system, the number of sensors and actuators can vary.

Environmental Friendly and cost-effective Technology for Coating Removal (EFTCOR, 2003) is one of these tele-operated systems. The EFTCoR system is a tele-operated platform for non-pollutant ship hull maintenance operations that is described in the following section.

4.2.1 EFTCoR: Environmental Friendly and cost-effective Technology for Coating Removal

The main scenario of the EFTCoR is a family of systems for hull maintenance operations. Mainly, it addresses operations of coating removal, washing and re-painting of hull of ships by using a family of robots that performs either different operations or the same operation but in a different way.

Because it is a family of systems, the benefits of a product-line approach have been detected in the EFTCoR. Previous projects, such as (GOYA, 1999) or (LARLASC, 2002), have helped to detect a great number of commonalities with valuable assets which could be reused and exploited to deploy several products. These assets include both hardware and software components and an architecture of reference. The introduction of this product-line approach would allow not only an increase of productivity in terms of development but also of maintenance costs.

These maintenance operations have a high impact in both, economical and environmental terms. The former is related to the time that the ship must go into the dry dock and to the costs derived of its maintenance. This means that a high performance is demanded for these operations, so that this time can be reduced to the minimum. The latter is due to the generated residues along the operations. They are paint, iron, etc, whose recycling is always required.

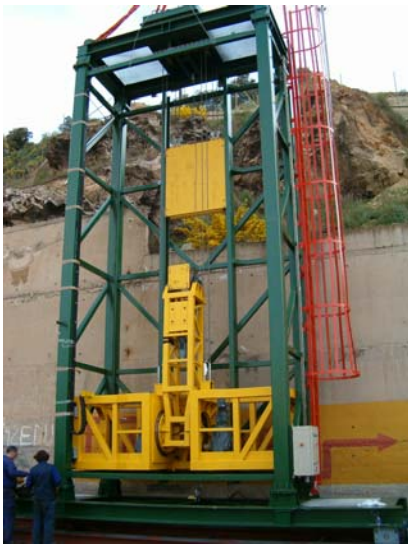


Figure 4-1. Primary Positioning System with both arm joint (yellow) and joint on tracks (green) of the EFTCoR

Furthermore, these processes are very hazardous for operators. Not only residues can come off and damage the operator, but also the self-movement of the robots can be very risky. Figure 4-1 shows an example of a primary positioning system. It has a height of twelve meters and a weight of twenty tons that make inevitable the consideration of safety requirements for the movement of the robot. The crane has in its central zone an articulated arm of two tons with a secondary positioning system at its end (an XYZ-table that includes a cleaning tool). It is indispensable that the system ensures a safe movement of the arm according to the received commands from the operator.

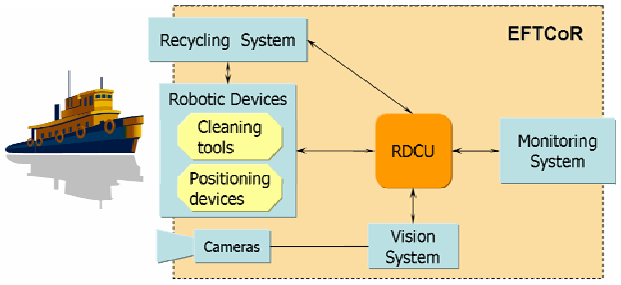


Figure 4-2 Tele-operation Robotic System for Hull Maintenance Operations

The identified robotic tele-operation platform consists of the following subsystems (illustrated in Figure 4-2):

– *Robotic Devices:* are in charge of both the movement of the EFTCoR and the cleaning task to be performed, that is, full blasting, spot blasting, etc., according to the commands introduced by the operator. *Primary* and *Secondary positioning systems* integrate the positioning devices. The former are in charge of the movement of the tele-operated platform across wide areas, in order to place the secondary system. The latter are in charge of the movement of the cleaning tools. Both Primary and Secondary systems are integrated by several joints to facilitate the movement. For instance, Figure 4-1 depicts an example of a primary system formed by an arm joint and a joint on tracks.

– *Vision System*: allows the inspection of the working areas in order to determine the areas of the hull to be cleaned and its state before and after the maintenance. In addition, it provides information for the automatic movement of the robotic devices across the hull surface.

– *Monitoring System*: encompasses the functionality concerning to the informational and managerial needs related to the ship maintenance operation that is going to be accomplished. In order to accomplish such task it exploits the information supplied by the Vision System.

– *Recycling System* retrieves the residues from the working areas and recycles them. As these residues have to be retrieved on line, there is a strong relation between the functioning of the blasting tool and the functioning of the recycling system. In fact, it is possible to consider the cleaning head as part of this system, though it is partially controlled by the Robotic Devices Control Unit (RDCU, Figure 4-2). This means that the RDCU and recycling systems must exchange a significant amount of signals.

– *Robotic Devices Control Unit*: interacts with the other robotic devices with the aim of getting the needed information to control the different devices (positioning systems and cleaning tools) to be used in the maintenance tasks. Control operations are accomplished according to the commands introduced by the operator.

Our case study focuses on the RDCU because it is not only software, but it is software-intensive. Its architectural definition is highly relevant because of the fact that several constraints have to be satisfied such as the support for dynamic behaviour of the system. This dynamism allows the EFTCoR to replace, at run time, each cleaning tools and positioning devices. For this reason, the RDCU should supply mechanisms for configuring both the systems and the tasks to be performed. In addition, the system must move during the operation so that it can go along the ship hull surface looking for spots, cleaning the surface and painting whenever it is needed. This means that the RDCU is in charge of commanding and controlling in a coordinated way the positioning of devices together with the tools attached to them.

Every operation has to be scheduled to accomplish strict deadlines. This comes out from the high cost derived not only from the budget required for these operations but also from the lack of incomes when the ship is in dry dock.

RDCU has also rigorous constraints in terms of reliability. It must work during hundreds of hours without stop in order to achieve the hard deadlines of the process. This implies that it should be reliable enough as to avoid the ship to stay in the dock more time than the needed for the maintenance operation. Furthermore, the RDCU should allow the system keep on working despite some failures, that is, it should admit degraded modes of operation.

Regarding availability and reliability, it is also recommendable that diagnosis services are provided by the RDCU. They are not only responsible for notifying if anomalous states arise but they should check as well, whenever the system is reconfigured, if the system remains working properly.

Any change or operation has to be safe, providing a means to stop it if any damage can be produced to the equipment, the environment or the operator. The RDCU must be fault tolerant in such a way that the system should be led to a safe and known state. These recommendations, among others, must be established by the safety regulations of the RDCU. They must be compliant with the standards and safety rules applicable to industrial installations such as (ANSI/RIA, 1999).

4.2.2 TeachMover

The Microbot TeachMover is a durable, affordable robotic arm used for the teaching of robotic fundamentals. It has been specifically designed to simulate industrial robot operations. This is why it has been also used in the context of this work to exemplify some terms and scenarios that would be too complex if the RDCU of the EFTCoR was used. This means that this will be the only robotic system that is considered in this work. However, similarly to the EFTCoR is the RDCU the final goal of our work.

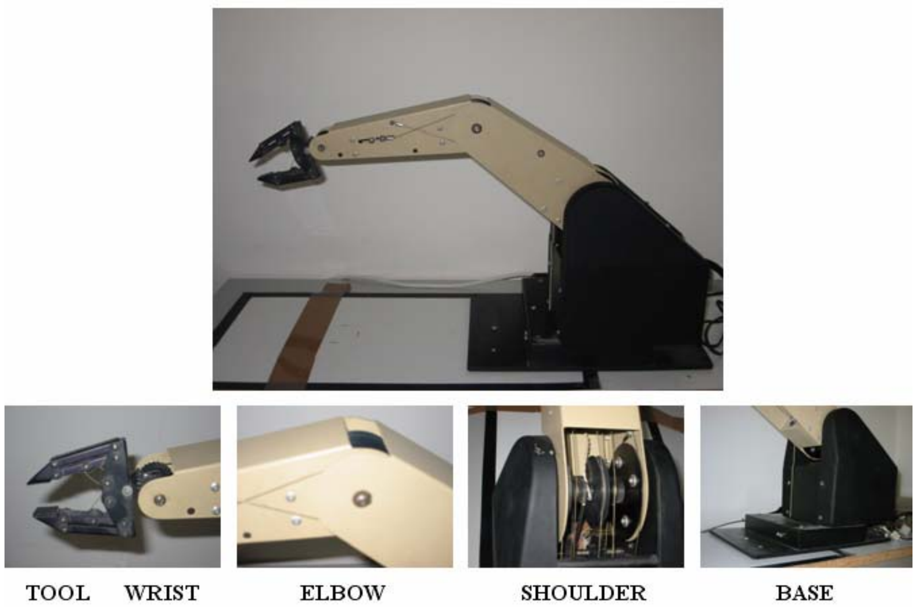


Figure 4-3 An illustrative view of the Teachmover

Figure 4-3 shows how the Teachmover looks like. It can be observed what the main joints of the robot are: the *tool* used to catch any necessary instrument; the *wrist* employed to articulate the movement of the instrument; the *wrist*, *elbow*, *shoulder* and *base* operated to close the robot to the working area. These five joints are the five axes of movement of the robot. The TeachMover's possible motions include base rotation, shoulder bend, elbow bend, wrist pitch and wrist roll, so it can simulate the motions of most industrial units in production situations. It must be mentioned that each joint and the tool are controlled by an actuator and monitored by a sensor.

Figure 4-4 depicts each joint along with its possible movements. Two kinds of movements are supported by the Teachmover:

– *Movement by steps* is specified by establishing the number of half-steps that a joint must move.

– *Movement by inverse kinematics* allows the Teachmover going to a specific point by specifying (x, y, z). These coordinates are translated to a set rotation angles. These angles are used for bending the base, shoulder, and elbow, the pitch and roll movements of the wrist and the open of the tool. These angles specify the degrees that each joint must have in relation to the Teachmover axis.

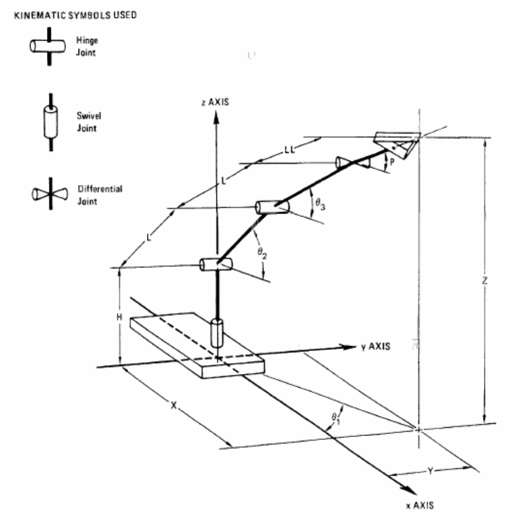


Figure 4-4 Geometry of joints of the Robot Arm

Both kinds of movements are performed according to a speed that can be changed if it is need. It must be highlighted that although both movements are allowed, the actuator internally communicates the movement to the joint by half-steps. This implies that the control unit to be developed must translate properly the movements.

Similarly to the EFTCoR, the Teachmover has rigorous constraints in terms of safety. Each movement must be safe before it can be performed. This means that if a movement can put in danger any element of the environment or the robot itself, it must not be executed.

4.3 An Introduction to PRISMA

As was described in section 3.3 PRISMA has been selected as AODL to describe the proto-architecture, that is, the final artifact of ATRIUM. PRISMA has been defined as an extension of a formal language called OASIS (Letelier et al., 1998), to provide semantics expressiveness for architectural models and allow the compilation and automatic generation of code schemas to be implemented. PRISMA is intended to design a great diversity of complex information systems, with a very dynamic nature such as on-line applications.

In PRISMA, type specifications include a set of elements, which are first order citizens: components, connectors, systems and aspects. Components are compositional units that provide specific functionalities to the system. Ports, whose types are interfaces, are their connection points to describe their interaction with the rest of the system. Connectors have connection points called roles and are typed by an interface. They are defined in terms of interactions among components and other connector. For this reason they provide a high cohesion and a loose coupling. Moreover, the separation of concerns, together with the difference among components and connectors, facilitates maintenance and reuse of large and complex software systems.

Components and connectors are defined in terms of composition aspects: functional, distribution, coordination, etc, following the Aspect Oriented Software Development approach. Crosscutting between concerns is managed because the integration is carried out at a high abstraction level, generating code separately for each aspect.

A system is a collection of components and connectors, together with a specification of how specific roles and ports are engaged with each another. All of the PRISMA elements – systems, components, connectors, ports, and roles – may be given specific properties. These are used to convey any non-topological properties of the described architectures.

In the following sections, a more detailed description of these elements is introduced. They briefly introduce the PRISMA metamodel. Sections 4.3.1 to 4.3.5 describe the types that can be used when a PRISMA model is being defined. Section 4.3.6 describes the mechanisms to build an architectural model. This description will facilitate the comprehension of the terms and metamodel elements used along this work. A more detailed description of both the PRISMA language and the PRISMA model can be found in (Pérez, 2006).

4.3.1 PRISMA Interfaces

In order to describe the visibility of services Interfaces are used in PRISMA. A PRISMA Interface describes a set of services to be published, InterfaceServices, whose behaviour is not defined. That is, an Interface only describes the signature of the InterfaceServices by identifying their names and parameters (Figure 4-5). Parameters are declared in a specific order describing its name and kind (input/output) (Figure 4-6).

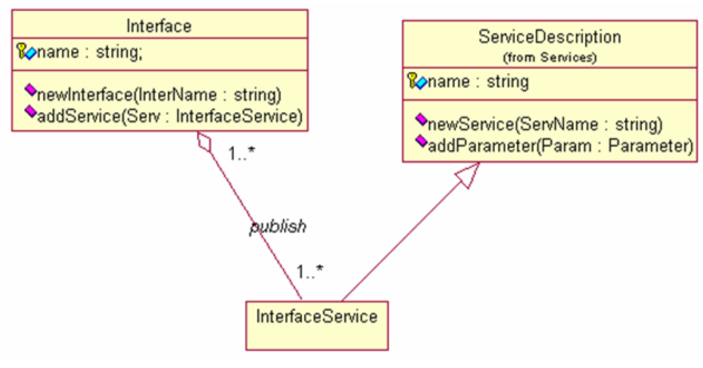

Interface
name : string;
newInterface(InterName : string)
addService(Serv : InterfaceService)
ServiceDescription
(from Services)
name : string
newService(ServName : string)
addParameter(Param : Parameter)
1..*
publish
1..*
InterfaceService

Figure 4-5 Describing PRISMA Interfaces (extracted from (Pérez, 2006))

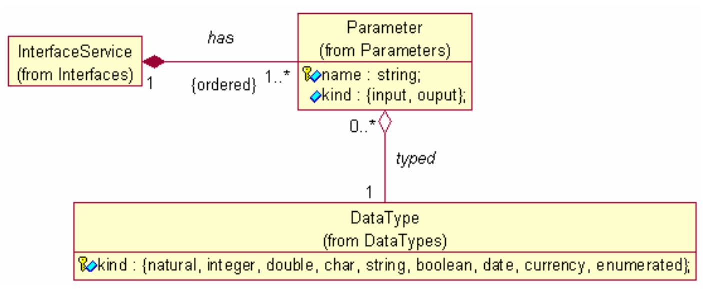


Figure 4-6 SignatureOfService of the PRISMA metamodel (extracted from (Pérez, 2006))

The PRISMA Interface metaclass has two services: *newInterface* for the creation of new interfaces; and *addService* for the addition of new services to the interface.

4.3.2 PRISMA Aspects

As was stated above, PRISMA exploits aspects as its main characteristic for the separation of concerns while defining a system. They are the minimal computation units that can be described in PRISMA according to some specific concern such as functionality, distribution, safety, etc.

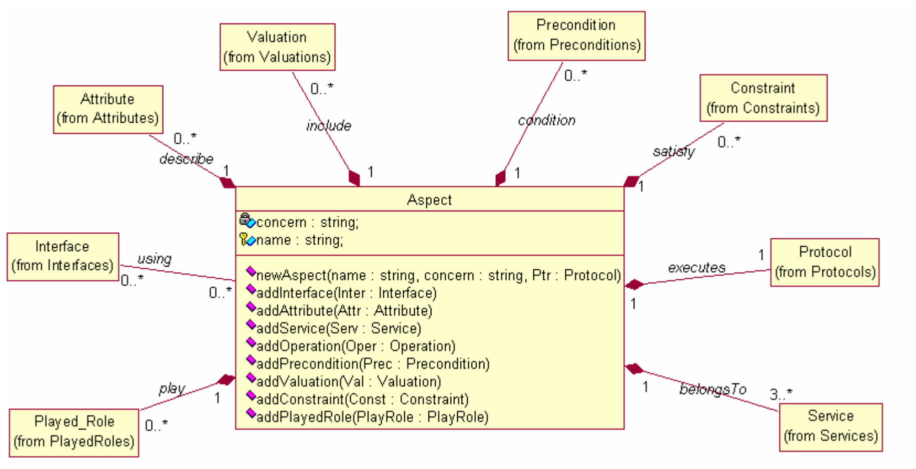


Figure 4-7 The metaclass Aspect of the package Aspect of the PRISMA metamodel (extracted from (Pérez, 2006))

Figure 4-7 shows how an Aspect is defined in the PRISMA metamodel. It can be observed that during its description the architect can describe:

– `Attributes` which store any needed information to perform successfully the Aspect computation.

– `Services` that describe the computation of an Aspect. Every Aspect must describe *begin* and *end* services in order to start and finish the execution of the aspect, respectively. Any other service performs the necessary computations of the aspect. When a service is being described the kind of behaviour of the service, in the context of the Aspect, must be specified, that is, it must be described if it is a provided (server behaviour), required (client behaviour) or provided/required (client/server behaviour) service. This is specified by preceding the service with the reserved words in, out and in/out respectively.

– `Interfaces` are used to describe the service-level visibility of the Aspect.

– `Preconditions`, `Valuations`, `Played_roles` and `Protocols` are in charge of describing the semantics of the Aspect services. A precondition describes what condition must be satisfied to perform a specific service. A Valuation specifies how the computation proceeds for a Service. A Played_role describes the orchestration process of the services of a specific Interface imported by the Aspect. Finally, a Protocol details the coordination process of the Aspect Services.

In order to describe each element, the Aspect provides a set of services, such as AddInterface, AddAttribute, etc, that allow its evolution.

While describing a SA several concerns can appear which crosscut across its definition, concretely, at the Aspect level. Concerns such as Distribution, Safety, Coordination, Functionality, etc, are clear examples in this sense. In order to describe to which concern an Aspect belongs to, an attribute called *concern* has been included in the Aspect metaclass.

4.3.3 PRISMA Architectural Elements

In PRISMA, there are three kinds of architectural elements: components, connectors, and systems. Because they share several commonalities, the abstract metaclass `ArchitecturalElement` has been included in the PRISMA metamodel.

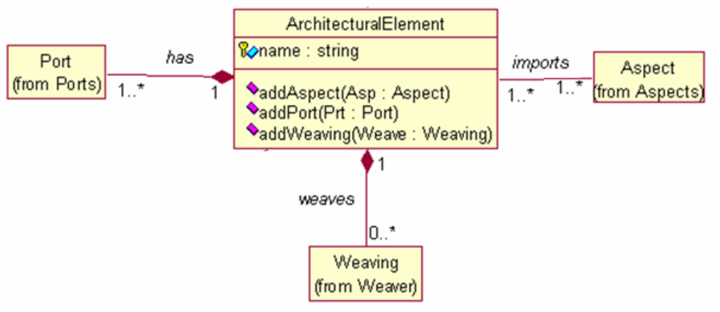


Figure 4-8 Describing Architectural elements (extracted from (Pérez, 2006))

Figure 4-8 depicts the main elements used while describing an ArchitecturalElement. Every ArchitecturalElement has at least a `Port` to describe the interaction of the ArchitecturalElement with its surroundings (see below PRISMA Port section). The Aspect has also an imports relation to specify which Aspects are to be imported for describing the computation of the ArchitecturalElement. How these imported Aspects are synchronized is described by means of `Weavings` relationships (see below Weaving section).

Figure 4-9 shows the hierarchy of architectural elements, where components and connectors are described by inheriting from Architectural element. Component and Connectors are described separately to distinguish the coordination role that Connectors must play when a SA is described. They are the only ones importing coordination aspects. In addition, each of them includes a service for the instantiation of these architectural elements.

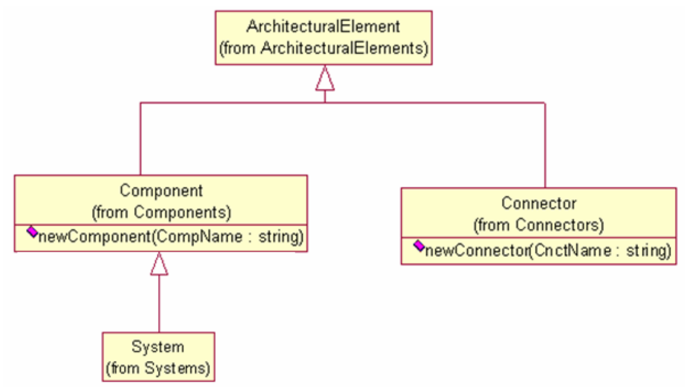


Figure 4-9 KindsOfArchitecturalElements of the PRISMA metamodel (extracted from (Pérez, 2006))

Figure 4-9 shows that another kind of ArchitecturalElement is `System`. Systems are used to describe complex Components, this is why they have been described by inheriting from the metaclass Component (see section 4.3.4 for more details).

PRISMA Port

As was stated above, Ports are in charge of describing how the interaction of the architectural element with its surrounding is. For this reason, when a Port is defined it is typed by an Interface to describe which services are public for the architectural element in that Port. In addition, its behaviour must be described as well by specifying which is its Played_Role (Figure 4-10).

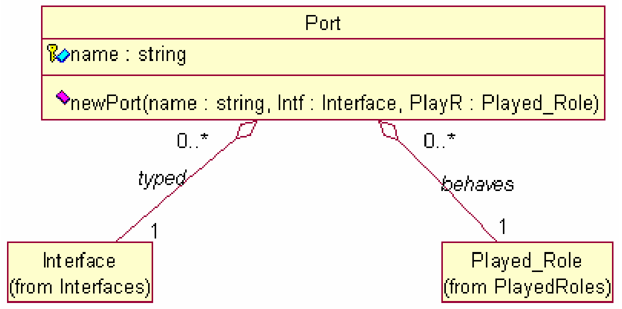


Figure 4-10 Describing PRISMA Ports (extracted from (Pérez, 2006))

PRISMA Played Role

It describes the behaviour associated to an Interface by establishing how its set of services can be executed. It means that it describes the execution process for one Interface, for this reason it inherits from Process. It should be taken into account that the services to be executed are defined by the Aspect which is using that Interface and is playing that Played_Role(see Figure 4-7).

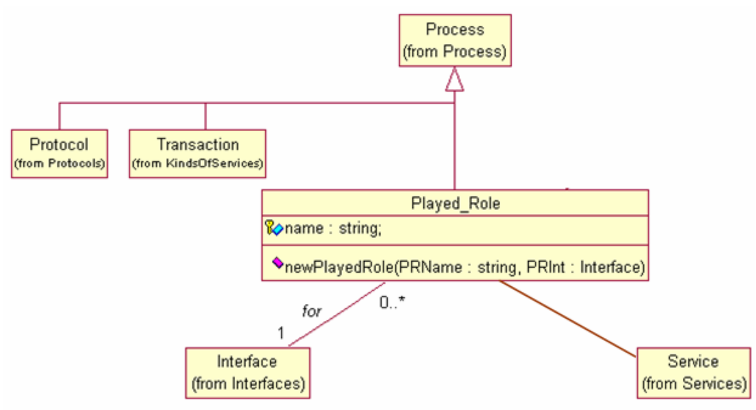


Figure 4-11 Describing PRISMA Played_Role (extracted from (Pérez, 2006))

PRISMA Weaving

Both Components and Connectors are internally described by importing Aspects and establishing the proper Weavings relationships. A Weaving relationship establishes a synchronization between two services described in two different Aspects. One of these services, which plays the role of the pointcut service, triggers the execution of the advice Service because the weaving establishes a causality relationship between them. The weaving methods that are typical of the AOP, and included in PRISMA, are the following:

– *after*: aspect1.service is executed after aspect2.service

– *before*: aspect1.service is executed before aspect2.service

– *instead*: aspect1.service is executed in place of aspect2.service

In addition, PRISMA extends the weaving operators with their respective conditionals:

– *afterif* (condition): aspect1.service is executed after aspect2.service if the condition is satisfied.

- *beforeif* (condition): aspect1.service is executed before aspect2.service if the condition is satisfied.
- *insteadif* (condition): aspect1.service is executed in place of aspect2.service if the condition is satisfied.

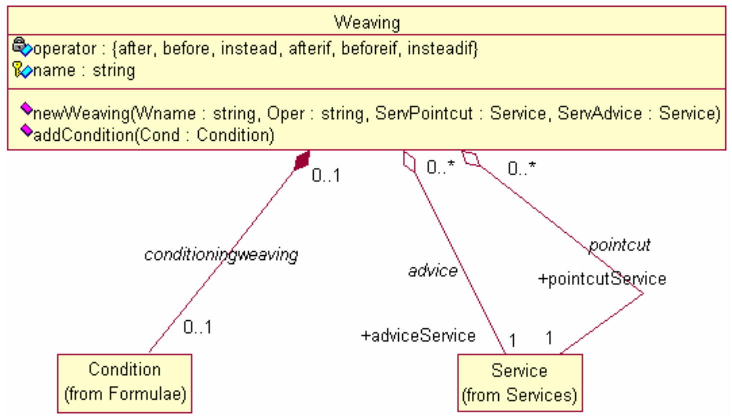

Weaving
operator : {after, before, instead, afterif, beforeif, insteadif}
name : string
newWeaving(Wname : string, Oper : string, ServPointcut : Service, ServAdvice : Service)
addCondition(Cond : Condition)
0..1
0..*
0..*
conditioningweaving
advice
pointcut
+pointcutService
+adviceService
0..1
1
1
Condition
(from Formulae)
Service
(from Services)

Figure 4-12 Describing the Weaving metaclass (extracted from (Pérez, 2006))

It is worth noting that Weaving relationships are described in the architectural element, not in the Aspect. Since, Aspects are specified in an independent way of the architectural elements that will use them. This is one of the main strengths of PRISMA because it facilitates not only the reuse of Aspects in different architectural elements, but also that architectural elements can change dynamically its behaviour by changing its weaving relationships.

4.3.4 PRISMA Systems

As was stated above, a System is a complex Component. Figure 4-9 shows that the PRISMA System metaclass is described by inheriting from Component. This means that when a System is specified it can import and weave Aspects for its description and it has Ports to describe its interaction with its surroundings.

However, more elements emerge during its description as Figure 4-13 depicts. A PRISMA System has a composition relation of a set of architectural elements connected by means of `Attachments` (see below 4.3.5 section). Whenever the services of a Component in a PRISMA System must be published externally to the System a `Binding` relationship is established between that architectural element and a System Port (see below PRISMA Attachments section).

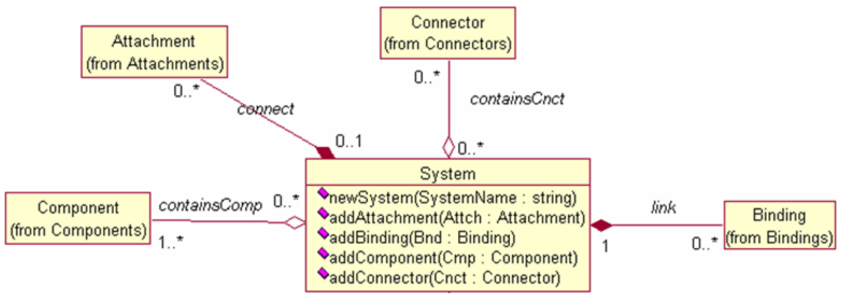


Figure 4-13 Describing System metaclass (extracted from (Pérez, 2006))

PRISMA Bindings

PRISMA Bindings are used to describe when the port of a Component in a System is connected to a System Port (Figure 4-14). In addition, when a Binding is being specified the cardinality on each end must be described. Minimum and maximum cardinalities on each end constraint how many instances of a specific Binding can be connected to a Port of a composing Component and a Port of the composed System.

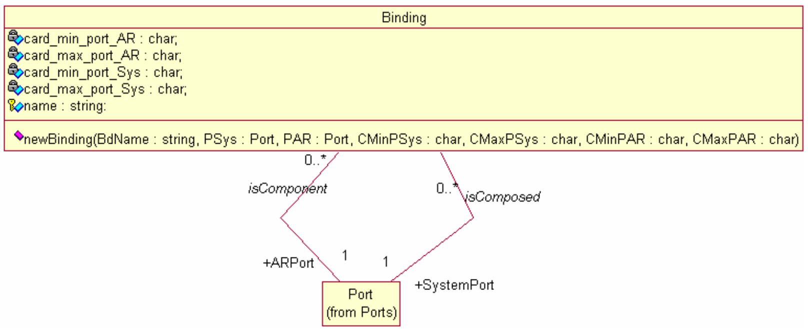

Binding
card_min_port_AR : char;
card_max_port_AR : char;
card_min_port_Sys : char;
card_max_port_Sys : char;
name : string;
newBinding(BdName : string, PSys : Port, PAR : Port, CMinPSys : char, CMaxPSys : char, CMinPAR : char, CMaxPAR : char)
0..*
isComponent
0..*
isComposed
+ARPort
1
1
+SystemPort
Port
(from Ports)

Figure 4-14 *Bindings* of the PRISMA metamodel (extracted from (Pérez, 2006))

4.3.5 PRISMA Attachments

An Attachment is employed to describe a communication channel between the port of a component and the port of a connector (Figure 4-15). This is because PRISMA intends a low coupling between Components, so they cannot directly connect but by means of a Connector. In addition, a minimum and maximum cardinality can also be described on each end so that the number of instances of that Attachment that can be connected to one instance of a Component and a Connector can be constrained.

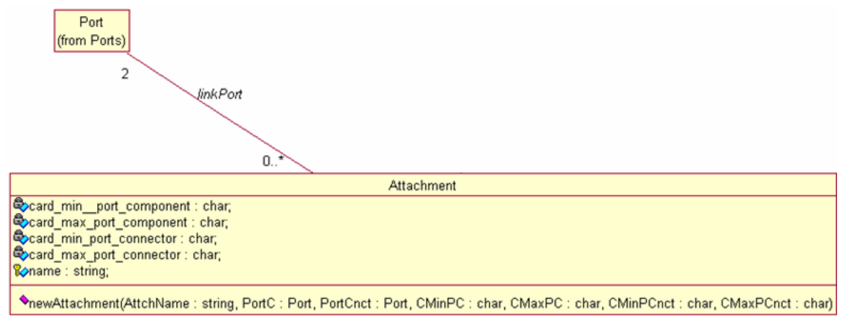

Port
(from Ports)
2
linkPort
0..*
Attachment
card_min__port_component : char;
card_max_port_component : char;
card_min_port_connector : char;
card_max_port_connector : char;
name : string;
newAttachment(AttchName : string, PortC : Port, PortCnct : Port, CMinPC : char, CMaxPC : char, CMinPCnct : char, CMaxPCnct : char)

Figure 4-15 Describing Attachment metaclass (extracted from (Pérez, 2006))

4.3.6 Instantiating a PRISMA model

In the previous sections, the collection of types defined by PRISMA has been presented. These types are available to the architect for the description of the architectural model. In order to carry out such a specification a metaclass *PRISMAArchitecture* has also been defined in the PRISMA metamodel (Figure 4-16). In this sense, the architectural model is defined in terms of the Components it includes, the Connectors it synchronizes, the Interfaces it uses, the Aspects it imports, and the Attachments it connects. With this aim, a set of services has been defined in PRISMAArchitecture to allow the addition of the previous elements. In addition, a PRISMAArchitecture is identified by means of its name.

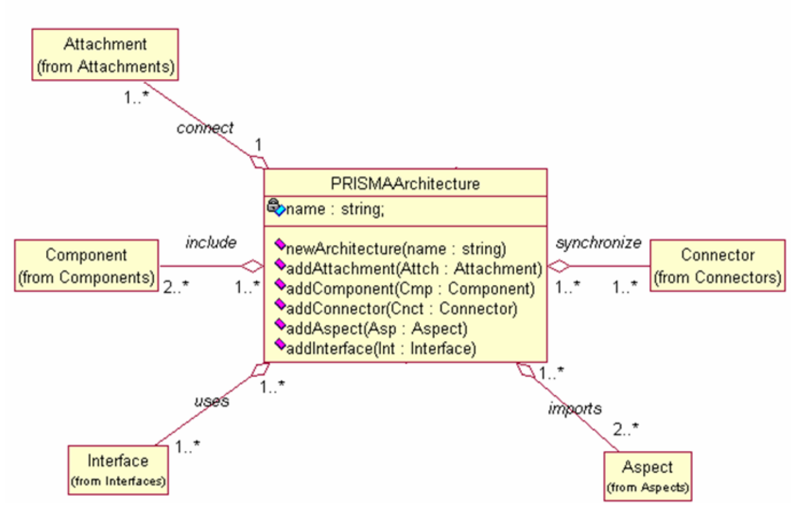


Figure 4-16 Describing a PRISMA architecture (extracted from (Pérez, 2006))

"The real voyage of discovery consists not in seeking new landscapes, but in having new eyes. "— Marcel Proust

CHAPTER 5

Goals: why the system will be

5.1 INTRODUCTION

IEEE 830-1998 (IEEE, 1998) recommendations are the milestone concerning the contents that are mandatory in a Software Requirements Specification (SRS). This standard involves a number of challenges related to quality characteristics that must be achieved by any SRS, such as correctness, unambiguousness, completeness, and consistency, ranked for importance and/or stability, verifiability, modifiability, and traceability. Regardless of the followed approach used for requirements specification, it must always be compliant with these requirements.

Requirements organization and presentation are crucial to facilitate their maintenance and ensure the quality characteristics stated above. In this sense, the IEEE 830-1998 describes different alternatives for the organization of the SRS that are based on "operation modes of the system", "user classes", "objects", "characteristics", "stimuli", "responses" or "functional hierarchies". However, in practice, when the number of requirements and/or the level of complexity (due to the high number of relationships between them) are significant, those recommendations are not enough. Finally, IEEE 830-1998 does not provide any assistance to the process of elaboration or analysis of the requirements.

Along the requirements elicitation and specification process, several stakeholders are involved (both from the customer and technical side) every one with their own interests and views of the system, which ought to be rightly represented and reconciled in the SRS. This means that the followed approach should appropriately address this issue.

A clear example of complexity can be found in the EFTCoR project (see section 4.2.1). This project exhibits several specific needs in terms of requirements specification, such as, the variability inherent in the family of robots to be handled; the high incidence of non-functional requirements (reliability, performance, safety, etc) which crosscut functional ones; the need to evaluate alternative designs meeting system requirements; and, finally, a large specification where an appropriate organization is unavoidable.

Taking into account these needs, a requirements model had to be defined. The first obstacle to overcome was to select what the best approach was. As was introduced in section 2.2.1, the Goal-Oriented approach exhibits several advantages that make it appropriate for our purposes as its capability for traceability and its ability for reasoning. However, it does not provide any help for dealing with the aspect-oriented approach nor with an effective management of variability (see section 2.2.6). This implied that this approach had to be adapted to deal with such concerns.

In addition, as was described in the chapter 1, an Action-Research methodology was applied throughout the definition of ATRIUM. This meant that change was an ever-present issue during this process. Practitioners were changing their needs of expressiveness very frequently, with their related drawbacks. This made emerge a metamodelling approach as a way to deal with such a wide diversity of terms and concepts and to manage properly the unavoidable change. This proposal is presented in section 5.2.

How the metamodelling proposal was put into practice for the ATRIUM Goal Model definition is described in 5.3. The process that has been defined for the construction of an ATRIUM Goal Model is introduced in section 5.4. Section 5.4.2 presents one of the main strengths of the Goal Models and how it has been accomplished in ATRIUM: reasoning.

5.2 A PROPOSAL FOR CUSTOMIZING RE METAMODELS

We must, firstly, point out that this proposal was defined in a wider context, i.e., not only goal-oriented and aspect oriented approaches were considered during its definition. On the contrary, most of the approaches described in section 2.2 and their needs of expressiveness were used during the process. Specifically, the Goal-Oriented, Aspect-Oriented, variability management and Use Cases were used for this work.

The first obstacle to overcome when integrating these approaches and their notations is the wide diversity of terms and concepts, with many overlaps among them. We have realized that a consensus would be very difficult if we had attempted to define a global notation so as to cope with the whole included expressiveness. Therefore, we have decided to organize our work in two parts. The first part, described in section 5.2.1, defines a metamodel for the essential concepts that allow us to deal with the generic expressiveness. By doing so, we could get a consensus more easily. The second part (see section 5.2.2) describes a process which specifies how this core metamodel can be tailored according to the specific needs of expressiveness.

5.2.1 A Metamodel for Requirement Specification

The core concepts and their relationships, which are taken from the described approaches, are shown in Figure 5-1. `Artifacts` are the essential concept in a requirement specification; they represent any kind of specification at a certain level of granularity. An *Artifact* can be a complete SRS document, a section in a document, a piece of text representing a requirement, etc. In addition to the artifacts, it is also necessary to establish relationships among artifacts of the SRS. Therefore, we have identified several kinds of relationships that are described below.

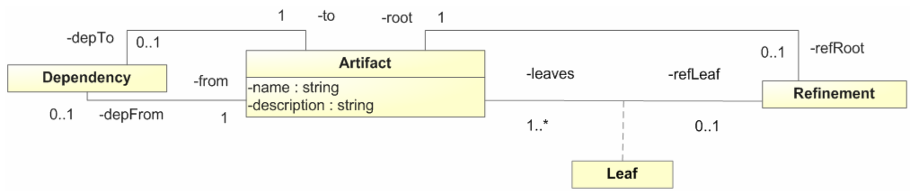


Figure 5-1 Metamodel for the Core Concepts.

Table 5-1 OCL Constraint for Refinement relationship

Relation	**Constraint**
Refinement	**context** Refinement **inv** Self.leaves->forAll(a:Artifact | a <> Self.root)

An artifact can be refined through other artifacts, forming a hierarchical structure. This has leaded us to introduce the `Refinement` relationship. This kind of relationship would allow the analyst to define hierarchy decompositions, refinements relationships, etc. In addition, it can be observed in the Table 5-1 that cycles are not allowed while defining a refinement relationship. This is especially meaningful for the propagation process that is introduced in section 5.4.2.

It can be observed that Refinements are related to Artifacts by means of an association class. It allows the analyst to attribute the relation between the leaves artifacts and the refinement relationship. We have observed this need after studying several proposals and it facilitates, for instance, that rationales to describe the relation are included.

In addition to this refinement relationship between artifacts, the dependency relationship has also been included in the metamodel. Perhaps this is the most conflictive relationship for consensus. As (Robinson et al., 2003) have described, this relationship is critical for what is known as `Requirements Interaction Management` (RIM):

> *"The set of activities directed toward the discovery, management, and disposition of critical relationships among sets of requirements."*

RIM is a crucial activity to obtain a proper requirements specification. In this work, an extensive survey of the kind of dependency relationships has been described along with different techniques and tools for its exploitation. It illustrates the large set of semantics interpretations that the dependency relationship has in the RE field. For this reason, we have preferred to represent it in its most generic form, i.e., by means of `Dependency` which is applicable to artifacts in the core concepts. Several kinds of dependency relationships are allowed between artifacts of different hierarchies. For example, non-functional requirements constrain functional requirements or goals, data are used for requirements, actors interact with use cases, variants require another variants, etc. Some kinds of dependencies between artifacts are bidirectional, such as those used to specify conflicting goals or mutual exclusion between variants. However, others are unidirectional, such as *Extension* dependencies between Use Cases or *Requires* dependencies between variants. Therefore, although, in general terms, we consider that dependencies are unidirectional, the Metamodel also permits the specification of dependencies that in some cases may imply their inverse. Table 5-2 describes a constraint related to the Dependency relation because it does not make any sense that a reflexive dependency could be established in a requirements model.

Table 5-2 OCL Constraint for Refinement relationship

Relation	**Constraint**
Dependency	**context** Dependency **inv** Self.to <> Self.from

Table 5-3 Mapping concepts: from concepts of Approaches to RE (rows) to our Metamodel (columns)

Approaches to RE		Proposed Metamodel		
	Concept	Artifact	Dependency	Refinement
Use Case	Use Case	Extending		
	Generalization			
	Communication			
	Include		Extending	
	Extend		Extending	
Goal-Oriented	Goal	Extending		
	Agent	Extending		
	Refinement Relationship			
	AND			Extending
	OR/XOR			Extending
Variability Management	Variant	An artifact with a Variability or relationship		
	Variation point			Extending
	Time link			Using an attribute on relation Variation
	Multiplicity			Using an attribute on relation Variation
	Types of Variability			Using an attribute on relation Variation
	Require		Extending	
	Exclude		Extending	
AORE	Concern	Extending		
	Crosscutting		Extending	

Table 5-3 summarizes the main concepts of each studied approach and how the mapping between them and those described in the proposed Metamodel (Figure 5-1) was established. We have to point out that some concepts do not appear in the core Metamodel but they are described by extending it according to the process described in section 5.2.2.

5.2.2 A process for customizing the core

Once we have described the metamodel, it has to be tailored according to the specific needs of expressiveness. With this purpose, we suggest the following steps to adapt and/or extend the metamodel, which must not necessarily be applied sequentially, but following the analyst preferences:

I. To define the types of artifacts which are relevant to the model. This permits the inclusion of the needed artifacts by means of specialization. In this sense, whenever a new artifact has to be defined a specialization is specified using as base class *Artifact* or another artifact already defined. The child artifact inherits any attribute that had been defined by its parent.

II. To establish the refinement relationships of interest. The essential metamodel provides the relationship Refinement. If necessary, any additional refinement type could be added as a specialization of any of the existing ones or as a new and independent one.

III. To establish the types of dependency relationships between artifacts. This allows the definition of the relevant relationships between artifacts from the metaclass *Dependency* or any other already defined by means of specialization.

IV. To include the attributes considered relevant to the types of artifacts, types of refinements and types of dependencies. It is necessary to bear in mind that, whenever a new kind of artifact, refinement relationship or dependency relationship is defined, all the attributes defined by its parent are inherited so that their semantics should be considered to avoid inconsistencies. Regarding *Refinement* relationships, the relation *Leaf* could be extended if it is necessary to describe any attribute in the link between the leaves *Artifacts* and *Refinement*.

V. To formalize artifacts and relationships. Those constraints to be applicable on artifacts and relationships have to be described by means of OCL (OCL Specification, 2005) (Object Constraint Language). This language has been selected for this purpose because it is a well-known and extended proposal. In addition, there are a wide set of tools (OCL tools, 2005) that allow for simple consistency checks and type checking in terms of defined OCL constraints.

As it can be observed in Figure 5-1, there is not a direct connection between the described relationships and the artifacts on which they have to be applied, but artifacts and relationships are specified independently. This alternative provides us with an improved readability and comprehensibility of the Metamodel. However, it means that whenever a new relationship is defined, the analyst has to constrain the set of artifacts on which the relationship can be applied by means of OCL following the step (V).

5.3 Describing the ATRIUM Goal Model

Section 3.3.1 introduces why a Goal Model was selected as the main approach for ATRIUM requirements specification, specifically, for its ability for back-tracing low-level details and reasoning about alternatives. However, there is no standard proposal to follow but several proposals have been introduced up to date, such as GBRAM, KAOS, NFR-Framework, etc (see section 3.2 for more details about them). This means that they were considered instead of describing our own approach from scratch, taking into account that they must provide support to both:

– `Functional requirements.` They describe services that the software provides, i.e., the transformations the system performs on the inputs.

– `Non-Functional requirements.` They describe conditions or constraints that the software must satisfy. They refer to how the services are provided, for instance, in terms of performance, adaptation, security, etc. We are highlighting them because they are especially meaningful in terms of software quality.

Among these proposals, KAOS and NFR Framework have emerged as the ones with greater impact in research and practice. KAOS provides a detailed framework for dealing with functional requirements, where verification capabilities can be exploited thanks to the temporal logic (Manna & Pnueli, 1992) used to formalize the goals. For this reason, how a Goal is specified in this proposal has been adopted in ATRIUM. In addition, this proposal has a meaningful help because a catalogue of refinement patterns have been defined in (Letier & Lamsweerde, 2002) which are first steps towards the automatic generation of operationalizations for those formalized goals.

However, it does not provide detailed enough mechanisms for dealing with non-functional requirements. This deficiency exhibited by KAOS has been overcome by the introduction of some concepts proposed by the NFR Framework. This proposal has another advantage that makes it especially suitable for its use in ATRIUM: it has several mechanisms for helping in the analysis of alternatives, which constitutes the main aim of ATRIUM. In the following sections, it is presented how each concept has been adopted.

In addition, it must be considered that ATRIUM is intended to follow an Aspect Oriented Software Development strategy. This means that aspects must be detected and traced from the early stages to implementation. For this reason, mechanisms for defining concepts of AORE had to be described. The Goal Model obtained from the integration of KAOS and NFR Framework was extended by integrating the aspect-oriented approach, in order to achieve both the efficient management of the crosscutting and the correct organization of the SRS.

Moreover, some special needs are demanded for the EFTCoR case study as was presented in 4.2.1. That is, EFTCoR is oriented to the description of a family of tele-operated systems, with common services and elements for several systems and different and specific ones for each one of them. For this reason, mechanisms for the variability management had to be introduced in order to apply the ATRIUM Goal Model to the definition of product lines.

Finally, we detected from the early use of the proposal, that practitioners faced problems for the application of the described Goal Model. Mainly, because they did not know what the appropriate point to start the specification was. This motivated that an initial framework for the selection of concerns was established as well, that is based on the standard ISO/IEC 9126 (ISO/IEC 9126). It also provides us with a strategy for the separation of concerns.

All these topics make emerge the ATRIUM Goal Model as an integrated proposal. Figure 5-2 shows how each approach provides us with a different view of the SRS along with the more meaningful concepts each one has provided us with.

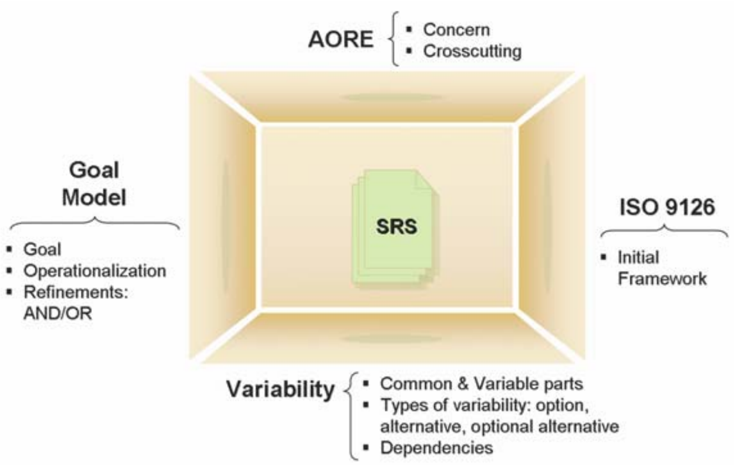


Figure 5-2 An integrated proposal for the Goal Model of ATRIUM

Taking into account these approaches the metamodel of the ATRIUM Goal Model shown in the Figure 5-3 was described. The artifacts and the identified relationships shown in the figure come from the application of steps described in section 5.2.2 extending the core metamodel of Figure 5-3. The building blocks of the Goal Model that have been defined applying the step (I) are described in section 5.3.1. These blocks are related to each other using a set of relationships that are described in section 5.3.2. These relationships have been described by means of the steps (II) and (III) of the metamodelling process.

The integrity constraints of the whole model when step (V) is executed are applied in addition to those determined by associations and multiplicities defined in the metamodel. In the ATRIUM Goal Model, these constraints refer to the types of artifacts that can be related by means of a specific kind of dependency or refinement. For this reason, while defining each relationship, its related constraints are also specified.

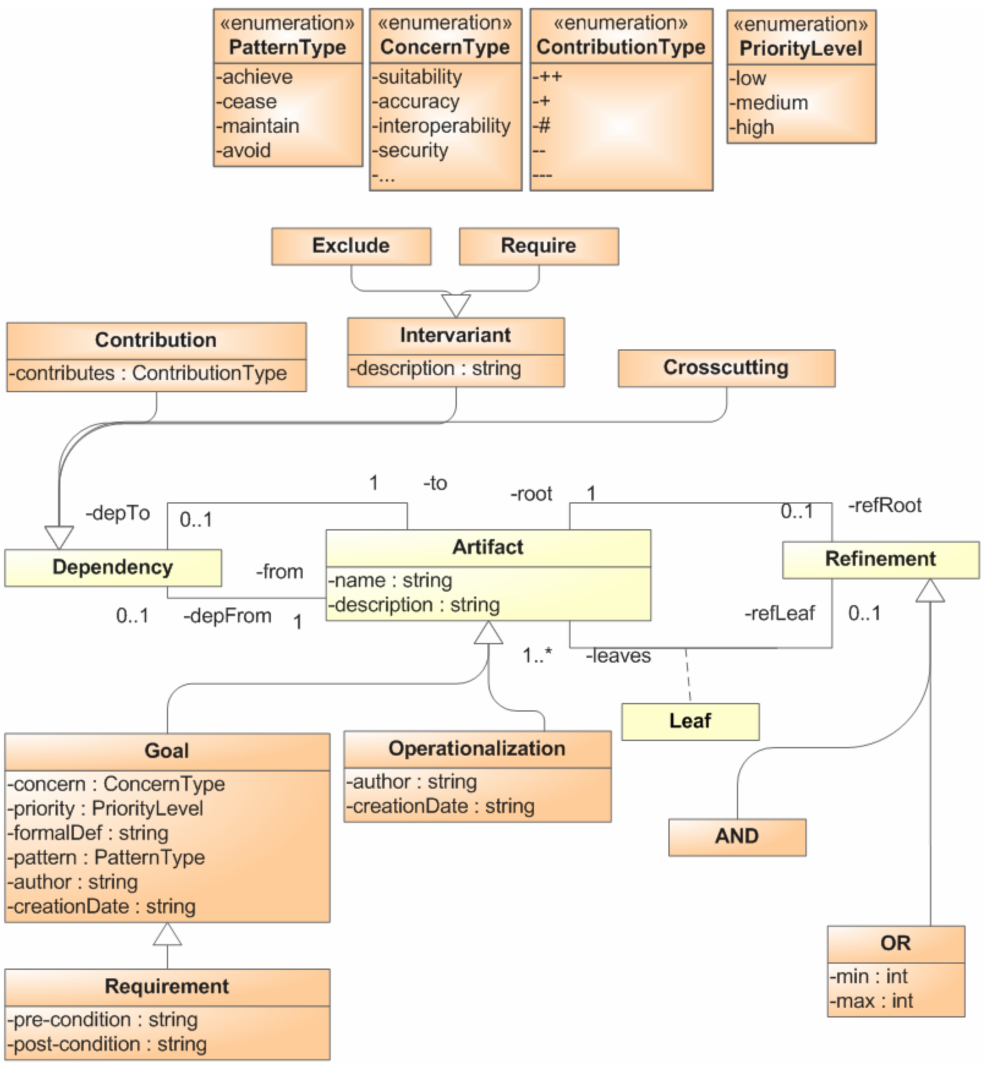


Figure 5-3 Metamodel of the ATRIUM Goal Model

5.3.1 Building Blocks for the Goals Model

As was stated above, the ATRIUM Goal Model provides a number of abstractions in terms of which constraints on the software system have to be defined. Mainly goals, requirements and operationalizations are the building blocks used during the ATRIUM Goal model construction, which are presented in the following sections.

Defining goals

A key element introduced in the model is a `goal`. It is defined as an objective that the system-to-be should achieve (Lamsweerde, 2001a), i.e., a constraint or obligation that the system should meet. As can be observed in Figure 5-3, it was described by extending from Artifact, which means dependency and refinement relationships can be applied on them. Due to this inheritance relationship a goal also has in its definition attributes for its *name*, which helps to identify it, and *description* to explain using natural language which intention it prescribes. These two attributes can be observed in the textual description of a goal:

Table 5-4 BNF for textually describing a goal

```
GOAL
NAME <identifier>                      — unique identifier of the goal
DESCRIPTION <expression>               — textual description of the goal intention
[FORMALDEF <expression>]               — formal definition of the goal
PATTERN achieve | cease | maintain | avoid | optimize
                                              —pattern of description of the goal
CONCERN suitability | accuracy | interoperability | security | ...
                                              —concern to classify the goal
PRIORITY High | Normal| Low            — priority of the goal
AUTHOR <expression>                    — who first describes the goal
CREATIONDATE <expression> — when the goal was specified
```

Table 5-4 depicts that the formal definition of the goal can also be introduced when it is defined. As was stated above, the temporal logic is used to accomplish such a definition following the recommendations of KAOS. By adopting such an approach, not only the capabilities provided by KAOS for the verification of the specification can be exploited but also the available tool called FAUST (Ponsard et al., 2004). This temporal logic uses some operators to formalize the definitions that are presented in the Table 5-5 along with their description.

Table 5-5 Describing the temporal logic operators

Operator	Description
∘	In the next state
•	In the previous state
◊	Sometime in the future
❑	Always in the future
W	Always in the future unless
U	Always in the future until
A⇒C	In every future state A implies C

@P	P holds in the current state and not in the previous state, i.e., $\bullet \neg P \wedge P$
$\Diamond_{\leq ku} P$	P holds in some future state within k times units u
$\square_{\leq d} P$	P holds in every future state up to some deadline d

Table 5-4 depicts that the property *concern* must also be described. It describes the type of need or expectation it refers to, i.e., suitability, accuracy, interoperability, security, etc. It is similarly described in KAOS by means of the property category. This property is specially significant for the analysis process as described in the section 5.4.2.

In addition, while describing a goal a *pattern* can also be used that specifies the required temporal behavior of the goal. With this aim, KAOS distinguishes the following four goal patterns (Letier, 2001):

– *Achieve*: this pattern specifies that the goal requires a property to be eventually hold.

– *Cease*: this pattern specifies that the goal requires a property to be eventually stopped.

– *Maintain*: this pattern specifies that the goal requires a property to be always hold.

– *Avoid*: this pattern specifies that the goal requires a property to be never hold.

Additionally, other aspects have to be stated when a goal is defined. Each goal has to be classified according to its *priority*, from *high* to *low*, for the system-to-be. This classification helps the analyst to focus on the important issues. These priorities can arise from several factors: organizational ones when they are critical to the success of the development, constraints on the development resources, etc. In addition to these properties, *author* and *creation* are also described to facilitate the trace of the goals. An example of a goal for the EFTCoR is described in the following:

Figure 5-4 Example of a Goal description

GOAL
NAME approachToTheArea
DESCRIPTION robot is close to the area for the treatment
FORMALDEF (∀r : EFTCoR, ∀s: Ship) move® ⇒ ◊ close(r,s)
PATTERN achieve
CONCERN suitability
PRIORITY High
AUTHOR Elena Navarro
CREATIONDATE 01/10/06

It should be highlighted that we are not distinguishing between functional and non-functional requirements as KAOS does using goals and softgoals. Even some works, such as (Bass et al., 2001), state that the use of the term softgoal for quality requirement is not appropriate. This is because they can be specified clearly and a process can be defined to determine whether they can be satisfied. We do not think this distinction can make a significant contribution to the SRS but it is more relevant the priority that concerns have in the system-to-be. This idea has been described in several proposals such as (Moreira et al., 2005), where the use of an appropriate taxonomy of concerns and the priority of the goals is more significant especially in domains where security, safety, performance, etc, are highly demanded.

Defining Requirements

Similarly to goals, other used elements for the Goals Model construction are the `requirements`. They also specify a need or constraint on the system-to-be, although its main difference with respect to goals is its capability to be operationalized, i.e., to be assigned to and realized by a set of agents. This constitutes the main difference between a goal and a requirement. The fact that a requirement can be made operational means that it can be verifiable in the system to be.

While defining a requirement, some more details may be provided in addition to those described for goals. These are a set of `preconditions` and `postconditions`. Preconditions establish which conditions must be hold before some operation is performed. Postconditions define the conditions that have to be satisfied after some operation is performed. Their evaluations help us to determine the best design alternatives among those that satisfy the post-conditions for the established goals. The temporal logic, similarly to goals, is used for their description. This means that the textual notation for a requirement is described in the following table.

Table 5-6 Describing a Requirement

```
REQUIREMENT
NAME <identifier>              — unique identifier of the goal
DESCRIPTION <expression>       — textual description of the goal intention
[FORMALDEF <expression>]       — formal definition of the goal
PATTERN achieve | cease | maintain | avoid | optimize
                                             — pattern of description of the goal
CONCERN suitability | accuracy | interoperability | security | ...
                                             — concern to classify the goal
PRIORITY  High | Normal| Low          — priority of the goal
AUTHOR <expression>                   — who first describes the goal
CREATIONDATE  <expression>— when the goal was specified
[PRE-CONDITION] <expression>
```

```
                                    — condition to be hold before the requirement is
met
[POST-CONDITION] <expression>       — when the goal was specified
                                    — condition to be hold after the requirement is
met
AUTHOR <expression>
CREATIONDATE  <expression>
```

Considering how a requirement is specified, Table 5-7 presents an example of a requirement of the EFTCOR system. We can appreciate that there is no specification for pre and post-condition because they are considered optional when a requirement is defined. It must be included if a verification process is going to be accomplished but this is currently out of the scope of ATRIUM. We should remind that this process has been borrowed from the KAOS proposal.

Table 5-7 Example of a requirement

```
REQUIREMENT
NAME moveArm
DESCRIPTION in order to close the arm of the robot to the area to be cleaned it has to
be moved to the indicated coordinates.
PATTERN achieve
CONCERN suitability
PRIORITY  High
AUTHOR Elena Navarro
CREATIONDATE  01/10/06
```

KAOS also employs requirements during the construction of its Goal Model. However, they make a distinction between requirements and `assumptions`. This is because each one distinguishes a different assignment of responsibility. That is, when the responsibility of its achievement is assigned to an environmental agent they speak about assumptions. However, when this responsibility is to be assigned to the system-to-be, they employ requirements. ATRIUM does not make this distinction to provide more flexibility to the specification. As shown in the following, it is at the operationalization level when this distinction is accomplished, that is, at this level it is decided who is in charge of performing such a requirement. In this way, we are following the (Lauesen, 2003)'s recommendations because premature decisions would limit our ability to define different systems depending on time, costs, available resources, etc.

Defining operationalizations

Aside from goals and requirements, another building block for the Goal Model is the `Operationalization`. When an analyst has refined the initial set of goals, he/she must offer a set of solutions that allow the system to achieve the established goals. An operationalization is a solution, i.e., an architectural design choice for the system-to-be to meet the users' needs and expectations. They are called operationalizations because they describe the operation of the system, i.e., the system behaviour, to meet the requirements either functional or non-functional. Operationalizations are textually described as Table 5-8 indicates:

Table 5-8 Describing operationalizations

```
OPERATIONALIZATION
NAME <identifier>                    — unique identifier of the operationalization
DESCRIPTION <expression>             — textual description of the operationalization
                                               intention
AUTHOR <expression>                  — who describes the operationalization
CREATIONDATE <expression> — when the operationalization is described
```

It can be noticed that there is only a section in the textual notation to define briefly the alternative solutions for satisfying a given requirement. It is in the Scenario Model (chapter 7) where these solutions are expressed in a detailed form. However, operationalizations are introduced in the Goal Model to represent conceptually each solution, so that relationships among the different alternatives can be established within the Goal Model. Operationalizations establish a coupling between the Scenario Model and the Goal Model, and traceability between operationalizations and a specific view of the Scenario Model is established.

We can observe, in the textual notation, that operationalizations are not characterized with the concern they are related to. This is because the same solution can be associated to different requirements. This implies that operationalizations can be classified according to different concerns.

We must emphasize that this concept has been partially borrowed from the NFR-Framework. It also describes partial solutions for the system-to-be by means of its use. However, it does not make a special distinction about which kind of solution operationalizations can describe. Instead, it is used to describe any kind of alternative that should be analyzed for the design of the system. Examples of these alternatives are the kind of ordering algorithms, kinds of accounts to be used in a system, etc. In ATRIUM, they are specifically used to describe architectural solutions for the system because it aims at obtaining the proto-architecture of the system-to-be. On the other hand, operationalizations in KAOS are not buildings blocks but relations that are employed to link a requirement or assumption with the operations to be performed by the system or the environment. It is when operations are defined when pre-conditions and post-conditions are described in a similar way as we describe requirements.

5.3.2 Relationships: An Element in the Refinement Process

The stated building blocks, goals, requirements and operationalizations, are inter-related by means of a set of *relationships*. They are in charge of gluing the different elements to complete the model and enhance its cohesion. Moreover, their relevance is not only restricted to this gluing but also they allow the analyst to introduce the rationale of the system design. In order to describe each identified relationship they are introduced according to the applied step of the process described in section 5.2.2. That is, firstly, the established refinement relationships are introduced and then dependency relationships are described. In addition, the necessary constraints are introduced along with each relationship.

Describing refinement relationships

When a Goal Model is being built, an intentional refinement is performed. It describes how a goal can be reduced into a set of subgoals/requirements via *AND/OR* relationships. These building blocks and relationships are structured as an acyclic graph, where the refinement is achieved along the structure, from the higher to the lower level, by applying intentional refinements.

Every goal, which is too coarse-grained to be operationalized, is refined into a set of subgoals that are a decomposition of the original one. In this way, an AND relationship between a goal *GoalX* and a set of sub-goals G_1, …, G_N or requirements R_1, …, R_N is established if the whole set of sub-goals has to be satisfied in order to satisfy *GoalX*. We say that a goal is satisfied if the system-to-be is able to achieve it by means of the behaviour it describes. The textual description and its related OCL constraint are introduced in Table 5-9 and Table 5-10, respectively. It can be observed that only goals can play the role of root when this relationship is used. This is because requirements are employed in the last step of the refinement process when a seamless transition from intentional to operational refinement is going to be performed.

Table 5-9 Describing AND relationships

AND <identifier>	— unique identifier of the refinement, it is internally specified
ROOT <identifier>	— identifier of the goal or requirement which plays the role of root
LEAVES <identifier> {<identifier>}	— identifier of the goals or requirement which plays the role of leaves

Table 5-10 OCL Constraints for refinement relationships

Relation	Constraint
AND	context AND inv Self.root. oclIsTypeOf (Goal) **and** Self.leaves->forAll(a:Artifact | a.oclIsKindOf (Requirement))

During this refinement process, it is also possible that the set of subgoals are not mandatory to be met by the system-to-be but only some of them are necessary. This set of subgoals would be considered alternatives in the refinement process. For this reason, this relationship is key in the process of alternative analysis. In this context is where the relationship OR is employed. A goal *GoalX* is related to a set of sub-goals G_1, …, G_N via an *OR* relationship when *GoalX* is satisfied if at least one sub-goal or requirement is satisfied. It is textually described as:

Table 5-11 Describing OR relationships

OR <identifier>	— unique identifier of the refinement, it is internally specified
ROOT <identifier>	— identifier of the goal or requirement which plays the role of root
LEAVES <identifier> {<identifier>}	— identifier of the goals or requirement which plays the role of leaves
MIN <number>	— minimum number of variants to select
MAX <number>	— maximum number of variants to select

Table 5-12 OCL Constraints for OR relationship

Relation	Constraint
OR	context OR inv Self.root.oclIsKindOf(Goal) and Self.leaves->forAll(a:Artifact | a.oclIsKindOf (Requirement)) and Self.max >= Self.min and self.max >= Self.leaves→size()
	context OR:: min: Integer **init** 1
	context OR:: max: Integer **init** leaves→size()

We should draw your attention to some peculiarities that make the OR relationship different from the AND. As can be observed, the OR relationship is inherently specifying a variability in the specification because it describes alternative goals or requirements that can be met or not for the final system.

This means that this relationship can be used to manage the variability as was presented in section 2.2.6, that is, either for dynamic architectures or product lines. For this reason, whenever an OR relationship is described, it must be understood that the root goal is going to play the role of a *variation point*, i.e., the point where the variability must be resolved. Moreover, each leaf of the relationship is described as a *variant* for the system-to-be, i.e., an alternative description of the system-to-be.

It can be noticed that it not only identifies which are the goals/requirements root and leaves but also other unusual attributes in the Goal-Oriented Approach. They are *min* and *max*. As was introduced in section 2.2.6, there are some needs in terms of expressiveness that are mandatory when dealing with the variability management. Specifically, the notation must provide means to identify when and how the variability must be resolved, i.e., when it must be decided which variants are to be introduced in the final product or dynamic architecture. Thus, these attributes are going to be established according to the following rules:

- Leaves goals and requirements are going to be considered as variants. This means that they are an intention or need that can be present or not in the final system, i.e., they can be met or not by the final product or dynamic architecture.

- According to (Trigaux & Heymans, 2003), it is mandatory to specify when the variability is resolved, i.e., when it is decided which variants are to be present in the system-to-be. In the ATRIUM Goal Model, when an OR relationship is specified, the variability has to be resolved in *specification Time*, i.e., it must be decided which needs or expectations must be satisfied by the system-to-be. This could mean a limitation because it does not specify the variability at run-time, which is so necessary to describe dynamic architectures. However, as it is described in section 5.4.1, ISO/IEC 9126 (ISO/IEC 9126) quality standard is used as a framework to identify the concerns of the system-to-be. In this standard, the quality characteristic called `Adaptability` is described as:

  The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for the software considered.

  This means that any requirement or goal that describes such a kind of capability will be described as a requirement/goal of adaptability. Some examples of this use are presented in the case study in chapter 6. In addition, it must be highlighted that one of the advantages of using OR relationships is that they can be used as evidences to detect the adaptability necessary for the system-to-be.

– *Min and max* speak about how many variants must be selected when the variability is resolved. On the one hand, *min* is assigned to a value in order to establish how many variants must be selected as a minimum. On the other hand, *max* determines the maximum number of variants that can be present when the variability is resolved. The number of variants obtained by refinement is the value by default for *max*. These constraints have been introduced in the Table 5-12.

The Goal-Oriented Approach has been recently applied to modelling and analyzing variability. Meaningful works in this area are those presented by (Gonzales-Baixauli et al., 2004) and (Hui et al., 2003). However, these proposals have some deficiencies from the point of view of the required expressiveness for variability; for example, they lack the concept of multiplicity or specific dependencies between variants, which are very important when the analysis activity and the product derivation are performed. However, they do provide sophisticated mechanism for analyzing alternatives that have been used to select the optimum variant for each product by (Gonzales-Baixauli et al., 2004) or for each profile/ability of the user by (Hui et al., 2003). The idea of including such capability in ATRIUM was to cover the needs exhibited by the EFTCoR project, overcoming the limitations exhibited by those approaches in terms of expressiveness.

Describing dependency relationships

We have defined a complete and disjoint hierarchy of types of dependencies by applying step (III) that can be observed in Figure 5-3. They were identified to deal with the expressiveness needs previously stated in terms of variability along with the aspect-oriented and goal-oriented approaches. They are described in the following:

– It is frequently the case that while defining a product line or a dynamic architecture some binary relationships must be established between the requirements. They are in charge of describing which combination of variants is allowed or not for some products or dynamic architectures.

  This has motivated the inclusion of the `Intervariant` dependency in the Metamodel. This has been extended to express properly the two kinds of possible relations among variants that are more widely used: `Required` and `Exclude.` When a goal/requirement describing a variant needs that other goals/requirements are also selected during the product derivation the relationship *Required* must be used to link them. If the selection of a variant implies that another variant cannot be selected during the product derivation the *Exclude* relationship must be employed to relate them. How these relationships are textually described is presented in Table 5-13. As can be observed, an attribute called *rationale* has been included as well. It is thought to describe why the relationship is being defined, as (Bühne et al., 2003) recommends. It must also be considered that the relationship, in our proposal, is established from the required goal/requirement to the requiring goal/requirement. This relationship is mandatory in order to describe properly the analysis.

Table 5-13 Describing Intervariants relationship

```
REQUIRED | EXCLUDE <identifier>      — unique identifier of the relationship,
                                                  it is internally specified
RATIONALE <expression>
FROM  <identifier>                   — identifier of the source requirement
TO  <identifier>                     — identifier of the destination requirement
```

In addition, it must be considered that while the variability is being defined, there are three kinds of interrelations that can emerge between variants and/or variation points, as (Halmans & Pohl, 2003) have stated:

a) *Dependency between variant and variation point.* A variant is obviously related to one or several variation points to describe an alternative to deal with such variability (see Figure 6-11).

b) *Dependency between variant and variant.* It must be taken into account that this dependency can be described in two different context: i) when both of them are related to the same variation point; ii) when they are related to different variation points (see Figure 6-10).

c) *Dependency between variation point and variation point.* This dependency arises when the variability must be resolved considering both variation points, because the variant/s to be selected from one of them depends on the variant/s to be selected from the other.

All of them are described by means of *Intervariant* (either *Required* or *Exclude*) relationships. Considering these kind of interrelations and the above description of *Intervariant* the following constraints were defined:

Table 5-14 OCL Constraints for Intervariant relationship

Relation	Constraint
Intervariant	**context** Intervariant **inv** ((Self.from.oclIsKindOf(Requirement) **implies** Self.to.oclIsKindOf(Requirement)) **or** (Self.from.oclIsTypeOf(Operationalization) **implies**

	Self.to.oclIsTypeOf(Operationalization))) and (-- describing dependency a) (((Self.from.refLeaf→size()>0)**and** (Self.from.refLeaf.oclIsTypeOf(OR)) and (Self.to.refRoot →size()>0)**and** (Self.to.refRoot.oclIsTypeOf(OR)))) or -- describing dependency b) (((Self.from.refLeaf→size()>0)**and** (Self.from.refLeaf.oclIsTypeOf(OR)) and (Self.to.refLeaf→size()>0)**and** (Self.to.refLeaf oclIsTypeOf(OR)))) or -- describing dependency c) (((Self.from.refRoot→size()>0)**and** (Self.from.refRoot.oclIsTypeOf(OR)) and (Self.to.refRoot →size()>0)**and** (Self.to.refRoot.oclIsTypeOf(OR)))))

Table 5-14 describes the constraints applicable to *Intervariant* and its refined relationships. As can be observed, they can only be established among goals/requirements or operationalizations but not mixing both types of artifacts. This is because it does not make any sense that a requirement could need that an operationalization was included. If this were the case, a *Contribution* relationship would be defined instead.

It was also considered that by means of *Exclude* or *Require* only the kinds of inter-relationships b and c can be described. For this reason, the comments mark that either among variants or among variation points the *Intervariant* can be employed.

– As was stated in section 3.3, ATRIUM is intended to provide traceability for crosscutting concerns throughout the whole lifecycle. This implies that mechanisms to detect and specify the crosscutting must be specified. These mechanisms are especially necessary in the context of our case study, where safety, performance, etc, appear in the specification crosscutting the main functionality of the EFTCoR.

  In order to address this issue a `Crosscutting` dependency was included. It is employed whenever a goal/requirement crosscuts another goal/requirement. This crosscutting is usually detected as a constraint or extension that is applied on the target goal/requirement, for instance, when a performance constraint is applied on a functional requirement. This permits to incorporate the (Moreira et al., 2002)'s recommendations, because it provides a mechanism to systematically integrate this quality requirements within the whole specification and allows traceability during later stages of the development (chapter 7).

In AOSD, it is frequently observed that crosscutting relationships, especially during the design and implementation stages, are characterized to express weaving operators. We could find in most of the proposals *before, instead,* and *after* as alternatives to describe the causality of the weaving services, so that: a) *after*: aspect1.service is executed *after* aspect2.service; *before*: aspect1.service is executed *before* aspect2.service; and, c) *instead*: aspect1.service is executed *in place of* aspect2.service.

In the context of AORE, (Rashid et al., 2002) have proposed a different set of operators, which they have called `constraints actions`. Some examples are: *Enforce* which describes an additional condition over a set of viewpoints; or *Ensure* which asserts that a condition should exist for a set of viewpoints. However, as (Rashid et al., 2002) recognize, this set of operators have not been widely validated up-to-date, even they think that some combination are only valid for the developed case study. On the other hand, they do presume that these operators are highly readable and understandable for any stakeholder because of its abstraction level.

This has led us to describe *Crosscutting* in its more generic form, i.e. describing a composition of goals/requirements without indicating anything else. It can be considered that its meaning is highly related to the concept of Extension (from Use Cases) because a base behaviour is extended with another one. We suggest that, in case special needs emerge from the domain, this relationship should be extended for its consideration. It was not necessary such an extension in the EFTCoR nor in the Teachmover.

Related to AORE, we should draw your attention to another work that combines the Goal-Oriented and Aspect-Oriented approach: (Yu et al., 2004). They have introduced the use of Goal Models to detect candidate aspects. In order to do so, they use those *tasks* (how a goal is operationalized) that contribute to more than a goal as candidates in the process. However, it does not solve a problem that is addressed with our proposal: the tangled specification. We are firmly convinced that it is always more convenient in terms of maintainability and legibility of the specification to specify separately the requirements since the crosscutting is detected and not later. This is why this alternative was selected.

Table 5-15 Describing Crosscutting relationship

CROSSCUTTING <identifier>	— unique identifier of the CROSSCUTTING, it is internally specified
FROM <identifier>	— identifier of the source artifact
TO <identifier>	— identifier of the destination artifact

Table 5-16 OCL Constraints for Crosscutting relationship

Relation	**Constraint**
Crosscutting	**context** Crosscutting **inv** Self.from.oclIsKindOf(Requirement) **and** Self.to.oclIsKindOf (Requirement)

Table 5-15 describes the textual description for Crosscutting relationships. Table 5-16 describes the applicable constraint that specifies that only requirements and goals can be related by means of this kind of relationship.

It could be thought that *Crosscutting* and *Require* have an overlapped semantics. In both cases, for a dependency relation from an artifact A to an artifact B, A can be satisfied if B is satisfied as well. However, when *Crosscutting* is used, it is denoted there is a dependency on the specification, that is, the specification of A is not complete if B is not also specified. On the other hand, when *Require* is employed it refers to a dependency of existence because it is just at the moment of resolving the variability when both A and B have to be present in the final product or dynamic architecture.

– The refinement relationships described in the previous section are introduced by the Goal-Oriented approach to specify that an intentional refinement is iteratively applied to the set of goals ending up when every sub-goal is refined as a requirement. Once the requirements have been specified, the operationalizations, i.e., how to provide such requirements, have to be described. It means that a seamless transition to an operational refinement is performed. In addition, it must be considered that, usually, there is not only a possible operationalization for a requirement but several alternatives can be applicable; each one with advantages and disadvantages not only for a specific requirement but also for any other established in the Goal Model. In such a case, it is up to the analyst to examine the impact of such methods on other requirements and decide on what and how many operationalizing methods must be applied via the proper relationship.

In order to cope with these needs the `Contribution` dependency has been introduced. It is a binary relationship to denote how an operationalization contributes to the accomplishment of a requirement. When an alternative is introduced several degrees of contribution can be established. For this reason, a set of symbols [++|+|#|-|--] are used to characterize the relationship. They denote how an operationalization collaborates to satisfy a requirement. Symbols ++ and + describe a strong positive and positive collaboration, i.e., it provides a sufficient or partially sufficient solution, respectively, to satisfy the related requirement. On the other hand, symbols—and - describe a strong negative or negative collaboration, i.e., the operationalization prevents or partially prevents, respectively, the satisfaction of the related goal. The # symbol is introduced to specify operationalizations whose impact (positive or negative) is unknown at that moment. The Contribution can be described as indicated in the Table 5-17. Table 5-18 establishes by means of a constraint that only requirements and operationalizations can be target and source, respectively, of a contribution relationship. It must be underlined that this relationship has been taken from the NFR Framework. It was mandatory to allow the analysis of alternatives, so necessary in the context of ATRIUM.

In addition, it must be considered that while describing the goals/requirements influences can arise among them both positive and negative. This situation has been specially detected in the specification of product-lines and dynamic architectures by means of the called `hinder` and `hint` dependencies. The former is used to describe when a variant has a negative influence on another one and the latter when the influence is positive. With the aim of avoiding an overloaded Metamodel and the overlapped semantics, it was decided that *Contribution* was employed to describe this dependency also among goals/requirements.

Table 5-17 Describing Contribution relationships

CONTRIBUTION <identifier>	— unique identifier of the CONTRIBUTION, it is internally specified
FROM <identifier>	— identifier of the source operationlization
TO <identifier>	— identifier of the destination requirement
CONTRIBUTES ++ | + | # | - | --	— kind of contribution

Table 5-18 Constraint for Contribution dependency

Relation	**Constraint**
Contribution	**context** Contribution **inv** ((Self.from.oclIsTypeOf(Operationalization) **implies** Self.to.oclIsTypeOf (Requirement)) **or** (Self.from.oclIsKindOf(Requirement) **implies**

	Self.to.oclIsKindOf (Requirement))

Table 5-18 depicts those necessary constraints that were explained above: if the origin of the dependency is an Operationalization then its destination must be a requirement; but if the source is a goal/requirement, then the sink must be a goal/requirement as well. In this way, *Contribution* represents the *hinder* and *hint* dependencies.

It can be observed that there is no notation for the `Conflict` relationship in Figure 5-2. This relationship can be set up between two goals if an incompatibility appears between them, in other words, whenever the satisfaction of a goal prevents the satisfaction of another goal. For instance, KAOS provides support for such a relationship. However, it has not been included in the Metamodel because both *Conflict* and *Contribution* would have an overlapped semantics. Both of them describe a conflicting situation that must be resolved in case it appears. For this reason, only *Contribution* has been included.

5.4 A PROCESS FOR THE ATRIUM GOAL MODEL

In order to define properly an ATRIUM Goal Model, a process was defined, providing the analyst with the necessary guidelines. SPEM (Software Process Engineering Metamodel, see Appendix A) was used for its definition. SPEM provides a Metamodel for describing methodologies or software processes. Figure 5-6 sketches the main activities of the defined process. Similarly to any process in RE, there is a first step devoted to the Elicitation and Specification of the ATRIUM Goal Model, which is more detailed in section 5.4.1. During this activity, the standard ISO/IEC 9126 (ISO/IEC 9126) acts as a framework for the initial selection of the concerns, which are to be provided by the system-to-be. This standard along with its exploitation in ATRIUM, is described in section 5.4.1 (ISO/IEC 9126: Selecting and Identifying Concerns).

Once this initial version of the Goal Model is obtained an analysis process is performed as described in section 5.4.2. The evaluation results of this activity are used as a feedback for the previous activity so as to refine and improve the Goal Model. This activity is critical for ATRIUM because one of its main aims is to exploit the Goal Model for the analysis of architectural alternatives.

Finally, the Validation activity is performed in order to confirm that the Goal Model, once analyzed, actually describes the needs and expectations of the stakeholders. It helps in a development technically correct and satisfactory from the point of view of the stakeholders. In case misalignments appear between the Goal Model and the needs and expectations, new iterations on the process would be performed.

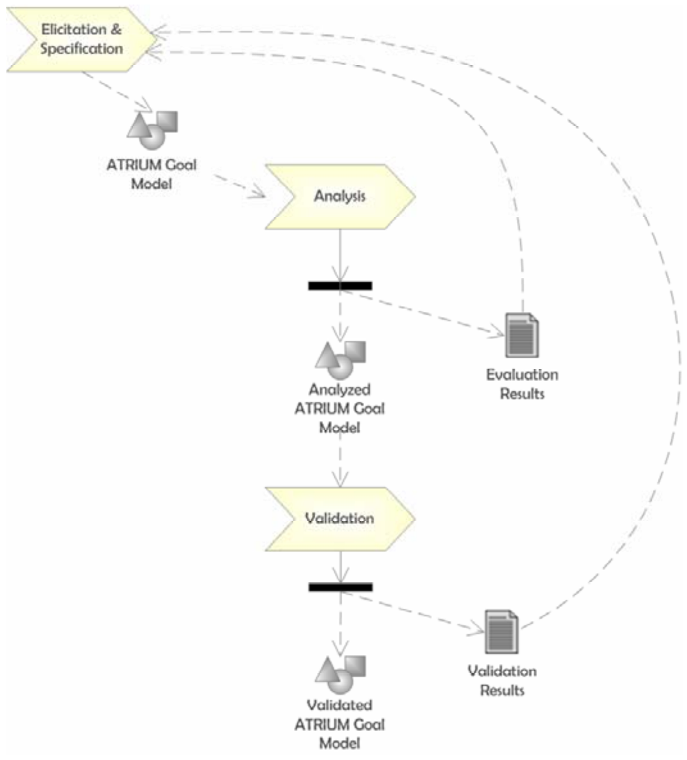

Elicitation & Specification
ATRIUM Goal Model
Analysis
Analyzed ATRIUM Goal Model
Evaluation Results
Validation
Validated ATRIUM Goal Model
Validation Results

Figure 5-5 Process for describing the ATRIUM Goal Model

5.4.1 Elicitation and Specification of ATRIUM Goal Models

Figure 5-6 establishes the set of steps for the Goals Model elaboration along with the input artifacts needed for its realization. Although Figure 5-6 shows only a sequential flow to apply the tasks, in practice, its application is iterative facilitating a progressive refinement of the ATRIUM Goal Model. As was stated in section 3.3.2, SPEM has been used as the process specification language to describe this activity of ATRIUM. For this reason, a refined activity diagram is used where each actionstate of the diagram is describing a step of the activity *Elicitation/Specification* identified in Figure 5-5. Below, each step is extensively described.

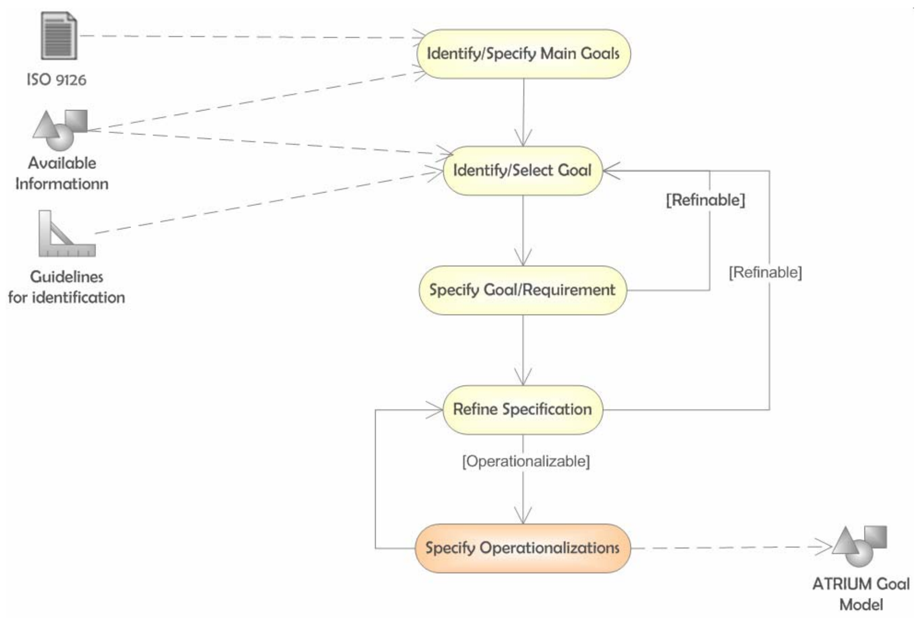


Figure 5-6 Workflow to specify goals and requirements

In addition, as can be observed, one of the artifacts used as an input for the process is the standard (ISO/IEC 9126). It is highly relevant for the final description of the Goal Model. For this reason, its implication in the process is described in the following section.

ISO/IEC 9126: Selecting and Identifying Concerns

Quality criteria, used for the software products assessment, are highly related to the requirements specified in their SRS. This means a practical binding with the global organization of the SRS in order to facilitate the subsequent evaluation of the software quality. In this sense, the ISO/IEC 9126 standard is an important reference as a software quality model. It defines a set of features that are mandatory for any software system that is built following the highest quality levels. This reason makes the ISO/IEC 9126 especially suitable as taxonomy of concerns. It provides an initial framework to elicit and organize goals and requirements. In this way, as the informal software needs are elicited, they can be analyzed, broken down and organized. This allows one to manage the specification crosscutting, by reducing or removing other drawbacks such as: redundancy, inconsistencies, etc. At the same time, crosscutting relationships can be defined in order to re-establish a tangled representation whenever it is needed.

Table 5-19 Quality Characteristics of the ISO/IEC 9126

Quality Type	Characteristic	Sub- Characteristic
Software Product Quality	functionality	Suitability Accuracy interoperability Security Compliance
	reliability	maturity Fault tolerance recoverability compliance
	usability	understandability learneability operability attractiveness compliance
	efficiency	Time behaviour resource utilisation compliance
	maintainability	analysability changeability stability testability compliance
	portability	adaptability installability co-existence replaceability Compliance
Quality in use	Effectiveness Productivity Safety Satisfaction	

ISO/IEC 9126 determines three software quality aspects: process quality, product quality and product in use quality. The main aim for ATRIUM is just the latter, i.e., to notice the product quality through the effect that its use causes. The quality in use depends on or is influenced by the internal and external characteristics of the software product. The software construction process affects these characteristics. With regard to the specific requirements, described in the SRS, we are interested in the characteristics defined for the software product quality and for the quality in use. These characteristics are listed in the Table 5-19, although more details about their description can be obtained from (ISO/IEC 9126).

We have to notice that from the software requirements perspective, this taxonomy goes beyond the traditional classification of functional and non-functional requirements that is not a meaningful contribution to the requirements organization. In fact, most of the typical functional requirements can be set with the *suitability* sub-characteristic. On the other hand, the software product capacity to satisfy the standards, conventions or regulations are broken down as sub-characteristics of type "compliance" below each quality software product sub-characteristic. Although the ISO/IEC 9126 provides us with a wide set of concerns, this set can be extended if needed. In this case, we propose the following alternatives:

a) Considering a new dimension for organizing additional goals/requirements, to the taxonomy proposed by ISO/IEC 9126. This option could be of interest whenever the additional characteristics are as relevant as those already considered and whether the crosscutting with them could be high.

b) Including a new characteristic/sub-characteristic for extending the taxonomy. This alternative would be recommended when the aspects, which are dealt with, are not as relevant as those considered and/or it is not expected that the crosscutting could be so high.

c) Dealing with this element as an attribute of the goal/requirement. This option is suggested when it is coped with aspects which are not so relevant (they are neither exactly goals nor requirements) or do not involve an important crosscutting. However, they are particularly interesting to group and present goals/requirements.

As was stated in the introduction, the IEEE 830-1998 offers several criteria and guidelines to organize specific requirements. It recognizes that there is not an optimal organization to be applicable to every system. Among the mentioned organization criteria are: operation system mode, user type, problem entities, system services, stimulus, answer and/or functions hierarchies. Therefore, as it is recommended by the *c)* alternative, those elements can be dealt as attributes of goals/requirements instead of extending the taxonomy. In this way, it is possible to offer a view related to the joins based on these elements, although they are not elements of the initial taxonomy. For instance, we could deal with the section "logic requirement of the database" (included in the IEEE 830-1998) as an additional dimension (called *data*). Figure 5-7 illustrates the framework extension with new dimensions and how the SRS can be unfold according to different dimensions.

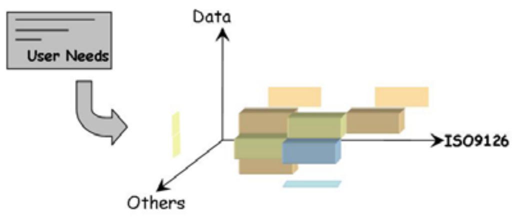


Figure 5-7 Unfolding a Software Specification

Although all the quality characteristics listed in Table 5-19 can have some impact on the Software Architecture, some of them exhibit a more relevant role. According to (Bosch, 2000) we can select *performance*, *maintainability*, *reliability*, *safety* and *security* as the most relevant for this task. Therefore, we are going to focus on this set of quality requirements since this moment.

Elicitation and Specification of the ATRIUM Goal Model

Considering the implication of the ISO/IEC 9126, the elicitation and specification is carried out by means of the following steps, depicted in the Figure 5-5:

– *Identify/Specify Main Goals* is the first task to deal with. As we can observe in Figure 5-6, the standard ISO/IEC 9126 is an input for this task. This model provides an initial view of the *concerns* that could be meaningful for the system. *Available Information* is another input for this task. It collects any information related to the system-to-be such as business goals, user needs, interviews, etc. They help the analyst to identify which concerns can be relevant for the system-to-be.

  In this way, those concerns described in the standard (Table 5-19) considered as relevant for the system-to-be are identified and specified as goals of the Goal Model. This means that every goal, established by means of this step, is aligned with the specification of *concerns* determined by the ISO/IEC 9126 model, and acts as a node for the Goal Model definition. This will provide us with a twofold advantage: on the one hand to facilitate the understanding of the specification, and, on the other hand, to drive the elicitation and analysis process.

  The analyst must fully describe the goals, being especially relevant the establishment of the priority attribute. In this way, those goals that are highly relevant for the stakeholders are the first selected to proceed with the process. As we elicit the requirements, it could be convenient to incorporate new *concerns* to include properly additional goals/requirements, according to the recommendations of the previous section.

– Once the main goals of the system-to-be have been identified and specified, it is just the moment to refine such goals. The task *Identify/Select Goal* is in charge of identifying new goals of the system-to-be being always catalogued according to the main goals. During this activity, the analyst also employs the available information of the system-to-be for this gathering process. This is a crucial task because the discovery of goals is not a straightforward task. Most of the works on Goal Model tackle the problem of goal achievement more than the Goal identification. For this reason, it is highly suggested the use of the existing guidelines. One of the main contributions in this field has been presented by (Antón, 1997). She has described a set of fourteen heuristics based on a question-guided process. An example of that set is described as follow:

  HIG 3. Action words that point to some state that is or can be achieved once the action is completed are candidates for goals in the system. They are identified by considering each statement in the available documentation by asking:

  (a) Does this behaviour or action denote a state that has been achieved, or a desired state to be achieved?

  If the answer is yes, then express the answer to these questions as goals that represent a state that is desired or achieved within the system.

  As can be observed, it is mainly oriented to analyze the available information of the system-to-be by making questions about its content and how it can be interpreted.

– In the task *Specify Goal*, their attributes, such as its name, priority, etc., are not only established but also the necessary refinement relationships (AND/OR) towards its parent. We have to bear in mind that whenever a goal is too coarse to be verifiable, it is decomposed into a set of sub-goals by means of these relationships facilitating a progressive comprehension of the system-to-be. However, when this task is performed it must be determined if the goal to be specified is verifiable, that is, whether a process to determine is achievement for the system can be described. If this is the case, a goal is not specified but a requirement. It means that it is a starting point to perform an operational refinement instead of an intentional one.

– When the *Refine Specification* task is performed, a deeper analysis of the goal or requirement is carried out. This task is mainly devoted to establishing every kind of necessary dependency relationship between the specified goal/requirement and other goals/requirements in the model.

It must be described any necessary Intervariant relationship. We have to include those necessary *Require* dependencies from/to the goal/requirement being specified and any other one when the former needs the latter. On the contrary, if *Exclude* is used, both goals/requirements cannot be simultaneously present when the variability is resolved. The main problem when modeling this kind of dependencies is to find out where they are needed. Most of the existing work is focused on the mechanisms for modeling dependencies (Halmans & Pohl, 2003), possible taxonomies of dependencies (Bühne et al., 2003), etc. However, as far as we know there is no work for helping during this process. It depends on the analyst capability their correct specification.

In addition, it is also during this task when the *Crosscutting* relationship is established. It is in charge of identifying and specifying the crosscutting inherent to any requirements specification. This relationship is used to describe how the behavior associated to a goal/requirement is constrained by or extended with that expressed by another one. Although *Crosscutting* was described in the Metamodel of the Figure 5-3 as a kind of Dependency relationship, similarly to *Intervariant*, they are two disjoint relationships, i.e., they do not have an overlapped semantics (see section 5.3.2, Describing dependency relationships).

In a similar way to the *Intervariant* dependencies, most of the works on candidate aspect identification during the requirements stage are oriented to analyze the requirements specification once it has been performed. (Baniassad & Clarke, 2004) have presented one of the most well-known works supporting this approach called *Theme/Doc*. Its proposal allows one to analyze the specification looking for `Themes`, that is, features of the system under development following an aspect-mining point of view. It is a powerful proposal, because those themes that appear tangled and scattered in the requirements specification can be identified and specified properly.

However, this proposal cannot be applied in ATRIUM, at least directly. Perhaps, as a previous step where the available information is analyzed, providing some notions about the existing crosscutting. But, our approach intends to describe a requirements specification, where those tangled and scattered requirements are properly specified since the very beginning. It could be convenient to provide the analyst with a set of heuristics which help him/her to identify when these *Crosscutting* relationships can emerge, in a similar way to that presented by (Antón, 1996). Unfortunately, to the best of our knowledge there is no work helping in this task. Therefore, the analyst is recommended to include such a relation when he/she detects that the goal/requirement, which is being defined, extends or constraints another one. This situation is usually detected when concerns such as safety, efficiency, or usability are dealt with. A clear example of this situation is introduced in the case study (section 6), where the proposal is put into practice.

– This refinement process of goals goes on until the goal is verifiable, i.e., we can describe scenarios of operationalization. At this moment, we change from an intentional refinement to an operational refinement, and consequently, to the specification of an operational solution from the system-to-be point of view. The task *Specify Operationalization* is responsible for identifying and specifying what elements, either from the environment or the system-to-be, collaborate to realize one requirement.

  During the Goal Model construction, the operationalization is a description of the proposed solution for the realization of a requirement, working this description as an input for the ATRIUM activity *Define Scenarios* (Figure 3-13). The latter cope with the whole definition of the solution through the relevant scenario specification. It is introduced in the Goals Model in order to allow us to describe the relationships between that solution and the already defined requirements in the Model. In this way, we can denote how a solution can contribute to positively realize some requirement and negatively to others. Thanks to these relationships, we achieve a more exhaustive analysis of the set of possible solutions and the establishment of the necessary traceability relationships. Therefore, this is a crucial step in the definition of ATRIUM. This has motivated that it is more exhaustively defined than the previous steps, in the following section.

The described process is iteratively applied across the full set of steps, facilitating that, at the end, an ATRIUM Goal Model of the system-to-be is described. It must be taken into account that a later activity must be performed, thought to analyze the model looking for conflicts, determining the satisfiability of the model, etc. This will facilitate that a proper model is used for the following process of ATRIUM. This activity is introduced in section 5.4.2.

Operationalizating the ATRIUM Goal Model

As was described in section 5.3.1, *operationalizations* are used in the ATRIUM Goal Model as a way of tracing the architectural alternatives from the established *requirements*. Therefore, this is a crucial step in ATRIUM. It entails several steps that Figure 5-8 depicts. They are iterated over mainly two phases.

The first phase focuses its efforts on requirements classified under the concern suitability. The second one only considers the remaining ones. This is because the remaining requirements are usually analyzed as constraints, for instance, efficiency and safety. These relationships are specified in the Goal Model as crosscutting relationships, as was described above. In this way, it is introduced the transformation recommended by Bosch (Bosch, 2000), i.e., integrating quality requirements into functional solutions. Consequently, this extends the architecture with functionality that is not related to the problem domain but used to fulfil the requirement.

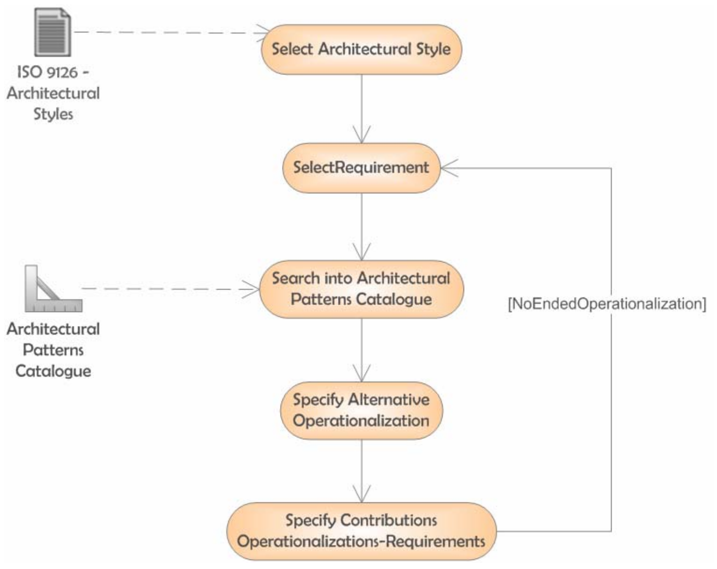

ISO 9126 - Architectural Styles
Select Architectural Style
SelectRequirement
Architectural Patterns Catalogue
Search into Architectural Patterns Catalogue
[NoEndedOperationalization]
Specify Alternative Operationalization
Specify Contributions Operationalizations-Requirements

Figure 5-8 Operationalizing the Goal Model

In addition, we should also bear in mind that crosscutting detected during the Goal Model elaboration does not necessarily imply the use of a specific aspect in the final architecture description but it can be traced to an architectural element, environment element, etc. The analyst must make a decision about what the best alternative is, without loosing the required traceability.

Taking into consideration these two phases, each step is described as follows:

– The initial activity is to *Select Architectural Style*. The Architectural Styles have a high impact on the final description of the architecture because they have an impact throughout the whole architecture or a great part (Bosch, 2000). In the bibliography several works have addressed this issue. For instance, (Shaw & Clements, 1997) offer a proper taxonomy where styles as Layered architecture, Blackboard, etc., have been described. However, domain driven Architectural Styles are lately suggested because they provide analysts with more information and guidance for the architectural description as they are not so oriented to a technological solution. An example in this sense has been the work of (Niemela et al., 2005) that have analysed several Architectural Styles to determine the most appropriate one for describing wireless services. (Fuxman et al., 2001) is an example following this approach. In this sense, they describe organizational Architectural Styles for multi-agent systems. Other examples can be found in other domains, for instance ACROSET (Ortiz et al., 2005). It is an architectural framework for the development of the control units of tele-operated service robots and it has been used in our case study. It promotes a concrete decomposition of the system and identifies some specific kind of components and interaction while defining this kind of systems. All these works have been used to fill in Table 5-20, where one Architectural Style per work has been included.

For this reason, before describing any possible operationalization, the analyst has to make a decision about what Architectural Style should be used for the system-to-be. Because the Goal Model uses the quality model ISO 9126 as the initial framework of concerns, along with their assigned priorities, it can be employed for the selection of the Architectural Style. Therefore, we can evaluate the impact of the Architectural Styles over the ISO characteristics/sub-characteristics, included in the Goal Model. In Table 5-20, three Architectural Styles have been classified according to their positive and negative contributions to every quality characteristic. It is worthy of note that the sub-characteristic suitability has been decomposed into several sub-kinds in order to describe if an Architectural Style is appropriate for a specific domain. For instance, it can be observed that ACROSET contributes positively towards the Tele-operated domain.

Table 5-20 ISO 9126 for selecting Architectural Styles

Characteristic	Sub- Characteristic		Layer	Joint Venture	ACROSET
functionality	suitability	tele-operated	+	--	++
		wireless	++	--	--
		Multi-agent		++	
		…	…	…	…
	accuracy		+	+	+
	interoperability		+		+
	security			-	
reliability	maturity				
	fault tolerance				

	recoverability			
usability	understandability		++	
	learneability		++	
	operability		++	
	attractiveness		++	
efficiency	time behaviour		+	
	resource utilisation			
maintainability	analysability	+		+
	changeability	+	+	+
	stability	+		+
	testability	+		+
portability	adaptability		+	+
	installability			
	co-existence			
	replaceability		+	
effectiveness				
productivity				
safety				+
satisfaction				

– *Select requirement to operationalize.* Analysts have to select from the Goal Model a requirement that has not been operationalized yet. It is suggested that those requirements attributed with higher priority should be selected first in this step.

– *Search into the Design Pattern Catalogue.* Whenever a new system has to be developed, several problems can emerge if the analyst does not have the appropriate knowledge. This deficiency means that poor decisions -and poor designs- can result. In this sense, *patterns* represent distilled experiences that, through their assimilation, enable expert analysts to convey their knowledge and insight to inexpert ones. For this reason, we have included in our proposal both domain and technologic patterns by describing them into a catalogue. These patterns have been indexed according to the following: the concerns of the ISO 9126 it positively or negatively contributes to; and, the Architectural Styles it is more appropriate for. Thus, the analyst can select a pattern or set of patterns that are instantiated to operationalize the selected requirement.

– *Specify alternative operationalizations.* Several alternatives can operationalize the same requirement, just like several alternative programs can implement the same specification. For this reason, the analyst must specify each detected operationalization considering it has to fulfill the requirement. During the specification of an operationalization, the analyst has to do two different tasks: he/she makes a textual description of how it can make operational the related requirement; and, he/she identifies software, hardware and environment elements that collaborate in the operationalization. It is worthy of note that the splitting of the work between the system-to-be and its environment is delayed until the operationalization specification. This avoids taking premature decisions which would limit our ability to define different systems depending on time, costs, available resources, etc, following the (Lauesen, 2003)'s recommendations.

– *Specifiy contribution relationships* In order to obtain the best global solution, operationalizations have to be analyzed caring if conflicts exist among them and other requirements included in the Goal Model. A positive contribution relationship is introduced from each alternative operationalization, identified in the previous step, towards the requirement that has motivated its definition. In addition, if their inclusion means that other requirements can be negatively/positively affected then negative/positive contributions are introduced as well. In case the analyst would want to make a deeper evaluation of any operationalization, it is also possible to describe its associated scenarios, by means of the activity *Define Scenario* explained in chapter 7.

Once the last step has finished, the analyst has to decide if the operationalization process has finished, i.e., if every described requirement has associated operationalizations by means of contributions. The process is iterated repeatedly until there are not more requirements to be operationalized. However, it is not necessary a whole specification to continue the process. On the contrary, a partial description would be enough to proceed with ATRIUM, that is, to analyze the architectural alternatives by applying the activity described in the following section and obtain a draft generation of the architecture.

5.4.2 Analyzing Goal Models

A primary benefit of modeling requirements is the opportunity it provides for analyzing them in order to offer an adequate definition of the system-to-be to the following activities of development. Several techniques have been proposed in the literature, but (Nuseibeh & Easterbrook, 2000) have selected as the most relevant and widely used those described subsequently. It is also explained their applicability in the context of ATRIUM and the decisions made in this sense.

- `Requirements animation` (Balzer et al., 1982) is a well-established technique for checking whether software specifications meet the real intentions and expectations of stakeholders. It tests the correspondences of the specification with the real world problem. Up to date, several approaches have been introduced to deal with animation. Most available tools suggest the specification of design behavior models to be executed. This alternative is widely used despite the fact that mismatches can arise between requirements specification and these models.

  In the context of the Goal-Oriented approach, a proposal has been described by (Van et al, 2004). They propose an animation tool whose aim is to animate a goal-oriented behavior model that is automatically generated from the specification. In this way, it faces the problem of misalignment between animated model and requirements. In addition, it checks whether the requirement specification is appropriate regarding users' needs. In order to exploit this proposal it is mandatory that the model is formalized. This means a problem in terms of non-functional requirements because most of them cannot be formalized.

- `Automated reasoning` is an approach for exploitation of RE specification that several proposals have followed. `Case Based Reasoning` (CBR) is one of these proposals as for instance that presented by (Leake, 1996). In CBR, the primary knowledge source is not a set of generalized rules but a memory of stored cases recording specific prior episodes. By using CBR new solutions can be generated by retrieving the most relevant cases from memory and adapting them to fit new situations.

  In the context of RE, CBR has been used with a clear purpose: requirements specification. During this stage, the ability of reusing similar cases to specify the requirements of new systems exhibits good benefits in terms of cost, time and reliability. Several proposals have been described following this idea such as (Maiden, & Sutcliffe, 1992) or (Massonet & Lamsweerde 1997). The latter could be very helpful in the context of ATRIUM because it has been specifically defined for its use in Goal-Oriented proposals. During the Elicitation/Specification activity, this technique could be used as a guideline to help in this process.

  `Knowledge based critiquing` is a technique that has been mainly used for Deficiency Drive Design Requirements Analysis (DDRA), where a design process is performed looking for a design free of deficiencies. This technique focuses on the exploitation of a base of knowledge where heuristics, design state and design operators are described. Requirements are formally specified (usually by means of temporal logic) in such a way that can be used as constraints to be checked. Based on those formal requirements the design of a system is proposed. This design is the input for an iterative process that refines such design by progressively removing `deficiencies`, that is, failures of the design to satisfy the requirements or constraints. This process uses the design operators to refine the design according to the recommendations of the established heuristics. The process stops when there are no deficiencies on the design. KAREN, proposed by (Fickas & Nagarajan, 1988), is one of most known proposals in this field.

Other approach related to automated reasoning is known as `Satisfiability Analysis`. According to this approach, the reasoning is performed by propagating the satisfaction from the leaf goals towards root goals of the graph according not only to goal evaluations but also to refinement and contribution relationships. These propagation techniques have been used in the Artificial Intelligence field since the sixties (Newell & Simon, 1963), for problem solving. In this case, agents are cooperating for addressing a specific *goal*. They are provided with a set of beliefs, admissible states and actions to look for a specific *plan* for its satisfaction. Since some years ago, these techniques have started to be applied in the RE arena for modeling and analyzing requirements. In RE, this reasoning is carried out to determine whether the goals of the system-to-be will be met mainly for defining, reasoning about and resolving design problems (Louridas & Loucopoulos, 2000), alternative designs (Chung et al., 2000), business goals (Yu & Mylopoulos, 1995), etc.

– `Consistency checking` approach is oriented to determine whether the formal requirements specification satisfy a set of properties needed for the system-to-be. Properties to be checked are, for instance, type correctness for each defined variable; reachability of every state defined in the specification (so necessary when safety requirements are being defined); etc. One of the most widely known proposals has been presented by (Heitmeyer et al., 1996). They propose an automated alternative that exploits SCR, a tabular specification technique for specifying reactive systems as finite-state machines.

In order to decide which technique is the most appropriate to be used in ATRIUM, two key factors should be borne in mind. Firstly, ATRIUM does not pay special attention to the taxonomy of functional and non-functional requirements but to a proper separation of concerns by using the (ISO/IEC 9126) as initial framework for requirements specification. However, when concerns, such as fault tolerance or adaptability, are introduced in an ATRIUM

Goal Model several problems emerge related to the analysis of the model. Techniques with formal foundations exhibit problems for dealing with such kind of concerns, therefore, they are not applicable, at least, in a straightforward way. This means that approaches such as animation, Knowledge based critiquing, and Consistency checking are not recommendable because they are based on formal foundations. Secondly, the main aim of ATRIUM is the analysis of architectural alternatives. Therefore, CBR, knowledge based critiquing or completeness checking are not helpful to address this topic because they are focused on different topics as was described above. This makes emerge *Satisfiability Analysis* as the most useful approach for our purposes, and thus, it was the selected alternative for the analysis of the ATRIUM Goal Model. Its basis and its customization in the context of ATRIUM are introduced in the following section.

Satisfiability Analysis: a Technique for Automated Reasoning

As was stated above, the Goal-Oriented approach has become highly relevant in the Requirements Engineering arena mainly because of the advantages it provides for requirements analysis. Its ability to specify and manage positive and negative interactions among goals allows the analyst to automatically reason about alternatives of the system-to-be.

Following the (Robinson et al., 2003) notation, we could describe the main aim of the satisfiability analysis to hold true the logic statement:

$$Operationalizations \models GM$$

This means that the set of selected Operationalizations exhibits the behavior required by the Goal Model (GM). We can state that the analysis proceeds to hold true the set of formulae (1)-(2). In this formulae, $\xrightarrow[sat]{}$ is used to describe that an operationalization is satisfying a goal; and, *sat(g$_i$)* is a function that informs whether a goal is satisfied. (1) describes that there are not operationalizations conflicts, that is, every operationalization o_1 satisfying a goal g_1 can be combined with any other operationalization o_2 satisfying another goal g_2, in such a way that both goals are also satisfied. In addition, (2) describes that there are no conflicts among goals, that is, there is not a goal whose satisfaction implies that another goal is not satisfied.

$$\exists o_1 \in O, g_1 \in G : o_1 \xrightarrow[sat]{} g_1 \qquad (1)$$
$$\exists o_2 \in O, g_2 \in G : o_2 \xrightarrow[sat]{} g_2$$
$$o_1 \wedge o_2 \xrightarrow[sat]{} r_1 \wedge r_2$$
$$\neg\exists g_1, g_2 \in G : sat(g_1) \Rightarrow \neg sat(g_2) \qquad (2)$$

This technique for satisfiability analysis performs a propagation of satisfiability from the leaves up to the root of the Goal Model looking that the above formulae hold true throughout the process. The propagation is computed throughout the set of refinements and dependency relationships that structure the Goal Model as a directed graph.

However, whenever fault-tolerance or adaptability are concerns of the system-to-be, techniques for reasoning about `partial goal satisfaction` must be introduced. This is because this kind of concerns cannot be said as totally satisfied but that only degrees of satisfaction (*d*) can be achieved. In this case, we cannot use the satisfaction relation as in formula (1), but $\xrightarrow[sat]{}_{d}$ is used instead. In this case, the propagation can be carried out by means of two different approaches:

a) `Qualitative approach`. The idea is to establish positive or negative influences (for instances by means of ++, +, #, -, --) of alternatives on goals in the Goal Model. In this sense, the degree of satisfaction does not have a precise interpretation, i.e., it is not based on domain or system properties but on the analyst criteria. (Chung et al., 2000) and (Giorgini et al., 2003) are examples of this approach.

b) `Quantitative approach`. In this case, weights are set for contribution relationships describing the satisfaction degrees that goals have among them. The propagation performs in a similar way to the previous one, but now a specific value of satisfiability is achieved. Those weights can be assigned according to quite different criteria:

    iii. *Subjective assignment* where only the analyst criteria is used to decide. (Giorgini et al., 2003) is a clear example following this approach;

    iv. *Objective assignment*, which is based on domain properties. Some examples in this category are, for instance, (Letier & Lamsweerde, 2004) for reasoning about partial satisfiability of requirements, by using probability density functions; fault trees (Hansen et al., 1998) to analyze if the system-to-be will break the safety constraints; or queues models (Schopf & Berman, 1998) for performance evaluation.

    We should point out that the quantitative approach takes a wide background from the Bayesian Network research because they provide a probabilistic reasoning mechanism. Their ability to reason about the beliefs that can be held under uncertainty means an ability to model the uncertainty nature of non-functional requirements.

The main problem, when looking for a proposal for goal analysis is that, currently, there is not a standard notation for goal-oriented specification but a diversity of proposals that have emerged. (Kavakli & Loucopoulos, 2005) have described a comparative framework where more than fifteen proposals following the Goal-Oriented approach have been studied.

However, there are not only differences among the proposals themselves but also about how they are applied. If we had a look to the proposals we would realize that they use different rules to evaluate the satisfiability and/or deniability of the goals. It does not only depend on the different kinds of artifacts and relationships but also on how these rules are applied, i.e., what the result of the evaluation is. In these terms, Table 5-21 shows an example of several goal-oriented proposals, concretely, (Chung et al., 2000), (Giorgini et al., 2003), (Letier & Lamsweerde, 2004) and (Yu & Mylopoulos, 1995). We can appreciate their different notation and some examples of how their rules look like.

Table 5-21 Mapping between proposals for propagation

	Concept	**Ghung et al.'s**	**Giorgini et al.'s**	**Letier et al.'s**	**YU et al.'s**
Nodes	Goal		Goal	Goal	Goal
	Softgoal	Softgoal	Not defined	Softgoal	Softgoal
	Requirement	≈Softgoal	Not defined	Requirement	≈Task
	Expectation	≈Softgoal	Not defined	Expectation	
	Claims	Claims			
	Agent (software, environment)	≈Operationalization		≈Agent	≈Actor
	Event		Event		
Relationships	AND	AND	AND	AND	≈Refinement
	OR	OR	OR	OR	Not defined
	CONTRIBUTION $g_s \xrightarrow[label]{} g_D$	*Label*∈{MAKE++, SOME+, =, SOME-, BREAK--}	*Label*∈{++, +, #, -, --,$D^?$, $S^?$ } $^?$∈{++, +, #, -, --}	Not defined	≈DEPENDENCY Label∈{±Δ∇}
Rules	$g_1 g_2 \xrightarrow[AND]{} g_D$	$Sat(g_d)=min(g_1, g_2)$ where g_i∈{D, W^-, U, W^+,S, C}	$Sat(g_d)=min(g_1, g_2)$; $Den(g_d)=min(g_1, g_2)$ where g_i∈{F,P,N}	$Sat(g_d)=min(g_1, g_2)$ where g_i∈{S, D, U}	
	CONTRIBUTION $g_s \xrightarrow[label]{} g_D$	*label*=SOME+ $Sat(g_s)=D \Rightarrow Sat(g_d)=W^-$ $Sat(g_s)=C \Rightarrow Sat(g_d)=C$ $Sat(g_s)=U \Rightarrow Sat(g_d)=U$ $Sat(g_s)=S \Rightarrow Sat(g_d)=W^+$	when label=+S $Sat(g_d)=min(Sat(g_s), P)$; $Den(g_d)=N$ where g_i∈{F,P,N}	Not defined	Not defined

It can be appreciated that the rules applied to determine the satisfiability of a goal are dependent on the proposal that is followed. For instance, it can be appreciated that when the satisfiability for a root node of an AND relationship is determined, Chung et al.'s proposal use the minimum of an enumerated set whose values are *denied, weakly denied, undecided, weakly satisfied, satisfied* or *conflict* (the goal is satisfiable and deniable simultaneously). However, Giorgini al.'s proposal propagates separately the satisfiability and deniability, so the values can be *fully, partially* or *none*. There is no notation for conflict but it can be detected whenever a goal is satisfied and denied (fully or partially) at the same time. Although all of them have been defined to analyze the satisfiability, each one describes a different proposal that could be additionally constrained by the application domain.

Furthermore, these rules could also be specific for the project or could even be modified in the same project in order to reflect some additional consideration during the analysis. Therefore, whereas many algorithms about reasoning with goal models already exist, the integration of one of them would spoil the dynamic possibilities and freedom any analyst would want for their projects. For this reason, a proposal for customizing these techniques according to his/her specific needs will improve the analysis and decision making process.

For this reason, the followed proposal was to introduce a framework for exploiting Goal Models that allows the analyst to customize the analysis mechanisms according to the project needs. This framework is based on the propagation algorithm proposed by (Giorgini et al. 2003), which establishes the essential computation of propagation. In addition, the metamodeling technique, introduced in section 5.2, constitutes a valuable asset to the definition of the framework. It provides the analyst with extensibility and customization mechanisms to specify the types of artifacts and relationships of the particular Goal Model being analyzed. These types are used to describe the necessary rules that are customized according to the application domain, business rules, etc. In the following section, it is introduced how this approach has been defined.

A customizable analysis process

As was stated above (Giorgini et al., 2003)'s algorithm has been selected to describe the main computation of our proposal because, as far as we know, this is the only proposal which clearly states this issue. In their proposal, they have specified that Satisfiabiliaty and Deniability, Sat(g_i) and Den(g_i) respectively, for each goal g_i is performed separately according to a defined set of rules. Both Sat(g_i) and Den(g_i) take their values from the ordered set {None, Partially, Fully}.

Table 5-22 Qualitative Propagation rules described by (Giorgini et al., 2003), where $\xrightarrow{++S}$, $\xrightarrow{--S}$, etc. are describing contribution relationships.

	$(g_2,g_3)\xrightarrow{and}g_1$	$g_2\xrightarrow{+S}g_1$	$g_2\xrightarrow{-S}g_1$	$g_2\xrightarrow{++S}g_1$	$g_2\xrightarrow{--S}g_1$
sat(g_1)	$\min\left\{\begin{matrix} sat(g_2), \\ sat(g_3) \end{matrix}\right\}$	$\min\left\{\begin{matrix} sat(g_2), \\ P \end{matrix}\right\}$	N	sat(g_2)	N
den(g_1)	$\max\left\{\begin{matrix} den(g_2), \\ den(g_3) \end{matrix}\right\}$	N	$\min\left\{\begin{matrix} sat(g_2), \\ P \end{matrix}\right\}$	N	sat(g_2)

Table 5-22 shows an example of how these rules look like. For instance, it can be observed that the contribution relationship cannot be symmetric, that is, only satisfaction (or denial) is propagated, which is indicated by means of the attribute S on the relationship (D for denial). This means that only when g_2 is satisfied the contribution relationship propagates its value. For instance, $g_2\xrightarrow{-S}g_1$ means that if g_2 is satisfied, then there is some evidence that g_1 is denied. For this reason, it can be appreciated that a rule has been defined for the deniability of g_1 but not for its satisfiability.

Giorgini et al. have also described a proposal to perform a quantitative propagation of the satisfiability. As was described above, in this case the satisfaction is not evaluated in an enumerated set but a numeric degree of satisfaction is determined. It proposes a subjective assignment of the satisfiability. The reader is referred to the Giorgini et al.'s work to obtain more details about this proposal.

In addition, Giorgini et al. have also described an algorithm for the propagation of the satisfiability, which is introduced in Table 5-23. The function *Label_Graph* iterates over the set of artifacts of the graph being analyzed to update the labels, where each label describes the values of satisfiability and deniability. The iteration ends when the set of labels does not change from iteration to another. *Update_Label* is in charge of determining the satisfiability of the goal G_i. In order to do so, for each relationship having G_i as destination, new values of satisfiability and deniability are derived applying the rules (partially described in Table 5-22).

Table 5-23 Giorgini et al.' algorithm for propagation of the satisfiability

```
label_array Label_Graph(graph (G, R) label_array Initial)
  label_array Current=Initial;
  do
      Old=Current;
      for each Gi ∈ G do
          Current[i] = Update_Label(i, (G, R) Old);
```

```
7      until not (Current==Old);
8      return Current;

9    label Update_Label(int i; graph (G, R) label_array Old)
10     for each Rj ∈ R s.t. target(Rj) == Gi do
11         satij  = Apply_Rules_Sat(Gi, Rj, Old);
12         denij = Apply_Rules_Den(Gi, Rj, Old);
13     return (max (maxj (satij);Old[i].sat), max(maxj(denij);Old[i].den))
```

In order to allow the analyst to customize the rules to be used during the propagation process an extension to the algorithm proposed by (Giorgini et al., 2003) has been defined. In this sense, the customization allows the analyst to include any kind of relationship and artifact along with their attributes to describe the propagation rules.

Table 5-24 shows how Giorgini et al.'s algorithm has been modified for customization purposes. It can be appreciated that lines 11 to 13 of the initial algorithm have been dropped. Instead, lines 11 to 22 have been added. As can be noticed, the initial proposal only describes the valuation for two attributes (*sat* and *den*) with a fixed set of rules (lines 11-12 of the Giorgini et al.'s algorithm). However, with our proposal the set of rules to be evaluated can be customized according to the specific needs of the project, as will be described below.

Table 5-24 Propagation algorithm based on (Giorgini et al., 2003)'s proposal

```
1    label_array Label_Graph(graph (G, R); label_array Initial)
2      Current=Initial;
3      do
4      Old=Current;
5      for each $G_i$ ∈ G do
6              Current[i] = Update_Label(i, (G, R) Old);
7      until (Current==Old);
8      return Current;

9    label Update_Label(int i; graph (G, R); label_array Old)
10     for each $R_j$ ∈ R s.t. target($R_j$) == $G_i$ do
11         rules= applicable_rules ($G_i$; $R_j$)
12         for each $rule_k$ ∈ rules                                //added
13             if applicable ($rule_k$, $G_i$, $R_j$, Old) then          //added
14                 valuate ($rule_k$, $G_i$, $R_j$, Old)
15     return set_valuable_attributes(rules, $G_i$)                //added

16   rules_array applicable_rules(Artifact $G_i$; Relation $R_j$)
17     //returns the set of rules that are applicable to the type of artifact of $G_i$ and
18     //the type of relationship of $R_j$

19   boolean applicable(Rule $rule_k$; Artifact $G_i$; Relation $R_j$; label_array Old)   //added
```

```
20      //it checks whether rule$_k$ holds true or not

21    valuate (Rule rule$_k$; Artifact G$_i$; Relation R$_j$; label array Old)              //added
22    // it applies rule$_k$ to compute the valuate of one of the valuable attributes of G$_i$
```

It is shown in Table 5-24 that *Update_label* has one more loop to iterate over the set of applicable rules for a given relationship R_j having G_i as destination. For each rule, two steps must be performed. First, *applicable* determines whether the condition holds true. If it is true, *valuate* specifies the computation of the propagation for the artifact G_i, as a second step. In addition, *Update_label* returns a label that is formed by the set of valuable attributes of G_i.

As can be observed, line 13 of the Giorgini et al.'s algorithm specifies that the maximum values between the old and new value of *sat* and *den* are returned. However, this is not indicated in the proposed alternative. This means that we are not selecting, by default, the maximum value between the new computed value and the old value for the attribute being valuated, but the analyst can decide whatever he/she needs.

In addition, it can be appreciated that rules have been split into two parts for their definition. The first one describes which condition must hold true to apply the rule. For the Giorgini et al.'s proposal, these conditions are those described in the first row of the Table 5-22. The second part describes how the computation is performed. For the Giorgini et al.'s proposal, it is specified in the following rows of the table. In view of this detail, Backus-Naur Form (BNF) notation has been employed to specify the grammar of the condition and the valuation in Table 5-25 and Table 5-26, respectively.

Table 5-25 describes two different grammars for the condition. Table 5-25 (b) describes the grammar when the rule for a dependency relationship is described. It can be appreciated that logic and relational operators can be employed to describe the condition. In addition, some functions are also made available to determine the minimum, maximum, etc, between two values by means of *<pairfunction>*. In addition, the *<Attribute>* is used to describe the attributes of the source and destination artifacts, and the attribute of the dependency relationship for which the rule is being described.

Table 5-25 (c) describes the grammar when the rule for a refinement relationship is described. It must be considered that a refinement relationship implies that several leaves artifacts are related to a root artifact (for instance, when the AND relationship is applied in the ATRIUM Goal Model). This means that group functions could be applied to determine the minimum, maximum, etc., value of the set of leaves artifacts. These group functions are described by means of <Function>. It can be appreciated that they can be applied either on <Attribute> or <LeafAttribute>, that is, the attributes of the leaves artifacts; or the attributes of each relation between the leaf artifact and the refinement relationship. As was presented in section 5.3, this expressiveness can be very helpful when any information regarding this link must be recorded in the model to be lately used.

Both in (b) and (c), there is no mention to whether a quantitative or a qualitative approach is applied. When the condition is being described any type of attribute, artifact, and relationship can be needed to describe which condition must be hold, providing the analyst with more expressive power. For this reason, any restriction has been considered to describe the BNFs of the condition.

Table 5-25. BNFs for describing condition grammar

(a) Sets and Terminals for the condition grammars

```
! ------------------------------------------------ Sets
{ID Head}      = {Letter} + [_]
{ID Tail}      = {Alphanumeric} + [_]
{String Chars} = {Printable} + {HT} - ["]
! ------------------------------------------------ Terminals
Identifier    = {ID Head}{ID Tail}*
StringLiteral = '"' {String Chars}* '"'
DecLiteral    = {Digit}+          ( [UuLl] | [Uu][Ll] | [Ll][Uu] )?
RealLiteral   = {Digit}*'.'{Digit}+
```

(b) Describing BNF for condition grammar when a dependency relationship is implied

```
"Start Symbol" = <conditionDependency>
! ------------------------------------------------ Rules
<conditionDependency> ::= <OrExp>

<OrExp> ::= <OrExp> '||' <AndExp>
      |  <AndExp>

<AndExp> ::= <AndExp> '&&' <Expression>
      |  <Expression>

<Expression>  ::= <Add Exp> '>'  <Add Exp>
           |  <Add Exp> '<'  <Add Exp>
           |  <Add Exp> '<=' <Add Exp>
           |  <Add Exp> '>=' <Add Exp>
           |  <Add Exp> '==' <Add Exp>    !Equal
           |  <Add Exp> '<>' <Add Exp>    !Not equal
           |  '(' <OrExp> ')'

<Add Exp>     ::= <Add Exp> '+' <Mult Exp>
           |  <Add Exp> '-' <Mult Exp>
           |  <Mult Exp>

<Mult Exp>    ::= <Mult Exp> '*' <Value>
```

```
            |  <Mult Exp> '/' <Value>
            |  <Value>

<Value>      ::=   <pairfunction>
            |   <Literal>
            |   <Attribute>
            |  '(' <Add Exp> ')'

<pairfunction> ::=  <KindFunction> '(' <Value> ',' <Value> ')'

<KindFunction> ::= 'max' | 'min' | 'avg' | 'sum'

<Literal> ::=  StringLiteral
            |  DecLiteral
            |  RealLiteral

<Attribute>   ::= Identifier'.'Identifier
```

(c) Describing BNF for condition grammar when a refinement relationship is implied

```
"Start Symbol" = <conditionRefinement>
! ------------------------------------------------ Rules
<conditionRefinement> ::= <OrExp>
<OrExp> ::= <OrExp> '||' <AndExp>
       |  <AndExp>

<AndExp> ::= <AndExp> '&&' <Expression>
       |  <Expression>

<Expression>  ::= <Add Exp> '>'  <Add Exp>
            |  <Add Exp> '<'  <Add Exp>
            |  <Add Exp> '<=' <Add Exp>
            |  <Add Exp> '>=' <Add Exp>
            |  <Add Exp> '==' <Add Exp>    !Equal
            |  <Add Exp> '<>' <Add Exp>    !Not equal
            |  '(' <OrExp> ')'


<Add Exp>     ::= <Add Exp> '+' <Mult Exp>
            |  <Add Exp> '-' <Mult Exp>
            |  <Mult Exp>

<Mult Exp>    ::= <Mult Exp> '*' <Value>
            |  <Mult Exp> '/' <Value>
            |  <Value>

<Value>      ::=   <function>
            |  <pairfunction>
            |   'count' '(' <Attributes> ',' <Literal>')'
            |   <Literal>
            |   <Attribute>
            |  '(' <Add Exp> ')'
```

```
<Function> ::= <KindFunction> '(' <Attributes> ')'

<pairfunction> ::=  <KindFunction> '(' <Value> ',' <Value> ')'

<KindFunction> ::= 'max' | 'min' | 'avg' | 'sum'

<Literal> ::=  StringLiteral
           |  DecLiteral
           |  RealLiteral

<Attributes>::= <LeafAttribute>
           |   <Attribute>

<LeafAttribute>   ::= Identifier'.Leaf.'Identifier

<Attribute>   ::= Identifier'.'Identifier
```

Table 5-26 shows the BNF for the valuation grammar. It can be appreciated that in this case, a distinction has been made not only for the relationship but also for the type of artifact being evaluated. Both the qualitative and quantitative approaches for propagation are supported, so that both enumerated and numeric attributes can be used when the valuation of a rule is described. For this reason, the arithmetic operators (+, -, *, /) are available only for numeric attributes in Table 5-26 (b) and (d), but not for the enumerated ones in Table 5-26 (c) and (e). In a similar manner to the condition, group functions are also available when the valuation of a rule for a refinement relationship is described. However, relational and logic operators cannot be applied in the valuation of any rule. It must be highlighted that the attribute being evaluated can be employed when the rule is described. Because of this, the analyst can establish if the new value of the attribute being evaluated will be the minimum or the maximum between the old and the new value.

Table 5-26 BNF for describing valuation grammar

(a) Set and Terminals for the valuation grammars

```
! ------------------------------------------------ Sets
{ID Head}      = {Letter} + [_]
{ID Tail}      = {Alphanumeric} + [_]
{String Chars} = {Printable} + {HT} - ["]

! ------------------------------------------------ Terminals

Identifier    = {ID Head}{ID Tail}*
StringLiteral = '"' {String Chars}* '"'
DecLiteral    = {Digit}+          ( [UuLl] | [Uu][Ll] | [Ll][Uu] )?
RealLiteral   = {Digit}*'.'{Digit}+
```

(b) Valuation when a refinement relationship has as destination artifact with a Numeric attribute being valuated

```
"Start Symbol" = <valuationNumericRefinement>
! ------------------------------------------------ Rules
<valuationNumericRefinement> ::= <Add Exp>

<Add Exp>     ::= <Add Exp> '+' <Mult Exp>
              |  <Add Exp> '-' <Mult Exp>
              |  <Mult Exp>

<Mult Exp>    ::= <Mult Exp> '*' <Value>
              |  <Mult Exp> '/' <Value>
              |  <Value>

<Value>       ::= <function>
              |  <pairfunction>
              |  'count' '(' <Attributes> ',' <Literal> ')'
              |  <Attribute>
              |  <Literal>
              |  '(' <Add Exp> ')'

<Function> ::= <KindFunction> '(' <Attributes> ')'

<pairfunction> ::=  <KindFunction> '(' <Value> ',' <Value> ')'

<KindFunction> ::= 'max' | 'min' | 'avg' | 'sum'

<Literal>::=   DecLiteral
              |  RealLiteral

<Attributes>::= <LeafAttribute>
              |   <Attribute>

<LeafAttribute>   ::= Identifier'.Leaf.'Identifier

<Attribute>   ::= Identifier'.'Identifier
```

(c) Valuation when a Refinement relationship has as destination artifact with an Enumerated attribute being valuated

```
"Start Symbol" = < valuationEnumeratedRefinement >
! ------------------------------------------------ Rules
<valuationEnumeratedRefinement> ::= <Value>

<Value>       ::= <function>
              |  <pairfunction>
              |  'count' '(' <Attributes> ',' <Literal> ')'
              |  <Attribute>
              |  <Literal>

<Function> ::= <KindFunction> '(' <Attributes> ')'

<pairfunction> ::=  <KindFunction> '(' <Value> ',' <Value> ')'
```

```
<KindFunction> ::= 'max' | 'min'

<Literal>::=   StringLiteral
            |  DecLiteral
            |  RealLiteral

<Attributes>::= <LeafAttribute>
            |   <Attribute>

<LeafAttribute>   ::= Identifier'.Leaf.'Identifier

<Attribute>   ::= Identifier'.'Identifier
```

(d) Valuation when a Dependency relationship has as destination artifact with a Numeric attribute being valuated

```
"Start Symbol" = < valuationNumericDependency >
! ------------------------------------------------- Rules
<valuationNumericDependency> ::= <Add Exp>

<Add Exp>     ::= <Add Exp> '+' <Mult Exp>
            |  <Add Exp> '-' <Mult Exp>
            |  <Mult Exp>

<Mult Exp>    ::= <Mult Exp> '*' <Value>
            |  <Mult Exp> '/' <Value>
            |  <Value>


<Value>       ::= <pairfunction>
            |  <Attribute>
            |  <Literal>
            |  '(' <Add Exp> ')'

<pairfunction> ::=  <KindFunction> '(' <Value> ',' <Value> ')'

<KindFunction> ::= 'max' | 'min' | 'avg' | 'sum'

<Literal>::=   DecLiteral
            |  RealLiteral

<Attribute>   ::= Identifier'.'Identifier
```

(e) Valuation when a Dependency relationship has as destination artifact with an Enumerated attribute being valuated

```
"Start Symbol" = < valuationEnumeratedDependency>
! ------------------------------------------------- Rules
<valuationEnumeratedDependency> ::= <Value>

<Value>       ::= <pairfunction>
            |  <Attribute>
            |  <Literal>
```

```
<pairfunction> ::=  <KindFunction> '(' <Value> ',' <Value> ')'

<KindFunction> ::= 'max' | 'min'

<Literal>::=  StringLiteral
          | DecLiteral
          | RealLiteral

<Attribute>  ::= Identifier'.'Identifier
```

It is worthy of note that a sugar syntax has also been used to facilitate the description of the source and destination artifacts. Concretely, "_S" is used as a post-fix for the source artifact; "_i" is employed when source artifacts are leaves; and, "_D" is used to specify the destination artifact. It facilitates the legibility of the rule and it enables that the destination artifact can be employed when the condition and the valuation are described.

For instance, considering how the *CONTRIBUTION* relationship (Table 5-22) is evaluated by Giorgini et al. we can appreciate that both the state of this relationship and the source Goal (G_S) are used to determine if the rule can be applied or not. In this sense, the condition could be described as: *G_S(satisfied) && label=--S*, i.e., G_S has an attribute that describes if G_S is satisfied. It is similarly applied to *label*, i.e., *CONTRIBUTION* relationship needs an attribute for specifying—*S* as its kind of contribution. In these terms, the best alternative is to represent these attributes following a syntax as described in Table 5-26 for *<identifier>*, i.e., by prefixing the attribute name with the name of the artifact or the relationship (see (3)).

(Goal_S.satisfied='Full') and (CONTRIBUTION.contributes)= '–S' (3)

In addition, when refinement relationships are considered, for instance an AND relationship, group functions can be used to determine the condition being applied to the set of artifacts G_1 to G_n. For instance, (4) describes that the valuation is performed when the destination Goal has Performance as concern, its priority is high and, at least, it has two fully satisfied leaves goals.

(Goal_D.concern='Peformance') and (Goal_D.priority='High') and count(Goal_i.satisfied,'F')>=2 (4)

As was stated above, only enumerated attributes are made available to the analyst while describing qualitative valuations. This restriction is straightforward so that possible valuations are always constrained to a set of values. For instance, when considering the satisfiability, as described by Giorgini et al., the set {*Full, Partial, None*} is used. It also facilitates the valuation of this kind of attributes by describing functions for enumerations (*<pairfunction>*). This requires the set to be defined as an ordered set in order to apply properly these functions (min and max). This means that the valuation for an AND relationship could be easily described as appears in (5).

$$Goal\_D.sat = min(Goal\_i.satisfied) \quad (5)$$

Related to the artifacts involved in a refinement relationship, aggregated functions (<*function_enum*>) can be used for its treatment as described in (6), where the satisfiability is the sum of the satisfiabilities of its leaves goals minus the product of their satisfiability.

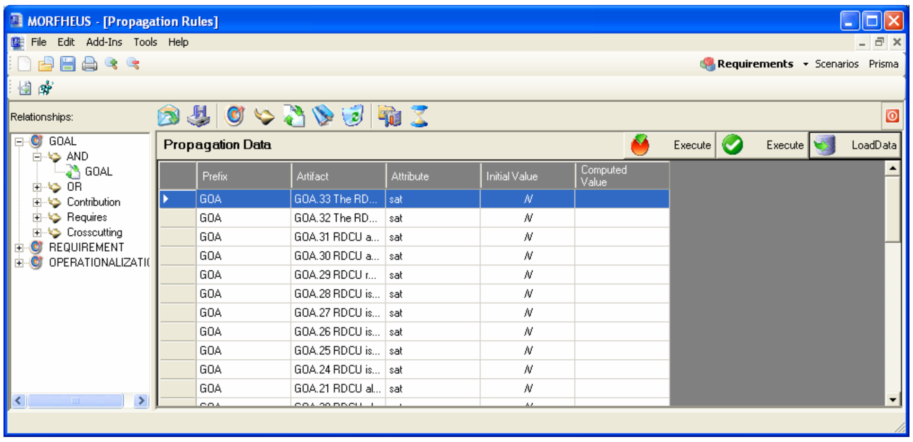
$$Goal\_D.sat = sum(Goal\_i.sat)\text{-} prod(Goal\_i.sat) \quad (6)$$

Taking into account how the grammar for *valuation* and *condition* has been defined, the rules established by Giorgini et al.'s are easily described using the proposal. Table 5-27 presents an example of how the rules shown in Table 5-22 can be described by means of this proposal.

Table 5-27 Describing the Giorgini et al.'s rules using the proposal

Relationship	**Condition**	**Valuation**
Sat $((g_2,g_3)\xrightarrow{and}g_1)$		Goal$_D$.Sat = max((min(Goal$_i$.Sat), Goal$_D$.Sat)
Den $((g_2,g_3)\xrightarrow{and}g_1)$		Goal$_D$. Den = max(max(Goal$_i$.Den), Goal$_D$. Den)
Sat$(g_2\xrightarrow{+S}g_1)$	(Contribution.contributes='+S')	Goal$_D$.Sat = max(min(Goal$_S$.Sat, 'P'), Goal$_D$.Sat)
Den$(g_2\xrightarrow{+S}g_1)$	(Contribution.contributes ='+S')	Goal$_D$.Den = max('N', Goal$_D$.Den)
Sat$(g_2\xrightarrow{-S}g_1)$	(Contribution.contributes ='-S')	Goal$_D$.Sat = max('N', Goal$_D$. Sat)
Den$(g_2\xrightarrow{-S}g_1)$	(Contribution. contributes =' -S)	Goal$_D$. Den = max(min(Goal$_S$.Sat, 'P')), Goal$_D$. Den)
Sat$(g_2\xrightarrow{++S}g_1)$	(Contribution.contributes =' ++S')	Goal$_D$.Sat = max(Goal$_S$.Sat , Goal$_D$. Sat)
Den$(g_2\xrightarrow{++S}g_1)$	(Contribution.contributes ='++S')	Goal$_D$.Den = max('N', Goal$_D$. Den)
Sat$(g_2\xrightarrow{--S}g_1)$	(Contribution.contributes ='—S')	Goal$_D$.Sat = max('N', Goal$_D$. Sat)
Den$(g_2\xrightarrow{--S}g_1)$	(Contribution.contributes ='—S')	Goal$_D$.Den = max(Goal$_S$.Sat , Goal$_D$. Den)

These rules would be included in the preceding algorithm, in such a way that *applicable* includes the condition to determine if the valuation can be performed, and *valuate* includes the valuation associated to each rule. It must be highlighted that this is not a theoretical proposal, but it can be put into practice by means of an add-in of MORPHEUS, the ATRIUM support tool. How this add-in has been developed using the facilities of dynamic compilation of code is presented in section 9.3.3 along with the capabilities it provides. However, Figure 5-9 depicts how the tool shows the results of the propagation performed using the Giorgini et al.'s rules. It describes the initial values before the propagation, and the computed values after the propagation. On the right side appears the type of artifacts and relationships used for describing the rules.



Prefix	Artifact	Attribute	Initial Value	Computed Value
GOA	GOA.33 The RD...	sat	N	
GOA	GOA.32 The RD...	sat	N	
GOA	GOA.31 RDCU a...	sat	N	
GOA	GOA.30 RDCU a...	sat	N	
GOA	GOA.29 RDCU r...	sat	N	
GOA	GOA.28 RDCU is...	sat	N	
GOA	GOA.27 RDCU is...	sat	N	
GOA	GOA.26 RDCU is...	sat	N	
GOA	GOA.25 RDCU is...	sat	N	
GOA	GOA.24 RDCU is...	sat	N	
GOA	GOA.21 RDCU al...	sat	N	

Figure 5-9 Propagation results for a simplified model of the Teachmover

However, it must be emphasized that the proposal is more powerful than it was thought at the beginning. As can be noticed, the computation can be performed using any type of artifact or relationship along with any attribute they have. This means that the satisfiability analysis can be customized, for instance, to take into account the variability expressiveness of the ATRIUM Goal Model, described in the section 5.3. In this case, it has to be considered that to describe a *variation point* its *multiplicity* must be specified, i.e., how many variants must exist at the same time in a product or architecture when the variability is being resolved. Table 5-28 shows how the rules for the OR relationship has been modified for dealing with variability expressiveness.

Table 5-28 Describing variability rules using the proposal

Relationship	Condition	Valuation
Sat $((g_2,g_3)\xrightarrow{or}g_1)$	(count(Goal_i.Sat, "+S") +	Goal_D.Sat =

	count(Goal_i.Sat, "++S")) >= OR.multiplicity.min) && (count(Goal_i.Sat, "+S") + count(Goal_i.Sat, "++S")) <= OR.multiplicity.max)	max (max(Goal_i.Sat), Goal_D.Sat)
Den $((g_2,g_3)\xrightarrow{or}g_1)$	(count(Goal_i.Sat, "+S") + count(Goal_i.Sat, "++S")) < OR.multiplicity.min) || (count(Goal_i.Sat, "+S") + count(Goal_i.Sat, "++S")) > OR.multiplicity.min)	Goal_D.Den= min(max(Goal_i.Sat), "P")

Analysing architectural alternatives

We should draw your attention to a key point. ATRIUM has been described to provide the analyst with guidance in the process of architectural specification. It was presented in section 5.3.1 that operationalizations are a building block in the construction of the ATRIUM Goal Model. Each operationalization describes an architectural alternative that is introduced to satisfy at least one requirement.

Considering this, the propagation, as was described in the previous section, can be applied with a straightforward purpose: `Forward reasoning`, i.e., it takes into account that certain leaf goals are fulfilled to determine whether all root goals are also fulfilled. Specifically, a set of Operationalizations could be set as Satisfied to determine if the ATRIUM Goal Model is satisfied and, consequently, the requirements are met.

This means that the activity *Analysis,* introduced in the Figure 5-5, could be applied as described in Figure 5-10. Firstly, the set of rules to compute the propagation must be selected, if it was previously described, or specified, using the Goal Model as an input for the activity. In case the rules are being specified, the Metamodel of the Goal Model is used to describe properly the rules providing the analyst with the types of artifacts, dependencies and refinements used in its definition. In addition, any additional information, such as standards, recommendations, etc, could also be used for the description of the rules.

Once the rules are available, the analyst must set the satisfiability of the operationalizations, according to the specific rules that are in use. For instance, it could be set to "F", "P" or "N" if the Giorgini et al.'s proposal was used. These values are used as an input to trigger the propagation, as described in the previous section, and obtain the valuation results, i.e., a table where each artifact, along with its initial and computed values, is shown.

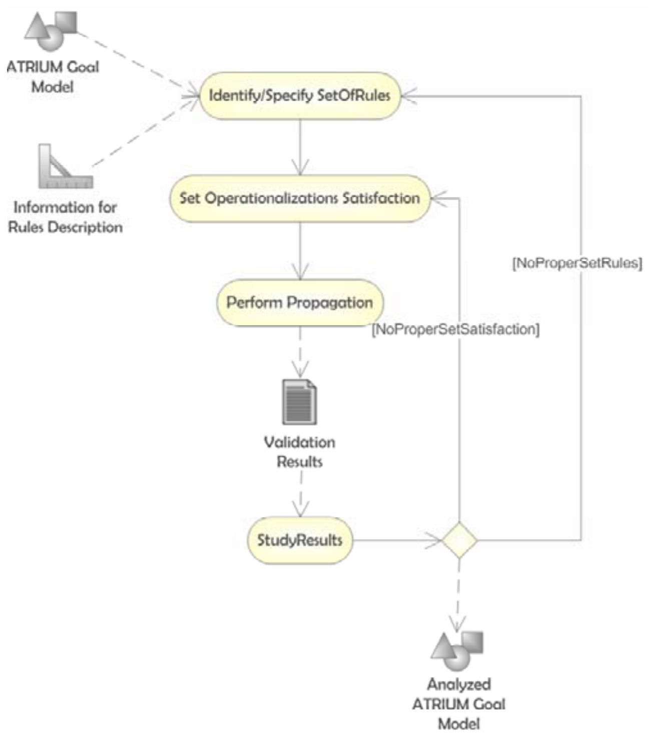

ATRIUM Goal Model
Information for Rules Description
Identify/Specify SetOfRules
Set Operationalizations Satisfaction
Perform Propagation
[NoProperSetRules]
[NoProperSetSatisfaction]
Validation Results
StudyResults
Analyzed ATRIUM Goal Model

Figure 5-10 Describing the Analyze activity

The valuation results are used as an input to determine whether the rules should be modified, reinitializing the process, or the operationalizations satisfaction should be changed to try with different architectural alternatives. However, if the analyst considers the results are the expected ones, he/she can proceed to the next activity of the methodology (chapter 7). In order to determine which alternative must be selected in the decision node, the analyst looks for conflicts in the results. For instance, when a goal is simultaneously satisfied or denied; or, if there are goals that have not been satisfied.

There is an alternative to that proposed in this section called `Backward reasoning.` It tries to determine the set of leaf goals that together fulfil all root goals. In this case, the analyst does not have to specify the satisfiability of the operationalization prior to computing the propagation, but an automatic process is performed. It is oriented to iteratively change the value of satisfiability of the operationalizations, propagate their values and detect how much conflicts exist for a specific set. This process converges when there are no conflicts, a reduced number is detected, or, there are no more alternatives to try. This kind of reasoning has not been included because the main computation to be used is the same as the one previously described. The difference comes from the computation to select the alternatives and iterating repeatedly over them.

5.5 CONCLUSIONS

Goal Models are a very promising technique to improve requirements elicitation. Thanks to their especial capabilities to analyze goals/requirements, they can be used at early stages of the requirements engineering process, when alternatives are explored, conflicts are identified and, in general, the project phase is the requirement negotiation. However, Goal Model techniques must face a common obstacle in RE: the diversity of proposals with an evident lack of integration and the specific needs of the project (or domain) which usually requires a customization of the requirement method and its notation.

One of the main challenges for RE is to prove in practice the advantages of the proposed techniques, providing facilities for the integration and adaptation of RE technology to real-life projects. Each project has its specific needs and requires to select, integrate and customize suitable techniques to define its RE method. In this chapter, we have presented an approach based on metamodeling to offer such an integration and adaptability.

We have dealt with the two challenges, the diversity of approaches and the need of adaptation, by using metamodeling. However, one important problem when defining highly expressive models (which can handle a wide range of types of artifacts and/or dependencies) is achieving a consensus with respect to their semantics. In order to establish a global set of RE concepts and the required expressiveness, we have studied four representative techniques for requirements specification. The main features of our approach are:

- Definition of a metamodel that includes the core set of concepts that corresponds to the essential expressiveness of some of the most popular and/or advanced approaches in requirements engineering. It allows us to adapt and extend a core set of concepts keeping a suitable level of semantics consistence.
- Establishment of guidelines for adapting the metamodel to specific needs, according to the required expressiveness. In this way and according to the project specific needs, it is provided a proper integration as well as scalability from simpler up to other more sophisticated RE techniques.

We consider that our proposal constitutes a step forward in achieving a successful application of RE techniques in real-life projects. We have obtained a preliminary validation of our proposal through its application in the medium-size project EFTCoR with satisfactory results. In addition, we believe that our proposal provides the analyst with an additional advantage: traceability between different requirements specifications. Because any type of artifact and relationship can be described, it would be possible, for instance, to introduce specifications following a goal-oriented approach and its traceability to a viewpoint approach to analyze the specification from different perspectives and techniques.

Using the described proposal, the ATRIUM Goal Model has been defined. This Goal Model offers a relevant improvement for the specification of requirements: it integrates the advantages of three prominent and modern approaches: Goal-Oriented and Aspect-Oriented Requirements Engineering, and Variability Management.

The Goal-Oriented approach provides us with the necessary capabilities for analyzing architectural alternatives: the main aim of ATRIUM. It also facilitates the backward traceability because each architectural alternative is related to the requirements that have determined its definition. In addition, the use of the operationalizations allows a delayed decision of how to split the work between the system-to-be and the environment. This idea is highly powerful because it provides us with more flexibility to define different systems depending on our resources, time, etc., and, thus, following the (Lauesen, 2003)'s recommendations.

The introduction of concepts from the AORE field facilitates that the crosscutting inherent to any requirements specification can be properly specified and managed. This is also a step ahead to the proper management of the called quality requirements. It must be emphasized that, in our proposal, the *aspect* concept does not explicitly appear as a constructor, as in other works. Instead, the candidate aspects implicitly arise on those goals/requirements with crosscutting relationships. This is because the aspect concept is specified in another model of the ATRIUM approach, as will be described in the next chapter.

The introduction of the expressiveness for variability management was also compulsory in the description of the proposal. The EFTCoR has been the context that has motivated the definition of ATRIUM. This project exhibits specific needs in terms of product lines and dynamic architectures that must be specified just from the very beginning of the specification.

Another advantage that offers our proposal is the use of the ISO/IEC 9126 as a starting point to establish the possible concerns of the system-to-be. Additionally, it is possible to tailor, in terms of content, the SRS to the IEEE 830-1998 but with meaningful advantages for elaboration and organization of the requirement specification.

We have exploited our metamodeling proposal to provide customizable support in Goal Model propagation analysis. We have illustrated how the propagation rules can be customized according to the needs of expressiveness of the project. It has been used to assist in the process of selecting which architectural alternatives contribute more positively and with fewer conflicts to the satisfaction of the requirements of the system to be. It must be taken into account, that rules can be described for any kind of artifact or relationships. This means that its use is not only constrained to the satisfiability analysis but also aims that are more ambitious can be achieved with their application. For instance, they could be used to determine the propagation of changes in the specification. We consider that they are just a first step towards describing an analysis process which could be called concern-oriented, i.e., a process where the rules to be applied depend on the concern that is being considered.

In addition, it should be mentioned that this is not a theoretical proposal but a tool, called MORPHEUS, gives assistance along the process. It allows the analyst to describe both the Metamodel and its immediate use for modelling purposes. It also facilitates the proper support for describing the rules analysis and its later propagation.

In order to facilitate the specification and analysis of the ATRIUM Goal Model a detailed process has been defined. It details the set of steps to be performed, the guidelines that can help in the process, the exploitation of the ISO/IEC 9126, etc.

The work related to the definition of the metamodeling proposal and its use for the definition and exploitation of the ATRIUM Goal Model has been presented in the following publications:

- E. Navarro, P. Letelier, J. A. Mocholí, I. Ramos, “A Metamodeling Approach for Requirements Specification”, Journal of Computer Information Systems, 46(5): 67-77, Special Issue on Systems Analysis and Design, ed. Keng Siau.
- E. Navarro, P. Letelier, D. Reolid, I. Ramos, “Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques”, Information Systems Development Advances in Theory, Practice, and Education (to be published).

- E. Navarro, P. Letelier, I. Ramos, "Integrating Expressiveness of Modern Requirements Modeling Approaches", Proceedings 3rd International Conference on Software Engineering Research, Management & Applications (SERA 2005), Mount Pleasant, Michigan, USA, August 11 - 13, 2005, IEEE Computer Society, ISBN 0-7695-2297-1.
- E. Navarro, P. Letelier, I. Ramos, "Goals and Quality Characteristics: Separating Concerns", Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, collocated to OOPSLA 2004, Monday, October 25, 2004, Vancouver, Canada.
- E. Navarro, P. Letelier, I. Ramos, "UML Visualization for an Aspect and Goal-Oriented Approach", The 5th Aspect-Oriented Modeling Workshop (AOM'04), collocated to UML 2004 Conference, Monday, October 11, 2004, Lisbon, Portugal.
- E. Navarro, I. Ramos, J. Pérez, "Goals Model Driving Software Architecture", Proceedings 2nd International Conference on Software Engineering Research, Management & Applications (SERA 2004), Los Angeles, California, USA, May 5-17, 2004, ISBN 0-97007769-6, pp. 205-212.
- E. Navarro, P. Letelier, I. Ramos, "Un Marco de Trabajo para Integrar y Adaptar Múltiples Enfoques para Especificación de Requisitos", Jornadas de trabajo DYNAMICA, Archena, Spain, Novembre 17-18, 2005.
- E. Navarro, P. Letelier, I. Ramos, P. Sánchez, B. Alvarez, "Variabilidad en un marco de requisitos basado en orientación a objetivos", Jornadas de trabajo DYNAMICA, Almagro, Spain, April 21-22, 2005.
- E. Navarro, P. Letelier, I. Ramos, B. Alvarez, "Especificación de requisitos software basada en características de calidad, separación de concerns y orientación a objetivos", IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004), Málaga, Novembre 10-12, 2004.
- E. Navarro, P. Letelier, I. Ramos, B. Alvarez, "Orientación a aspectos y Orientación a objetivos: una propuesta para su integración", Desarrollo de Software Orientado a Aspectos, collocated to IX JISBD'2004, Málaga, Spain, Novembre 9, 2004.
- E. Navarro, P. Letelier and I. Ramos, ATRIUM, Arquitecturas Software a partir de Requisitos - El Modelo de Objetivos", Jornadas de trabajo DYNAMICA, Málaga, Spain, Novembre 11, 2004.

"Although nature commences with reason and ends in experience it is necessary for us to do the opposite, that is to commence with experience and from this to proceed to investigate the reason"— Leonardo da Vinci.

CHAPTER 6
Playing with ATRIUM Goal Models

6.1 INTRODUCTION

In order to offer a proper solution from the perspective of RE to the tele-operated domain, it was necessary to understand what problematic exists in this domain. For this reason, the document "Remote Control Unit Requirements" of the EFTCoR (RDCU, 2003) was the starting point to gain an overview of this kind of systems. This document exhibits requirements of both the tele-operated systems and the Robotic Devices Control Unit (RDCU, Figure 4-2). This study allowed us to identify several key points for the validation of our proposal. These points are described in the following:

– *Establishment of System Requirements*. It is necessary to take into account that while describing the tele-operated systems it is usual that several types of hardware/software components can be employed to satisfy the same requirements but with a different assignment of responsibilities hardware/software. Each one has different "qualities" in terms of performance, safety, and, specially, cost. This means that to delay the selection of this assignment to later stages provides the development process with greater flexibility. This leads to a decision: not to perform a premature decision of hardware or software requirement but to work with system requirements. This delayed decision about the elements to be included in the system-to-be is facilitated by means of the operationalizations. They describe how the requirements are met by the system-to-be. This means that while describing the requirements of the system, no decision about which services are to be performed by each part of the system or the environment is made.

- *Organization of the Specification*. The analysis of the RDCU document highlighted how tangled the requirements of the system were. This means that a proper organization of the specification was unavoidable for its appropriate exploitation. This point was crucial because of the high number of requirements of the EFTCoR. In this sense, (Lemos & Saeed, 1995) have highlighted that it is highly recommendable to organize the specification according to the operation mode of systems. This recommendation was also taken into consideration for the specification of the requirements.
- *Product families*. When the document describing the EFTCoR project was analysed, it was frequent to find generic requirements so that they facilitate the description of several specific systems. This kind of requirements is usually employed in the definition of product lines. This was the reason to perform the specification of the EFTCoR from this perspective, elaborating a specification for a family of products exploitable for the derivation of products. Some sentences, found in the reference documents, as for instance:

  The RCU should be adaptable to different positioning systems. OR/AND. Different RDCUs should be developed for the different positioning systems.

  The RDCU should be adaptable to different combinations of primary and secondary positioning systems. OR/AND. Different RDCUs should be developed for the primary and secondary positioning systems and they should be capable of working in a coordinated way.

It must be highlighted that EFTCoR is hardware intensive because they are to be formed by robotic units in charge of specific tasks. Software is usually used to control the behaviour of these robotic devices. Figure 4-2 sketches how EFTCoR is formed by five sub-systems. Cleaning Tools, Recycling System, Vision System, and Positioning Systems, both Primary and Secondary, are the mechanical components of the EFTCoR. The aforementioned RDCU integrates all the required functionality to manage the EFTCoR. We focused our efforts on the RDCU mainly because its Software Architecture is highly relevant to be compliance with the constraints described in section 4.2.1.

Figure 5-5 depicts the activities for the ATRIUM Goal Model. In the following sections, the accomplished specification of the EFTCoR is presented to provide a better understanding of both the process and the model. The full description of the RDCU is not presented in this section because it is very extensive and it would not help to the understanding of the work. This has motivated that only a partial view of the specification is introduced but providing enough details as to put into practice the proposal and comprehend its advantages.

6.2 ELICITATION AND SPECIFICATION

According to the process described in Figure 5-5, the first activity to perform was the *Elicitation&Specification*. This activity is oriented to describe the Goal Model and is unfolded in several steps (Figure 5-6) guiding this process. The first step was the identification of the concerns of the EFTCoR system using the ISO/IEC 9126. As was described in section 5.4.1(ISO/IEC 9126: Selecting and Identifying Concerns), five quality characteristics (*maintainability*, *performance* –efficiency in the ISO9126–, *reliability*, *safety* and *security*) are going to be selected to apply this process. This is because according to the (Bosch, 2000)'s recommendations they have the greatest impact on the SA. In addition, *suitability* is also employed along the process because the user objectives and needs are going to be described under this characteristic.

The following sections introduce the description that was performed for each of the selected quality characteristics and sub-characteristics. It will facilitate a better guidance of the explanation and a proper organization of the specification. In addition, it must be underlined that the step *Operationalization* is especially meaningful for the ATRIUM aim: Software Architecture description. For this reason, it is presented at the end of this section.

6.2.1 Functionality

Suitability

The ISO/IEC9126 describes this software quality as:

"The capability of the software product to provide an appropriate set of functions for the specified tasks and user objectives."

Taking into account this definition, it seems obvious that the description of the main goals of the EFTCoR had to be defined as refinements of this characteristic. Considering the described patterns we could state that one of the goals of the system would be:

GOAL	
NAME	GOA.1
DESCRIPTION	RDCU should be suitable for the user needs
PATTERN [6]	achieve
CONCERN	suitability
PRIORITY	High
AUTHOR	Elena Navarro
CREATIONDATE	10/10/2006

In order to obtain the goals of the EFTCoR, the (EFTCOR, 2003) document was used as available information. Several works for dealing with robotic specifications were also employed as an input for the process. Especially, it was taken into account the (Lemos & Saeed, 1995)'s recommendations to split the specification according to the operation modes of systems. During the elaboration of the specification, six operation modes were detected: *working*, *calibration*, *learning*, *diagnosis* and *configuration* (these four are *maintenance* modes) and *safe stop*. This means that six goals were described as refinements of the one described above. For a better understanding of the case study, we only include the Working Mode. This means that all the goals, which are described in the following, are catalogued according to this operation mode. Figure 6-2[7] shows how these sub-goals are refining GOA.1 by means of an AND refinement relationship. This means that all of them must be met by the system-to-be. All of them use the pattern *Achieve* because they describe a property to be eventually held by the goal.

GOAL	
NAME	GOA.2
DESCRIPTION	RDCU allows working operation
PATTERN	achieve
CONCERN	suitability
PRIORITY	High

[6] In the following, both AUTHOR and CREATIONDATE are not described because they are not going to be exploited for the process. It can also be observed that the formalization of the Goals was not considered either.

[7] In the following, the graphical notation is used instead of the textual one.If any attribute, such as pattern or concern, changes, the textual notation will be introduced

GOAL	
NAME	GOA.3
DESCRIPTION	RDCU allows safe stop
PATTERN	achieve
CONCERN	suitability
PRIORITY	High

GOAL	
NAME	GOA.4
DESCRIPTION	RDCU allows maintenance operation.
PATTERN	achieve
CONCERN	suitability
PRIORITY	High

GOAL	
NAME	GOA.5
DESCRIPTION	RDCU allows learning
PATTERN	achieve
CONCERN	suitability
PRIORITY	High

GOAL	
NAME	GOA.6
DESCRIPTION	RDCU allows calibration
PATTERN	achieve
CONCERN	suitability
PRIORITY	High

GOAL	
NAME	GOA.7
DESCRIPTION	RDCU allows diagnosis
PATTERN	achieve
CONCERN	suitability
PRIORITY	High

GOAL	
NAME	GOA.8
DESCRIPTION	RDCU allows configuration
PATTERN	achieve
CONCERN	suitability
PRIORITY	High

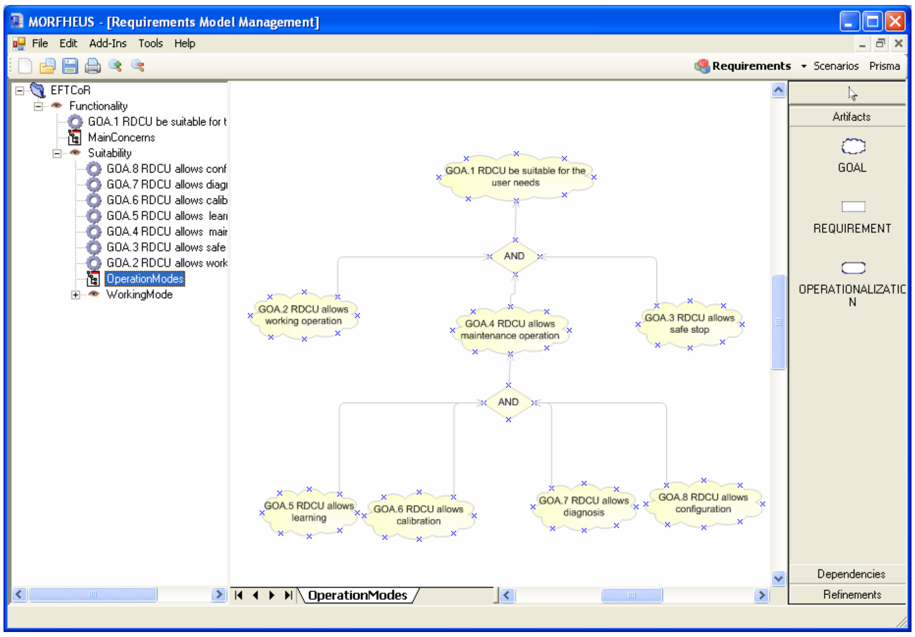

MORFHEUS - [Requirements Model Management]
File Edit Add-Ins Tools Help
Requirements Scenarios Prisma
EFTCoR
Functionality
GOA.1 RDCU be suitable for t
MainConcerns
Suitability
GOA.8 RDCU allows conf
GOA.7 RDCU allows diag
GOA.6 RDCU allows calib
GOA.5 RDCU allows lear
GOA.4 RDCU allows mai
GOA.3 RDCU allows safe
GOA.2 RDCU allows work
OperationModes
WorkingMode
GOA.1 RDCU be suitable for the user needs
AND
GOA.2 RDCU allows working operation
GOA.4 RDCU allows maintenance operation
GOA.3 RDCU allows safe stop
AND
GOA.5 RDCU allows learning
GOA.6 RDCU allows calibration
GOA.7 RDCU allows diagnosis
GOA.8 RDCU allows configuration
OperationModes
Artifacts
GOAL
REQUIREMENT
OPERATIONALIZATIO N
Dependencies
Refinements

Figure 6-1 Refining the GOA.1 RDCU should be suitable for the user needs.

Considering the EFTCoR document, we wondered *why* the EFTCoR was necessary, and as a result, two new goals emerged refining the goal GOA.2. Figure 6-2 shows how the refinement proceeds.

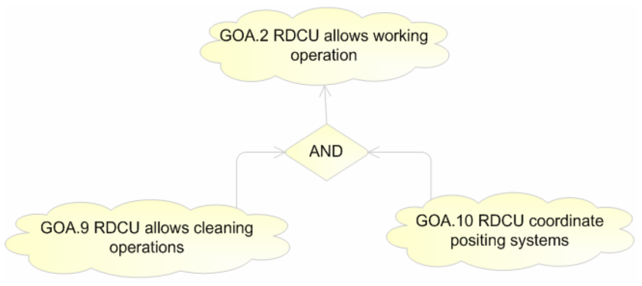

GOA.2 RDCU allows working operation
AND
GOA.9 RDCU allows cleaning operations
GOA.10 RDCU coordinate positing systems

Figure 6-2 Refining GOA.2 RDCU allows working operation

Related to GOA.2, it is clear that the main goal to be achieved by the RDCU is to be able to clean the hull surface. While this cleaning is performed, several operations can be performed, such as: fresh water washing previous to blasting and painting after blasting. This means that a family of tools can be attached to the RDCU to either perform different operations (i.e. blasting or painting) or the same operations in a different way (i.e. coating removal by blasting or by pressured water). This means that the previous GOA.9 could be refined in several goals. It can be observed that an OR relationship was used to describe the refinement so that GOA9 is to be satisfied if any one of them is satisfied. This means that at least one of the goals must be met by the system-to-be. Taking into account the description of an OR, presented in Table 5-4, two attributes were described: *min* that was set to 1; and, max that was set to 5, i.e., the maximum number of alternatives.

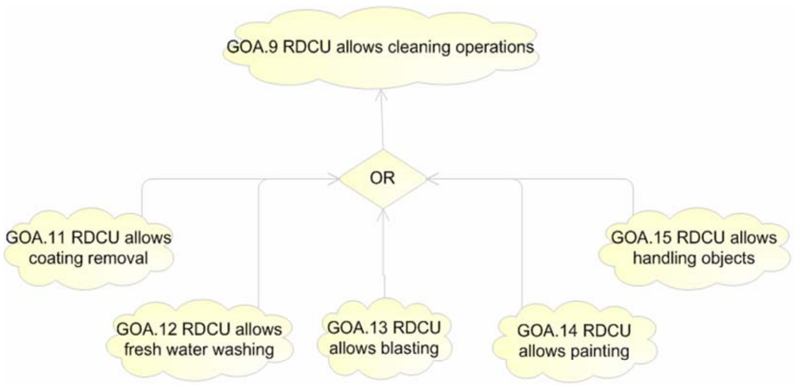

GOA.9 RDCU allows cleaning operations
OR
GOA.11 RDCU allows coating removal
GOA.15 RDCU allows handling objects
GOA.12 RDCU allows fresh water washing
GOA.13 RDCU allows blasting
GOA.14 RDCU allows painting

Figure 6-3 Refining GOA.9 RDCU allows cleaning operations

When this level of refinement is achieved, requirements can be described for the RDCU, that is, goals which are verifiable in the system-to.-be. In this sense, the GOA.15 could be refined as described in Figure 6-4. Considering that it describes the management of objects, the robot could open and close the tool attached to it to manage these objects, for this reason, REQ.1 and REQ.2 were described.

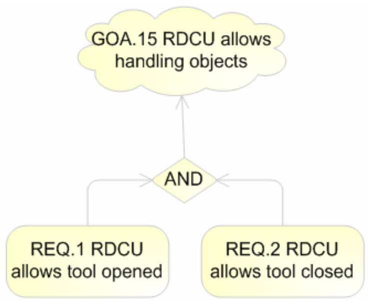

GOA.15 RDCU allows handling objects
AND
REQ.1 RDCU allows tool opened
REQ.2 RDCU allows tool closed

Figure 6-4 Refining GOA.15 RDCU allows handling objects

GOA.10 states that RDCU should coordinate positing systems. It must be considered that EFTCoR is a family of robots that should be able to work with a wide diversity of hulls such as tankers (ships designed to carry bulk liquids, particularly oil) or frigates (ships smaller and faster than a ship-of-the-line, used for patrolling and escort work). This diversity also implies that the EFTCoR should be able to move in different areas such as vertical surfaces or bows (the foremost point of the hull of a ship or boat). For this reason, it seems obvious that the EFTCoR should be able to move across wide surfaces but also to make precise movements.

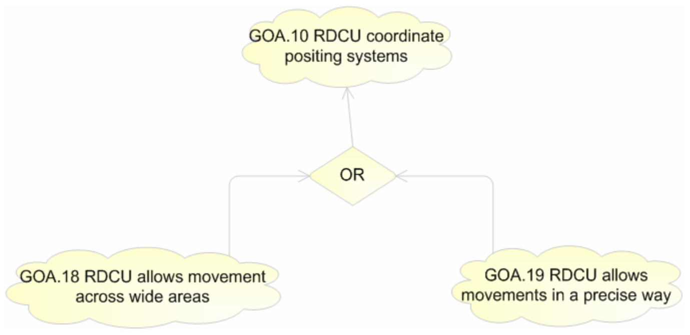

GOA.10 RDCU coordinate positing systems
OR
GOA.18 RDCU allows movement across wide areas
GOA.19 RDCU allows movements in a precise way

Figure 6-5 Refining GOA.10 RDCU allows coordinate positioning systems

GOA.11 describe that EFTCoR has to move across wide areas, although they can be either horizontal, such as the bows or the dockyard, or vertical. This means that the following goals can be described:

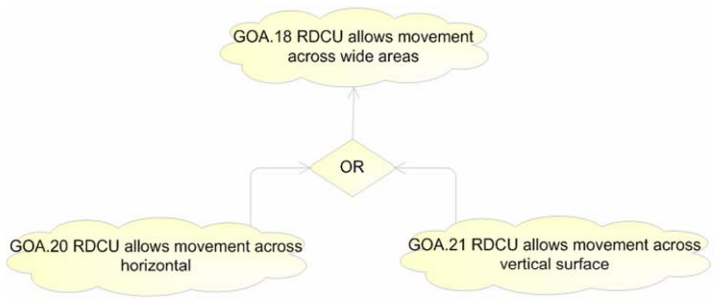

GOA.18 RDCU allows movement across wide areas
OR
GOA.20 RDCU allows movement across horizontal
GOA.21 RDCU allows movement across vertical surface

Figure 6-6 Refining GOA.18 RDCU allows movement across wide areas

As described in Figure 6-7, GOA.19 can be refined. But, similarly to GOA.15 (Figure 6-4), it is refined into requirements. We should consider that in EFTCoR systems, when dealing with such kind of precise movements, the secondary system is employed. The requirements, described in Figure 6-7, are introduced to facilitate the movement of the secondary system, mainly, to bring the tool closer to the required area. REQ.6 and REQ.8 are directly related to bring the tool closer and farther of the area being treated. The secondary system is integrated by several joints, for this reason, REQ.3 and REQ.9 are described to facilitate the movement of a specific joint either using a delta increment or an angular increment. Because most of the movements are described as an increment from the current position, REQ.7 describes the capability to set the start point for a joint. It is also possible to describe, co-ordinately, the arm to an angular destination as REQ.4 prescribes. Independently of the movement being performed, it must be possible for the RDCU to stop such movement, as REQ.5 states.

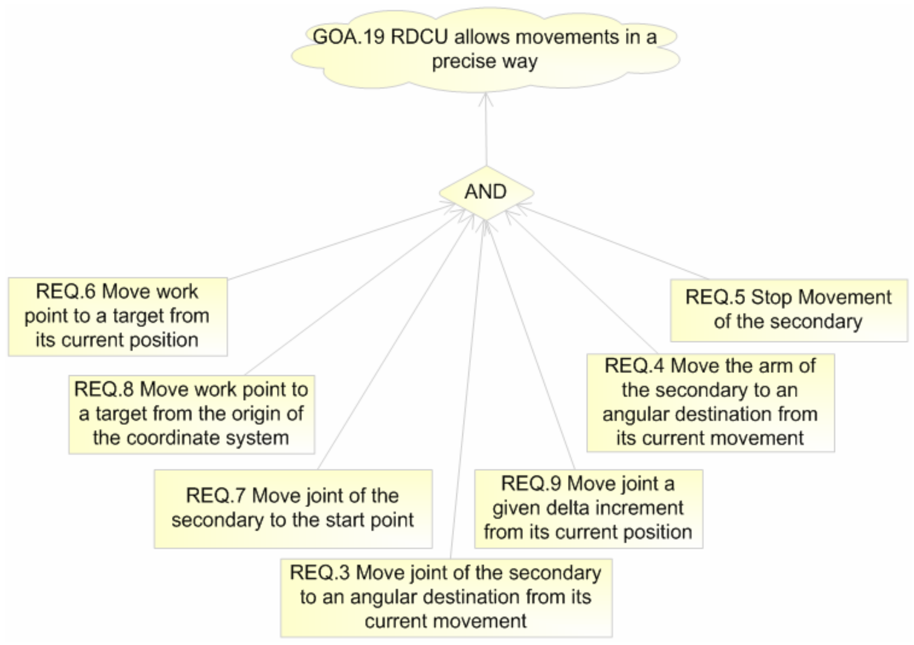

GOA.19 RDCU allows movements in a precise way
AND
REQ.6 Move work point to a target from its current position
REQ.5 Stop Movement of the secondary
REQ.8 Move work point to a target from the origin of the coordinate system
REQ.4 Move the arm of the secondary to an angular destination from its current movement
REQ.7 Move joint of the secondary to the start point
REQ.9 Move joint a given delta increment from its current position
REQ.3 Move joint of the secondary to an angular destination from its current movement

Figure 6-7 Refining GOA.19 movements in a precise way.

Security

The ISO/IEC9126 describes the `Security` as:

> *"The capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them."*

The reference document was analysed to gather the goals and requirements related to this sub-characteristic. However, no details or comments were detected indicating any necessity in these terms. It must be taken into account that this kind of systems operate in areas where the access is restricted to authorised personnel. This means that there are physical mechanisms responsible for these tasks. This has motivated that no goal or requirement was included under this sub-characteristic.

6.2.2 Reliability

The ISO/IEC9126 describes the `Reliability` as:

> *"The capacity of the software product to maintain a specified level of performance when used under specified conditions."*

This characteristic is split into threes sub-characteristics:

> `Maturity`. *"The capability of the software product to avoid failure as a result of faults in the software"*
>
> `Fault Tolerance`. *"The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface."*
>
> `Recoverability`. *"The capability of the software product to re-establish its level of performance and recover the data directly affected in the case of a failure."*

While analyzing the (RDCU, 2003) `Availabity` emerges as an important need for the RDCU and is described in this document as:

> *"The proportion of time the system is up and running."*

In the case of the RDCU, the practitioners stated that it admissible a large rate of failures if the mean time to repair is short. However, the ISO/IEC 9126 describes that *availability* is a combination *maturity*, *fault tolerance* and *recoverability*. This means that the need of *availability* can be described in terms of *recoverability* for the RDCU. Therefore, the availability has been specified by means of these three sub-characteristics.

Taking the above into account, several goals for the RDCU can be described as refinements of maturity, fault tolerance and recoverability. GOA.28 states that the RDCU must be available to work during hundreds of hours. GOA.29 describes that in case of failure the RDCU repair time should be as short as possible; it must be highlighted that a high rate of failures is allowed if the repair time is short. GOA.30 describes that the RDCU admits degraded modes of operations that allows operators to perform the maintenance operations in case of partial failures of the RDCU or failures of the external systems connected to the RDCU. This goal can be refined into REQ.11 that describes that it should be possible to separately operate each joint of the system if the system fails.

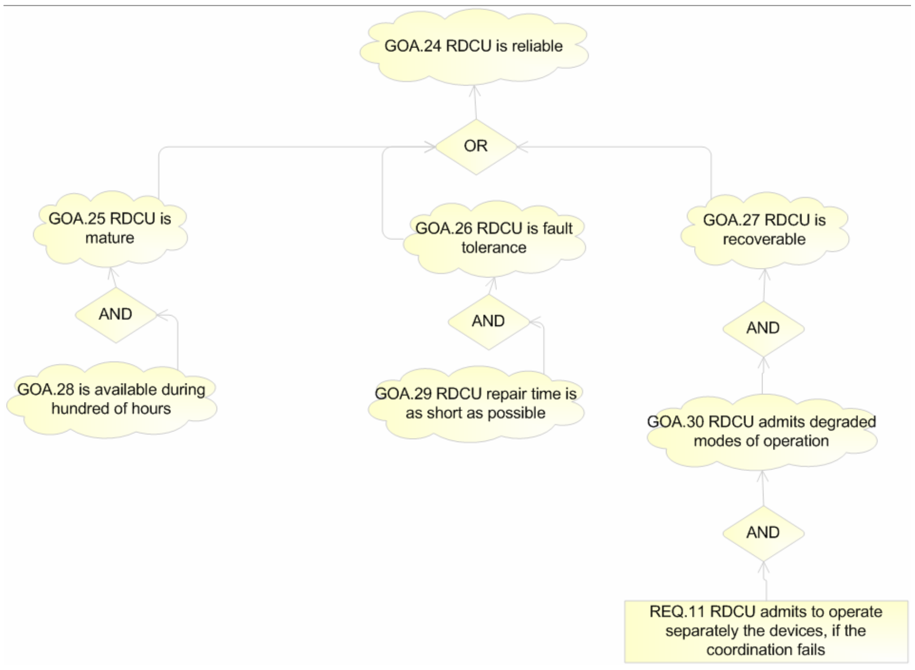

GOA.24 RDCU is reliable
OR
GOA.25 RDCU is mature
GOA.26 RDCU is fault tolerance
GOA.27 RDCU is recoverable
AND
AND
AND
GOA.28 is available during hundred of hours
GOA.29 RDCU repair time is as short as possible
GOA.30 RDCU admits degraded modes of operation
AND
REQ.11 RDCU admits to operate separately the devices, if the coordination fails

Figure 6-8 Describing Reliability goals of the RDCU

6.2.3 Efficiency

The ISO/IEC9126 describes `Reliability` as:

> *"The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions."*

This characteristic has been split into two sub-characteristics:

> *"Time behaviour. The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions."*

> *"Resource utilisation. The capability of the software product to use appropriate amounts and types of resources when the software performs its functions under stated conditions."*

In the case of the EFTCoR, only *Time behaviour* can be established as a goal for the RDCU. In general terms, it is required that the time that a ship can stay in the shipyard should not be longer than the time that it stays using the current maintenance methods. It means that the ship should not be in the shipyard if any operation is to be performed. This general goal of the EFTCoR must be translated to the goals and requirements of the RDCU.

Requirements related to *time behaviour* for the RDCU come from the operations of maintenance, that is, the positioning of the cleaning tools and their task are those to constraint the time required for the system, but not the time required for the RDCU. This means that no concrete requirements neither goals have been established as refinement of *Performance*.

6.2.4 Maintainability

The ISO/IEC9126 describes the `Maintainability` as:

The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

There were no specific goals neither requirements related to this characteristic, but to the next one: Portability. This was because the analysed documents refer to changes in the functionality to adapt the system to specific needs.

6.2.5 Portability

The ISO/IEC9126 describes the `Portability` as:

"The capability of software product to be transferred from one environment to another."

There is a sub-characteristic, called `Adaptability`, which is highly important for the RDCU description:

"The capacity of the software product to be adapted for different specified environments without applying actions or means other that those provided for this purpose for the software considered."

This sub-characteristic is highly relevant during the application of ATRIUM. As was previously stated, the variability that must be resolved at run time must be described as a goal o requirement described as a refinement of adaptability. It does not only provide a proper organization of the specification, but also an important guidance for the selection of operationalizations. These operationalizations are going to instantiate architectural patterns that provide support for variability at run time.

Therefore, both the portability characteristic and the adaptability sub-characteristic can be stated as goals GOA.32 and GOA. 33 of the RDCU shown in Figure 6-9. As can be observed, GOA.33 has been defined by means of an OR refinement, because it can be present or not in the RDCU, depending on the specific needs of the instance of the EFTCoR to be developed.

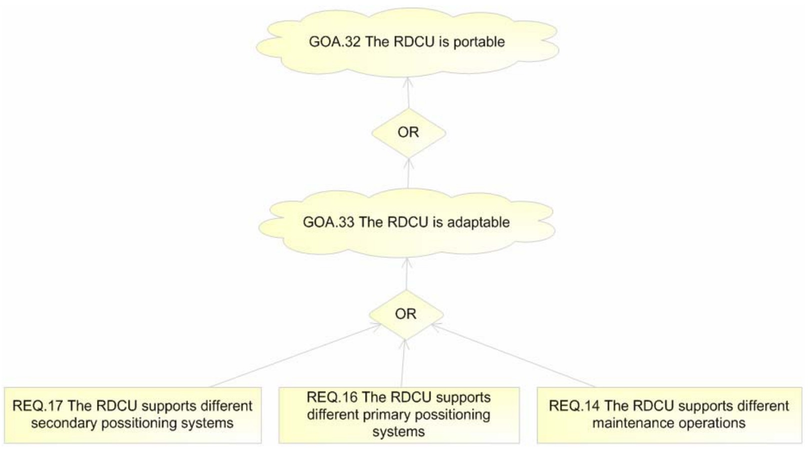

GOA.32 The RDCU is portable
OR
GOA.33 The RDCU is adaptable
OR
REQ.17 The RDCU supports different secondary possitioning systems
REQ.16 The RDCU supports different primary possitioning systems
REQ.14 The RDCU supports different maintenance operations

Figure 6-9 Describing RDCU goals related to portability

The adaptability required to the RDCU is mainly focused on changes in the functionality of the EFTCoR, that is, maintenance operations. For this reason, three requirements have been introduced: REQ.14 describes the possibility of using the RDCU to control different maintenance tasks; and, REQ.16 and REQ.17 describe that the RDCU should be able to support different combinations of primary and secondary positioning systems, respectively. They have been introduced by means of an OR relationship because it can be present or not in final system. In addition, these requirements have a *Require* relationship with two variation points. Figure 6-10 depicts the relation between the variant described by the REQ.14 and the variation point described by the GOA.9. This means that if the system provides support for adaptability of different maintenance operations, it requires providing support for the different alternatives already described. However, this support must be given at run time because REQ.14 is a requirement of adaptability.

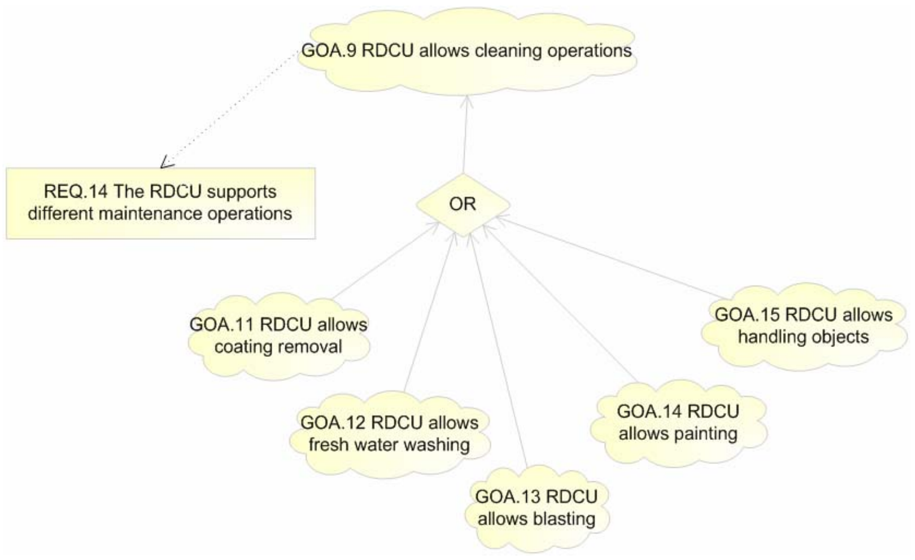

GOA.9 RDCU allows cleaning operations
REQ.14 The RDCU supports different maintenance operations
OR
GOA.11 RDCU allows coating removal
GOA.12 RDCU allows fresh water washing
GOA.13 RDCU allows blasting
GOA.14 RDCU allows painting
GOA.15 RDCU allows handling objects

Figure 6-10 Describing a Require relationship between a variant REQ.14 and a variation point GOA.9

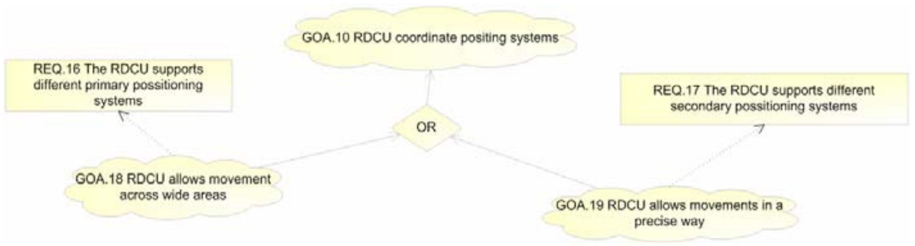

GOA.10 RDCU coordinate positing systems
REQ.16 The RDCU supports different primary possitioning systems
REQ.17 The RDCU supports different secondary possitioning systems
OR
GOA.18 RDCU allows movement across wide areas
GOA.19 RDCU allows movements in a precise way

Figure 6-11 Describing Required relationships between variants REQ.14 and REQ.17 and variants GOA.19 and GOA.18

Figure 6-11 shows how a *Require* relationship is established between variants. This means that to provide support for different primary and secondary systems, the system must support the movements across wide areas and in a precise way.

6.2.6 Safety: being one step ahead

Robotics systems are substantially different from other software applications because it is mandatory to consider aspects such as interaction with the environment, presence of perturbations, etc. Moreover, if their use has an important impact on persons and equipment when errors arise then safety aspects must be considered during their development. Besides the risks inherent to their use, make work places higher risk areas. (Douglass, 2003) gives, among others, the following list of damage sources: errors in the execution of the control system (hardware and software), people who access to forbidden walking areas, human errors, broken mechanical parts, liberation of stored energy, and so on.

Nowadays, when we analyze the development of tele-operated systems, we can observe that there is no well-known integration of safety requirements within the process of requirements specification as a whole and there are no methodologies supporting it in an integrated way.

It must be noticed that one of the most popular approaches to identify, evaluate and manage safety requirements is the technique named `fault trees` (Hansen et al., 1998). These trees provide a graphical notation and a formal support that facilitate the analysis from the perspective of the system failures and their causes. Nevertheless, they do not offer a global framework for requirement specification as a discipline. Despite this deficiency, they are widely used to analyze safety specifications. From the point of view of requirement refinements, our proposal is analogous to the use of fault trees; however, in our work the analysis of safety requirements is integrated and derived from the set of functional requirements of the system.

(Letier & Lamsweerde, 2002) have also proposed the use of KAOS for safety related requirements specification. They have introduced the concept of `obstacle` as *a set of non-desirable behaviours*, the presence of those obstacles imply the obstruction in the fulfilment of the objective. At the same time, the negation of the obstacle generates the preconditions needed for the satisfaction of the requirements. The safety goals of the system being developed are formally specified by using temporal logic, and the obstacles (similar to hazards) are automatically obtained by the negation of the safety objectives and following the patterns given in their proposal (Lamsweerde & Letier, 2000). However, the KAOS proposal does not provide a specific process for dealing with safety goals in the context of safety requirement specifications, nor does it consider factors such as severity, exposition time, etc, to be exploited during the analysis of the safety specification.

In order to cope with these deficiencies, we have considered the *goal oriented* ATRIUM framework for guiding the requirement engineering process. This framework has been extended for considering *safety* requirements by following the ANSI/RIA R15.06-1999 standard (ANSI/RIA, 1999). Furthermore, our proposal has been enriched both with the ideas by (Lemos & Saeed, 1995) for the division of operation modes of systems, and the patterns and heuristics given by (Douglass, 2003) for the consideration of this kind of requirements. It is mandatory in our proposal because the kind of system described in the EFTCoR, by its nature, entails a greater probability of danger. Thus, a precise identification, specification and trace of safety requirements turn out essential.

In order to clarify the following discussion, the main used concepts are defined in the following. Hence, it is mandatory to define which meaning of `Safety requirement` has been used from the set of definitions in the bibliography. ANSI/RIA R15.06-1999 standard, main basis for this work, defines Safety requirements as:

> *"those to be satisfied for any industrial robotic system to assure the safety of personnel associated with its use."*

In this work, this definition has been extended, with (Leveson, 1995)'s ideas, by including damage or destruction of property or injury or damage to any living being, especially, human beings.

When the behaviour of a system is being described, `tasks` to be provided by any system component, whether software or hardware, have also to be described. The execution of these tasks is the potential source of damage or injuries the system can cause, so that the most serious risks for safety arise from deficiencies of functionality, reliability or usability as ISO/IEC 9126 standard (ISO/IEC 9126) states.

The early detection of `Hazards` is a challenge during the specification of Safety requirements. A hazard is any potential source of damage or injury to an entity of the system, from an operator to an entity of the environment or the system itself. In this way, during the gathering and specification of Safety requirements, the analyst has to identify the likelihood of the system hazards and analyze them to determine which strategy is the most appropriate for their management. For this analysis, the `risks` related to each hazard have to be established, i.e., the damage or injury to any entity.

These concepts indicate that, although the process of gathering and identifying this kind of requirements is quite similar to those applied in other contexts, it does show some meaningful differences. Furthermore, the fact that tele-operated systems are Safety Critical emphasizes how important it is to provide the analyst with both a specific process and a specific notation.

Identifying and Specifying Safety Requirements

Both the ANSI/RIA R15.06-1999 standard and the approaches proposed by Douglass and Lemos have been integrated to establish a process for identification and specification of Safety Requirements in Tele-operated environments. The Goal Model of ATRIUM is the notation used during the process. This model exploits the standard ISO/IEC 9126 as a starting point to organize the requirements specification.

The established process entails several steps. The first ones (i-ii) are related to the behaviour specification of the system:

i. *To identify system operation modes*, according to Lemos' recommendations for control systems. These operation modes are specified as system goals and have a refinement relationship towards the characteristic ISO/IEC 9126 Suitability. Considering these system goals, an intentional refinement process is triggered, as was described in the section 5.4.1, repeatedly until goals have enough granularities to allow the identification of tasks, i.e., requirements to be met.

ii. *To identify Tasks* (T_i) *associated to each operation mode.* In the Goal Model, these tasks are specified as requirements which have an AND/OR refinement relationship towards their parent goal. In addition, it provides an improved visibility of which hardware/software components of the system are involved in which operation modes because of their traceability to operationalizations, i.e., it provides scenarios that describe how these components interact to fulfil a requirement.

Both the identification of operation modes and tasks provide the system behaviour specification. This identification must be performed before Safety requirements are identified, so that no injury or damage is caused when the system is performing these tasks. Once this view of the system has been established, Safety requirements are determined by using the ISO/IEC 9126 Safety category. Therefore, the following steps are added to the process:

iii. *Determine Safety Goals of the system.* For each identified task T_i its capacity to cause any damage is evaluated. If this happens, a Safety goal, *Safe* (T_i), will be specified as *to safeguard* T_i. This implicitly means a crosscutting relationship between a Safety requirement *Safe* (T_i) and a Functional requirement T_i at the task level.

iv. *Determine System Hazards.* For each safety goal *Safe* (T_i) its related hazards are identified and specified as a sub-goal *Avoid* (Hz_j). An AND/OR relationship is established between *Safe* (T_i) and its set of hazards to be managed. The relation to apply depends on whether the whole set of hazards or some of them, respectively, have to be avoided to safeguard the task T_i. We have to bear in mind that the same hazard can be specified as a refinement of several Safety goals. In addition, according to Douglas' evaluation of risks, this rationale also entails some necessary information for the risk specification: identification of causes (software, hardware, human, …) which can result in a hazardous situation, reaction required for its management and maximum deadlines of exposure, detection and tolerance.

v. *Identify Risks* (R_k) *associated to each pair Safe* (T_i)— *Avoid* (Hz_j). The associated risk R_k has to be identified, so that the appropriate strategy for the management of Hz_j is selected. A Hazard can be related to several risks depending on which task is to be evaluated.

vi. *Determine the Risk Reduction Category (RRC) to apply to each relation Safe* (T_i)— *Avoid* (Hz_j), taking into account its associated risks R_1 … R_n. RRC determines which actions must be performed in order to properly manage the hazard. With this aim, the following three attributes must be firstly evaluated:

  a. *Severity*. Level of damage that an entity of the environment or the system itself can suffer. The table which is provided by ANSI/RIA R15.06-1999 for evaluation has been modified so as to deal with the widest sense of the term, i.e., including not only damages to the health or the environment but also to the system itself. Therefore, the meanings associated to both the S1 and S2 categories are described now as follow.

Level	**Description**
S2	Serious injury to the operator requiring more than first-aid. Damage of a system component which is irreplaceable both in time and cost.
S1	Serious injury to the operator only requiring first-aid. Damage of a system component which is replaceable both in time and cost

  b. *Exposure*. Frequency of exposure to the hazard. ANSI/RIA R15.06-1999 defines two categories: *E2* as *frequent* and *E1* as *infrequent*.

  c. *Avoidance*. Likelihood of avoiding the exposure to the hazard. ANSI/RIA R15.06-1999 defines two categories: *A2* as *not likely* and *A1* as *likely*

ANSI/RIA R15.06-1999 describes eight combinations of these values, which are specified in Table 6-1, as Risk Reduction Categories (RRCs) along with the recommended actions to manage properly the Hazard. It can be observed that there are actions such as *eliminating* the Hazard, i.e., it is not allowed that such a Hazard can emerge while the system is working; *preventing* that it can arise, etc.

Table 6-1 Risk Reduction Categories

Exposure	Avoidance	Severity	RRC	Action on Hz
E2	A2	S2	R1	Eliminate/Substitute
		S1	R2C	Prevent/Cease
	A1	S2	R2A	Cease
		S1	R3A	Isolate
E1	A2	S2	R2B	Prevent/Cease
		S1	R3A	Isolate
	A1	S2	R3B	Isolate
		S1	R4	Warning/Training/Protect

It must be highlighted that the Metamodel presented in Figure 5-3 cannot be used as it is. It is not expressive enough to describe the risk associated to each pair Safe (Ti)—Manage (Hzj), or the severity, exposure and avoidance necessary to determine the RRC. This means that an extension, which is shown in Figure 6-12, using the process of section 5.2.2, had to be accomplished. In the following it is described how each step was performed to extend the Metamodel:

I. A new type of artifact, *Hazard*, was specified to describe any potential source of damage or injury to an entity of the system, from an operator to an entity of the environment or the system itself, to deal with the step (iv).

   Another type of artifact, *Risk*, was also included to specify any damage or injury an entity of the system can suffer when a Hazard arises.

II. Both *AND* and *OR* refinement relationships were specialized, as *ANDsafety* and *ORSafety*, because it was necessary to describe some attributes applicable to the link between the leaves (hazards to manage) and the refinement relationship. In this way, it is provided the expressiveness described in the steps (v-vi).

III. Several attributes were added to the description of both artifacts and relationships:

   a. The enumeration *PatternType* was extended to include *safe* as another available pattern for the specification of goals.

   b. Several attributes were included in the description of *Risk* to consider its cause, expected reaction, and maximum exposure, detection and tolerance allowed.

   c. For each pair Safe(T_i)—Avoid(Hz_j), the attributes severity, exposure, avoidance and risk were included by means of the extension of *Leaf* as *LeafSafety*. It must be taken into account that they cannot be established on Avoid (Hz_j) because, depending on the task being performed different risks can arise. In addition, the RRC was also included as an attribute of this link because several RRC are going to be established for each Avoid(Hz_j), one for each task it is related to.

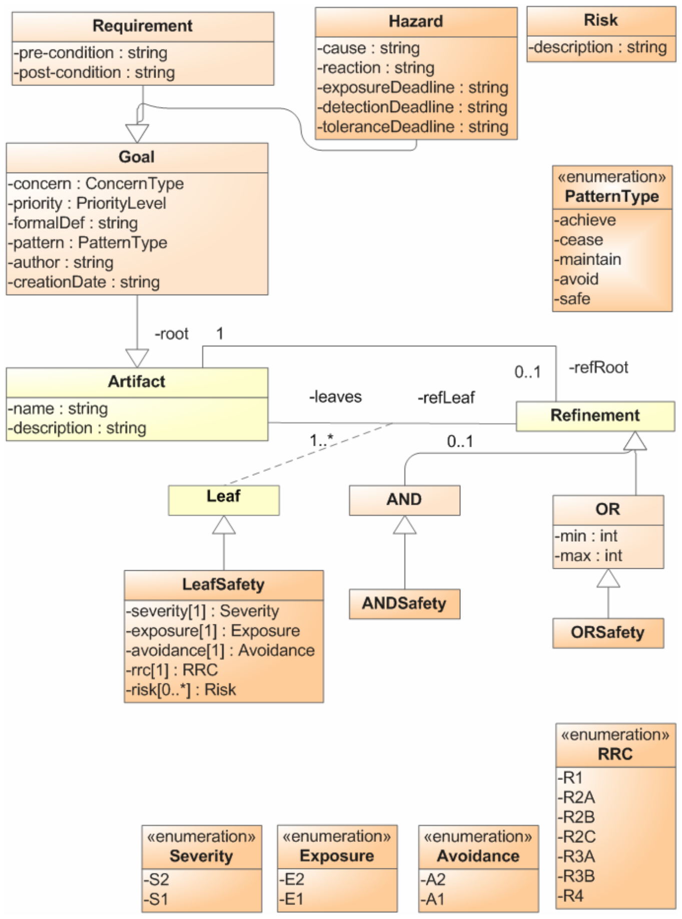


Figure 6-12 Extension to the ATRIUM Goal Model

IV. Some additional constraints were also introduced to describe which kinds of relations are allowed. In addition, *Leaf* was extended to describe the necessary information, i.e., severity, exposure, etc. They are described in the following table.

Relation	**Constraint**
ANDSafety	context ANDSafety inv

	Self.root.concern =“Safety” **and** Self.leaves->forAll(a:Artifact | a.concern =“Safety”)
ORSafety	context ORSafety inv Self.root.concern =“Safety” **and** Self.leaves->forAll(a:Artifact | a.concern =“Safety”)
LeafSafety	**context** LeafSafety **inv** Self.refLeaf..oclIsTypeOf(ANDSafety) **or** Self.refLeaf..oclIsTypeOf(ORSafety)

Hence, the evaluation of the values (Severity, Avoidance, Exposure) is carried out on each pair Safe (T_i)—Avoid (Hz_j) using as an input its associated risks R_k. According to the RRC an action has to be selected to eliminate, substitute, prevent, isolate or cease the Hz_j. If different RRCs are applicable for the same pair Safe (T_i)—Manage (Hz_j), the severest will be the selected one to apply the recommended action. These actions mean that different operationalizations can be included in the Goal Model to incorporate safety mechanisms; other can be modified to prevent or cease a hazard; or, even they can be totally eliminated because they are not appropriated for the safety levels that are necessary for the system-to-be.

The application of the actions, determined by the RRC, can imply that some hazards can be eliminated, risks can be avoided, the severity of the damage can change, etc. Therefore, it is mandatory to re-evaluate the set of RRC for the Goal Model to analyze those changes. This procedure must be repeated until all the hazards are considered “tolerable”, i.e., risk level for the system is acceptable. These hazards are to be considered residual risks of the system.

Applying the process

This section illustrates how the stated proposal was put into practice to gather the safety requirements of the industrial project EFTCoR. In the section 6.2.1, the operation modes of the system were already established, along with the behaviour of the system to be safeguard. For this reason, the step (iii) can be now applied. With this aim, those tasks to be safeguarded are specified as Safety goals, *Safe* (T_i), of the system.

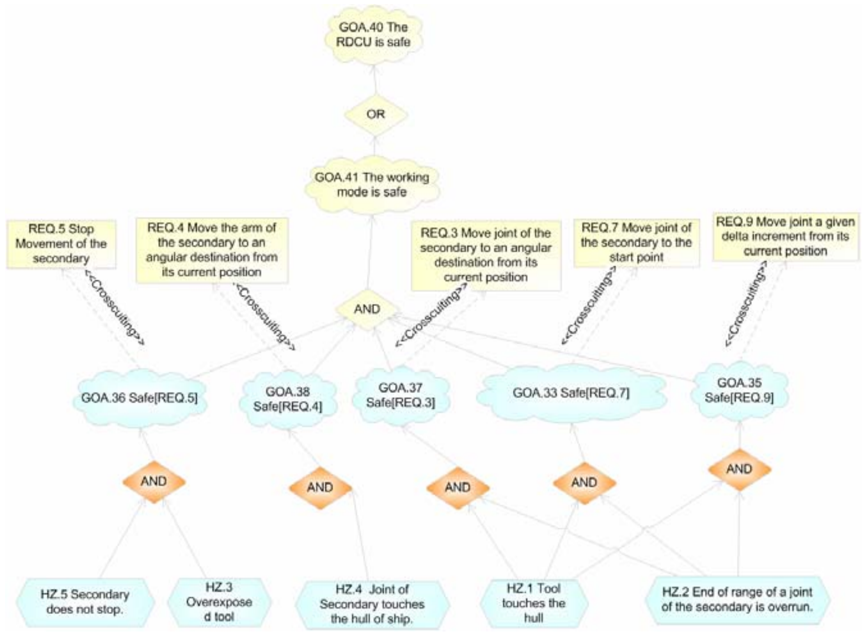

GOA.40 The RDCU is safe
OR
GOA.41 The working mode is safe
REQ.5 Stop Movement of the secondary
REQ.4 Move the arm of the secondary to an angular destination from its current position
REQ.3 Move joint of the secondary to an angular destination from its current position
REQ.7 Move joint of the secondary to the start point
REQ.9 Move joint a given delta increment from its current position
<<Crosscutting>>
AND
GOA.36 Safe[REQ.5]
GOA.38 Safe[REQ.4]
GOA.37 Safe[REQ.3]
GOA.33 Safe[REQ.7]
GOA.35 Safe[REQ.9]
HZ.5 Secondary does not stop.
HZ.3 Overexpose d tool
HZ.4 Joint of Secondary touches the hull of ship.
HZ.1 Tool touches the hull
HZ.2 End of range of a joint of the secondary is overrun.

Figure 6-13 Describing safety goals

As can be observed in Figure 6-13, the goal GOA.40 is established to describe safety as a concern of the RDCU. It is refined in several goals, for the different operation modes, as for instance the GOA.41 describes that the working mode must be safe. Considering this goal, each requirement related to that operation mode was analysed to determine if its realization could have associated hazards. If it was the case, it means that they had to be safeguarded. In Figure 6-13, each safeguarded requirement is established as a goal *Safe* (T_i). It can be observed that a crosscutting relationship has also been established between the safety goals and the suitability goals in order to describe that their behaviour must be constrained by the safety requirements.

Once the requirements to safeguard were established, the hazards Hz_j of the RDCU were determined by applying step (iv). Figure 6-13 shows that they were described as sub-goals *Avoid* (Hz_j) refining, by means of *ANDSafety* relationships, those requirements to safeguard. Therefore, a relationship exits between *Safe* (T_i) and *Avoid* (Hz_j) because during the execution of T_i is when Hz_j can arise. As can be observed, the same hazard can be related to several tasks to safeguard. A summary of the hazards that can arise, when the RDCU is controlling the Secondary Positioning Subsystem, is shown in the following.

HAZARD	
NAME	HZ.1
DESCRIPTION	Tool touches the hull of the ship
PATTERN	avoid
CONCERN	safety
PRIORITY	High
SOURCE	Breakage or error in the control of approach of the tool
REACTION	To stop the joint and separate the tool

HAZARD	
NAME	HZ.2
DESCRIPTION	End of range of a joint of the secondary is overrun.
PATTERN	avoid
CONCERN	safety
PRIORITY	High
SOURCE	Breakage of the sensor of end of range or error of control software
REACTION	To stop the electrical supply

HAZARD	
NAME	HZ.3
DESCRIPTION	End of range of a joint of the secondary is overrun.
PATTERN	avoid
CONCERN	safety
PRIORITY	High
SOURCE	Breakage of the sensor of end of range or error of control software
REACTION	To stop the electrical supply

HAZARD	
NAME	HZ.4
DESCRIPTION	Joint of Secondary touches the hull of ship.
PATTERN	avoid

CONCERN	safety
PRIORITY	High
SOURCE	Error of control software to compute the path. Breakage of the sensor of end of range. Breakage of the joint. Breakage of the electricity supply.
REACTION	To stop the joint and the electrical supply

HAZARD	
NAME	HZ.5
DESCRIPTION	Secondary does not stop.
PATTERN	avoid
CONCERN	safety
PRIORITY	High
SOURCE	Error of control software of the joint Breakage of the wiring.
REACTION	To stop the electrical supply. Emergency stop.

Applying the step (v), the risks of each pair *Safe* (T_i) — *Avoid* (Hz_j) were established. Figure 6-14 depicts the risks associated to each pair, specifically for the requirements associated to the control of the Secondary Positioning System.

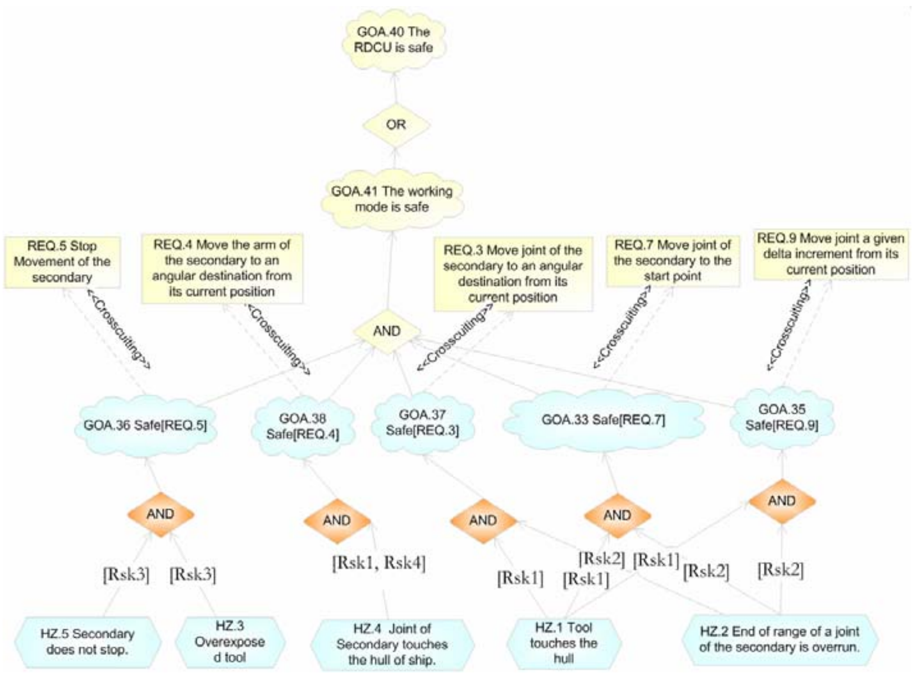

GOA.40 The RDCU is safe
OR
GOA.41 The working mode is safe
REQ.5 Stop Movement of the secondary
REQ.4 Move the arm of the secondary to an angular destination from its current position
REQ.3 Move joint of the secondary to an angular destination from its current position
REQ.7 Move joint of the secondary to the start point
REQ.9 Move joint a given delta increment from its current position
<<Crosscutting>>
AND
GOA.36 Safe[REQ.5]
GOA.38 Safe[REQ.4]
GOA.37 Safe[REQ.3]
GOA.33 Safe[REQ.7]
GOA.35 Safe[REQ.9]
[Rsk3]
[Rsk3]
[Rsk1, Rsk4]
[Rsk1]
[Rsk1]
[Rsk2]
[Rsk1]
[Rsk2]
[Rsk2]
HZ.5 Secondary does not stop.
HZ.3 Overexposed tool
HZ.4 Joint of Secondary touches the hull of ship.
HZ.1 Tool touches the hull
HZ.2 End of range of a joint of the secondary is overrun.

Figure 6-14 Specifying risks for each pair Safe(T_i)-Avoid(Hz_j)

Table 6-2. Summary of Risks for the Secondary Positioning System

Risk	Description
Rsk1	Damage of the tool or any mechanical component of the Secondary Positioning subsystem.
Rsk 2	Mechanical damage to the arm joint.
Rsk 3	Damage to the hull surface.
Rsk 4	Mechanical damages to the Secondary Positioning subsystem.

Once the risks are determined, the step (vi) was applied to determine the RRCs to be used in each specific case. As was indicated in the previous section, it is necessary to determine the *Severity*, *Exposure* and *Avoidance* on each pair *Safe* (T_i) — *Avoid* (Hz_j) to establish the appropriate RRC. Several RRCs can be associated to a specific Hazard, for this reason, the severest is the one to be applied. Table 6-3 shows the evaluation of the RRCs after the attributes severity, exposure, and avoidance were established. As can be observed, most of the identified RRCs do not determine any important modification on the system. For instance, the RRC R2A established for the HZ1 determines that mechanisms to cease that hazard if it appears have to be included. However, if the R1 RRC had been established then the hazard would have to be eliminated. Each time this modification is performed a re-evaluation of the graph must proceed in order to determine if the residual risks are tolerable for the system. However, for the RDCU it was determined that the residual risks are tolerable.

Table 6-3 Determining the RRC for each established Hazard

Hazard	Risk	Severity	Exposure	Avoidance	RRC
HZ1	Rsk1	S2	E1	A2	R2B
HZ2	Rsk2	S1	E1	A1	R4
HZ3	Rsk3	S1	E2	A1	R3A
HZ4	Rsk1 Rsk4	S2	E1	A2	R2B
HZ5	Rsk3	S1	E1	A1	R4

Related to the analysis of residual risks of the system, the customizable analysis process described in section 5.4.2 can be exploited. A set of rules were described to determine if any hazard had a RRC greater than R2A, propagating its values to the safety root goal. Table 6-4 shows the rules to determine the current level of tolerance. They determine that if any RRC is R1, the safety root node is denied.

Table 6-4 Describing safety rules to determine actual tolerance level

Relationship	Condition	Valuation
Sat $((g_2, g_3) \xrightarrow{and} g_1)$	Goal$_D$.Concern="Safety"	Goal$_D$.Sat = min(Goal$_i$.Sat)
Den $((g_2, g_3) \xrightarrow{and} g_1)$	Goal$_D$.Concern="Safety"	Goal$_D$.Sat = max(Goal$_i$.Den)
Sat $(g_2, g_3) \xrightarrow{or} g_1)$	Goal$_D$.Concern="Safety"	Goal$_D$. Sat = max(Goal$_i$.Sat)
Den $(g_2, g_3) \xrightarrow{or} g_1)$	Goal$_D$.Concern="Safety"	Goal$_D$. Den = min(Goal$_i$. Den)
Sat$((g_2, g_3) \xrightarrow{andSafety} g_1)$	(Goal$_D$.Concern="Safety") && (min(Hazard$_i$.RRC)< "R1")	Goal$_D$.Sat = 'F'
	(Goal$_D$.Concern="Safety") && (min(Hazard$_i$.RRC)== "R1")	Goal$_D$.Sat = 'N'
Den$((g_2, g_3) \xrightarrow{andSafety} g_1)$	(Goal$_D$.Concern="Safety") && (max (Hazard$_i$.RRC)== "R1")	Goal$_D$.Sat = 'F'
	(Goal$_D$.Concern="Safety") && (max (Hazard$_i$.RRC)< "R1")	Goal$_D$.Sat = 'N'
Sat$((g_2, g_3) \xrightarrow{orSafety} g_1)$	(Goal$_D$.Concern="Safety") && (max (Hazard$_i$.RRC)< "R1")	Goal$_D$.Sat = 'F'
	(Goal$_D$.Concern="Safety") && (max (Hazard$_i$.RRC)== "R1")	Goal$_D$.Sat = 'N'
Den$((g_2, g_3) \xrightarrow{orSafety} g_1)$	(Goal$_D$.Concern="Safety") && (min(Hazard$_i$.RRC)== "R1")	Goal$_D$.Den = 'F'
	(Goal$_D$.Concern="Safety") && (min(Hazard$_i$.RRC)< "R1")	Goal$_D$.Den = 'N'

By employing the described rules, the propagation of satistiability was performed for the safety goals. As can be observed, all the safety goals were fully satisfied because any hazard has R1 as RRC.

Table 6-5 Propagation results according to the described rules

Goal	Sat	Den
GOA.40	F	N
GOA.41	F	N

GOA.36	F	N
GOA.38	F	N
GOA.37	F	N
GOA.33	F	N
GOA.35	F	N

6.2.7 Operationalizing the model

Once the goals and requirements have been established, the Goal Model can be operationalized, that is, the description of how the requirements can be made operational by the system-to-be is performed. With this aim, for every requirement described in the previous sections, their possible operationalizations are specified. In order to facilitate the guidance of the process they are described in the following according to the different identified concerns of the RDCU.

As was described in Figure 5-8, the first step to perform, previously to describe the operationalizations, is to select the Architectural Style to apply. In this sense, the ACROSET (Ortiz et al., 2005) was selected because it has been defined for the description of robotic families, as the RDCU is. This Architectural Style proposes an initial assignment of responsibilities in layers:

– The first layer is devoted to actuators and sensors, the elements in charge of controlling joints of the systems, tools, and any robotic devices existing in the system.

– The second layer, called *Simple Unit Controller (SUC)* consists of actuators and sensors for every joint, tool and device element.

– The third layer, called *Mechanism Unit Controller* (*MUC*), is employed to command a set of SUCs. The set is described to accomplish a specific aim, for instance, a coordinated movement of the joints.

– The fourth layer, called *Robotic Unit Controller* (RUC), is in charge of the coordination of the whole system by means of its access to the MUCs of the system.

Therefore, this Architectural Style guides the analyst in the assignment of responsibilities that must be performed during the operationalization activity. In addition, some patterns have been defined along with this Architectural Style that facilitate the proper definition of operationalizations that contribute towards non-functional requirements. It is presented in the following how these patterns have been applied.

In addition, it must be mentioned that the following sections are going to focus their attention on the Teachmover (presented in the section 4.2.2). This is one of the systems that must be controlled by the RDCU. It has been selected because although it presents the main requirements of any system of these characteristics, it is simple enough to facilitate the comprehension of the explanation. Besides, the Teachmover provides a COTS component that can be used to facilitate the interaction with it.

Suitability

A wide set of requirements have been described regarding this concern. An example is showed in Figure 6-4 were two requirements, REQ.1 and REQ.2 to open and close the tool, respectively, were described. Both requirements can be operationalized as:

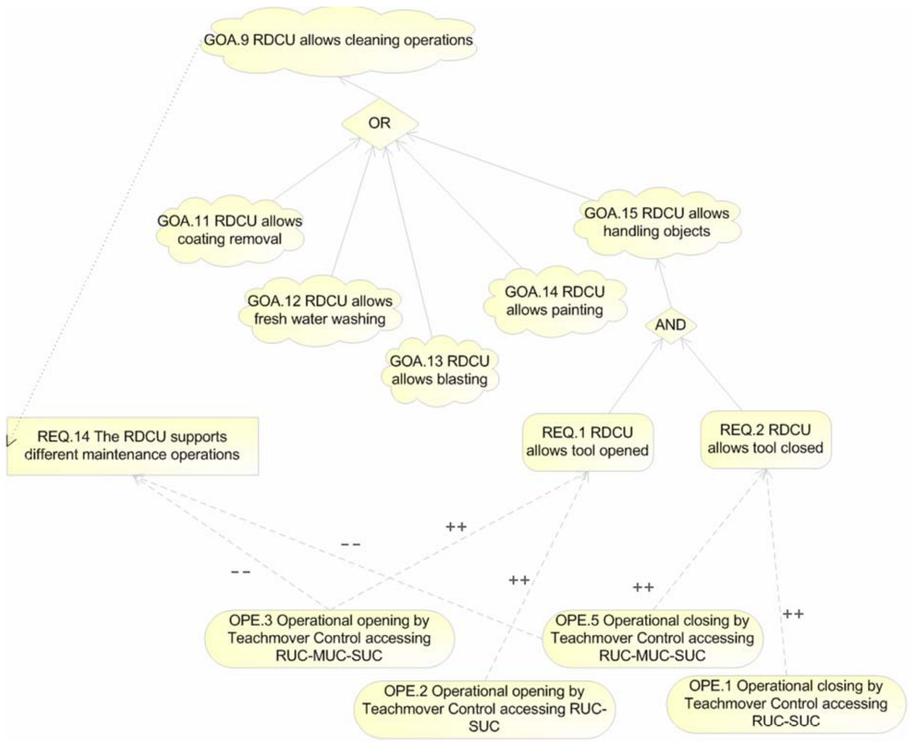

GOA.9 RDCU allows cleaning operations
OR
GOA.11 RDCU allows coating removal
GOA.15 RDCU allows handling objects
GOA.12 RDCU allows fresh water washing
GOA.14 RDCU allows painting
AND
GOA.13 RDCU allows blasting
REQ.14 The RDCU supports different maintenance operations
REQ.1 RDCU allows tool opened
REQ.2 RDCU allows tool closed
++
--
--
++
++
++
OPE.3 Operational opening by Teachmover Control accessing RUC-MUC-SUC
OPE.5 Operational closing by Teachmover Control accessing RUC-MUC-SUC
OPE.2 Operational opening by Teachmover Control accessing RUC-SUC
OPE.1 Operational closing by Teachmover Control accessing RUC-SUC

Figure 6-15 Operationalizing the requirements of the tool

It can be observed that two alternatives have been described for each requirement. One of them considers that the RDCU has direct access to the SUC, and the other one establishes that the control must be performed by using the MUC. However, the second alternative, as is also described in the figure, has a negative contribution towards the other described goals. This is because if the MUC controls the tool, the number of operations that can be performed are limited to only the active tool. This limitation implies that the requirement REQ.14 has a negative contribution from OPE.3 and OPE.5.

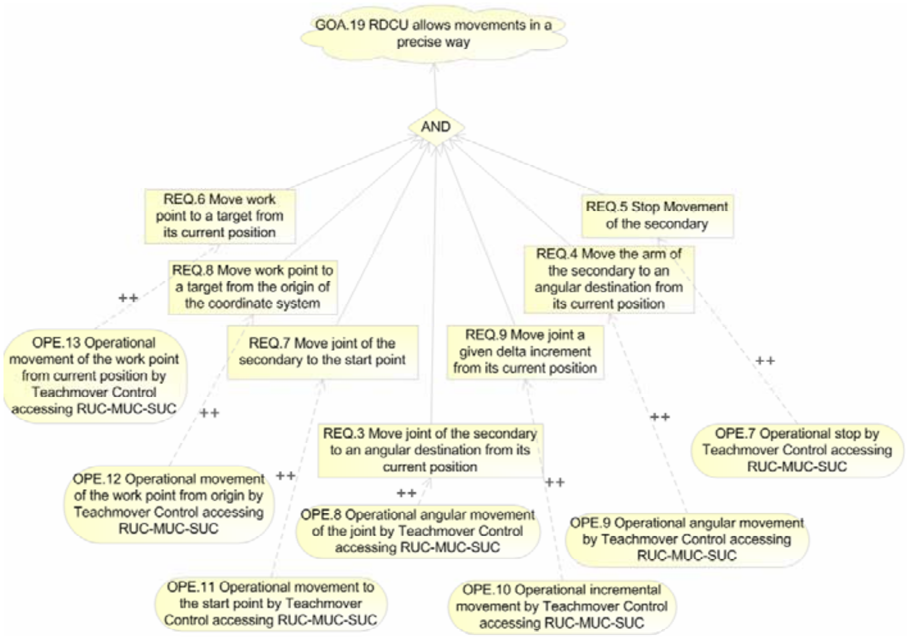

GOA.19 RDCU allows movements in a precise way
AND
REQ.6 Move work point to a target from its current position
REQ.5 Stop Movement of the secondary
REQ.8 Move work point to a target from the origin of the coordinate system
REQ.4 Move the arm of the secondary to an angular destination from its current position
++
OPE.13 Operational movement of the work point from current position by Teachmover Control accessing RUC-MUC-SUC
REQ.7 Move joint of the secondary to the start point
REQ.9 Move joint a given delta increment from its current position
++
++
++
REQ.3 Move joint of the secondary to an angular destination from its current position
OPE.7 Operational stop by Teachmover Control accessing RUC-MUC-SUC
OPE.12 Operational movement of the work point from origin by Teachmover Control accessing RUC-MUC-SUC
++
++
++
OPE.8 Operational angular movement of the joint by Teachmover Control accessing RUC-MUC-SUC
OPE.9 Operational angular movement by Teachmover Control accessing RUC-MUC-SUC
OPE.11 Operational movement to the start point by Teachmover Control accessing RUC-MUC-SUC
OPE.10 Operational incremental movement by Teachmover Control accessing RUC-MUC-SUC

Figure 6-16 Operationalizing the requirements of movement of the secondary system

As can be observed in the Figure 6-16 each requirement was operationalized by using the same pattern, that is, using the control provided by the RUC-MUC-SUC, but each one of them is going to be described to perform a different operation.

Reliability

As was described in section 6.2.2, several goals were specified related to *reliability*. However, only goals and requirements related to *recoverability* were specified. Specifically, the REQ.11, i.e., the ability to operate separately the devices, was established as a refinement of recoverability. Figure 6-17 shows how the REQ.11 has been operationalized. As can be observed, some operationalizations, which were already specified for requirements of suitability, are positively contributing towards this requirement. They provide an operational description in the system-to-be to meet that requirement too.

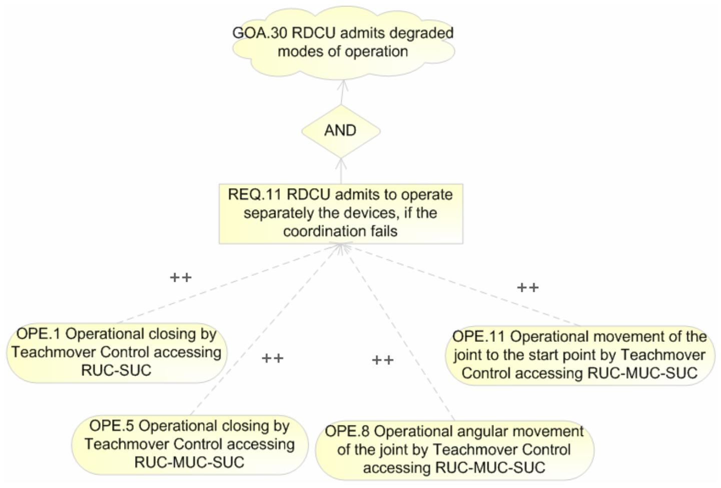

GOA.30 RDCU admits degraded modes of operation
AND
REQ.11 RDCU admits to operate separately the devices, if the coordination fails
++
++
++
++
OPE.1 Operational closing by Teachmover Control accessing RUC-SUC
OPE.11 Operational movement of the joint to the start point by Teachmover Control accessing RUC-MUC-SUC
OPE.5 Operational closing by Teachmover Control accessing RUC-MUC-SUC
OPE.8 Operational angular movement of the joint by Teachmover Control accessing RUC-MUC-SUC

Figure 6-17 Operationalizing the requirement of recoverability

Safety

In the context of the ACROSET, several architectural patterns have been defined related to the safety concern. They have been described in the domain of the tele-operated domain so that they can be very helpful for the operationalization step. In the following, two patterns have been used as alternatives to operationalize the safety requirements that were described, that is, the hazards to avoid in the system-to-be. They are described as:

a) *Introduction of a Safety Aspect in the SUC connector*. This aspect is going to factorize the safety mechanisms to avoid that hazards may arise. It is defined by means of a set of services that will check if every movement has safety parameters, i.e., they are not going to force the system for a position located out of its reach.

b) *Introduction of a Redundant Node Control in the SUC*. This node is going to check if every step performed by the system is executed according to the ones initially calculated, in such a way that if any difference is detected then the movement is automatically stopped. It must be considered that the introduction of alternatives means a decrement of the efficiency in terms of time response and resource utilization. Every step of every joint must be checked after it is performed thus the system reduces its speed to perform a maintenance task. In addition, a software connector is always heavier, in terms of memory, than an aspect.

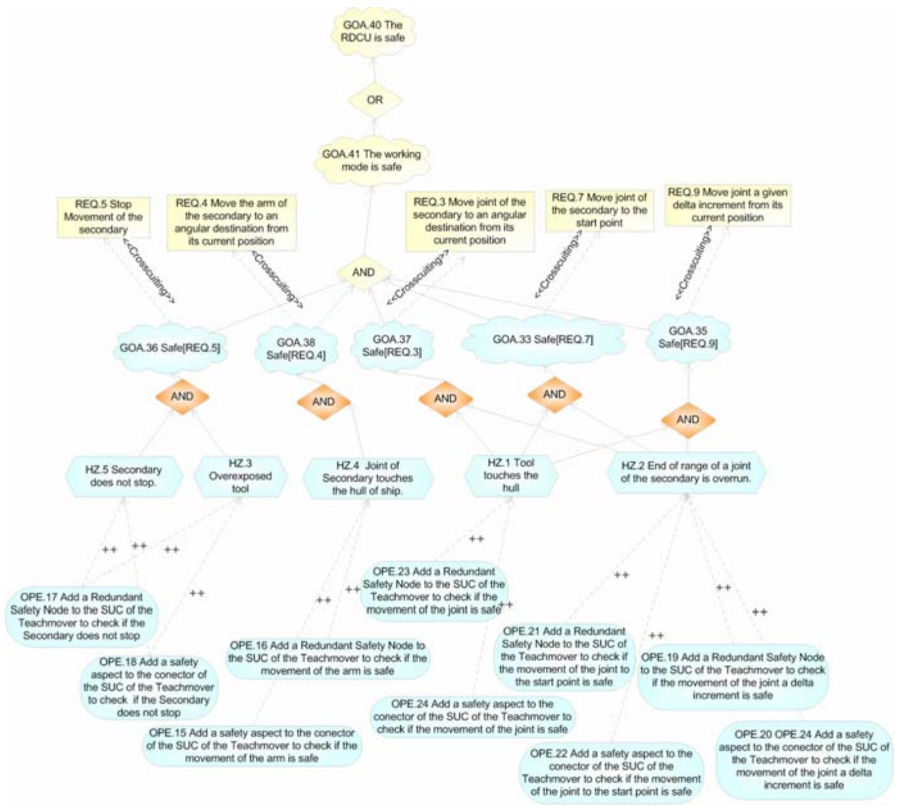

GOA.40 The RDCU is safe
OR
GOA.41 The working mode is safe
REQ.5 Stop Movement of the secondary
REQ.4 Move the arm of the secondary to an angular destination from its current position
REQ.3 Move joint of the secondary to an angular destination from its current position
REQ.7 Move joint of the secondary to the start point
REQ.9 Move joint a given delta increment from its current position
<<Crosscuting>>
AND
GOA.36 Safe[REQ.5]
GOA.38 Safe[REQ.4]
GOA.37 Safe[REQ.3]
GOA.33 Safe[REQ.7]
GOA.35 Safe[REQ.9]
HZ.5 Secondary does not stop.
HZ.3 Overexposed tool
HZ.4 Joint of Secondary touches the hull of ship.
HZ.1 Tool touches the hull
HZ.2 End of range of a joint of the secondary is overrun.
++
OPE.23 Add a Redundant Safety Node to the SUC of the Teachmover to check if the movement of the joint is safe
OPE.17 Add a Redundant Safety Node to the SUC of the Teachmover to check if the Secondary does not stop
OPE.16 Add a Redundant Safety Node to the SUC of the Teachmover to check if the movement of the arm is safe
OPE.21 Add a Redundant Safety Node to the SUC of the Teachmover to check if the movement of the joint to the start point is safe
OPE.19 Add a Redundant Safety Node to the SUC of the Teachmover to check if the movement of the joint a delta increment is safe
OPE.18 Add a safety aspect to the conector of the SUC of the Teachmover to check if the Secondary does not stop
OPE.24 Add a safety aspect to the conector of the SUC of the Teachmover to check if the movement of the joint is safe
OPE.15 Add a safety aspect to the conector of the SUC of the Teachmover to check if the movement of the arm is safe
OPE.22 Add a safety aspect to the conector of the SUC of the Teachmover to check if the movement of the joint to the start point is safe
OPE.20 OPE.24 Add a safety aspect to the conector of the SUC of the Teachmover to check if the movement of the joint a delta increment is safe

Figure 6-18 Operationalizing safety requirements

Considering both patterns, the specification of the Figure 6-18 was performed to operationalize the safety requirements. It can be observed that the two patterns described above were used to describe each operationalization, considering the requirements that have to be safeguarded and the operationalization to modify to deal with the safety concern. It is worthy of note that there is no contribution from these operationalizations towards the suitability requirements. It must be taken into account that they are not describing an operationalization of the suitability concern but they are defined to meet the safety requirements.

6.3 ANALYZING THE ATRIUM GOAL MODEL

Once the Goal Model has been operationalized, it is just the moment to perform the analysis. According to the process described in the Figure 5-10, the first step to perform is to describe the set of rules that must be used for the propagation. The set of used rules are described in Table 6-6. It must be noted that the Giorgini's rules have been extended to take into account the set of artifacts and relationships described in this proposal.

Table 6-6 Rules for reasoning Analysis

Relationship	Condition	Valuation
Sat $((g_2,g_3)\xrightarrow{and}g_1)$		$Goal_D$.Sat = max((min($Goal_i$.Sat), $Goal_D$.Sat)
Den $((g_2,g_3)\xrightarrow{and}g_1)$		$Goal_D$. Den = max(max($Goal_i$.Den), $Goal_D$. Den)
Sat $((g_2,g_3)\xrightarrow{andSafety}g_1)$		$Goal_D$.Sat = max((min($Goal_i$.Sat), $Goal_D$.Sat)
Den$((g_2,g_3)\xrightarrow{andSafety}g_1)$		$Goal_D$. Den = max(max($Goal_i$.Den), $Goal_D$. Den)
Sat $((g_2,g_3)\xrightarrow{or}g_1)$		$Goal_D$.Sat = max((max($Goal_i$.Sat), $Goal_D$.Sat)
Den $((g_2,g_3)\xrightarrow{or}g_1)$		$Goal_D$.Den = max((min($Goal_i$.Sat), $Goal_D$.Sat)
Sat$(g_2\xrightarrow{++}g_1)$	(Contribution.contributes='++')	$Goal_D$.Sat = max($Goal_S$.Sat, $Goal_D$.Sat)
Den$(g_2\xrightarrow{++}g_1)$	(Contribution.contributes='++')	$Goal_D$.Den = max($Goal_S$.Den, $Goal_D$.Den)
Sat$(g_2\xrightarrow{+}g_1)$	(Contribution.contributes='+')	$Goal_D$.Sat = max(min($Goal_S$.Sat, 'P')), $Goal_D$.Sat)
Den$(g_2\xrightarrow{+}g_1)$	(Contribution.contributes ='+')	$Goal_D$.Den = max(min($Goal_S$.Den, 'P')), $Goal_D$. Den)

Sat($g_2 \xrightarrow{-} g_1$)	(Contribution.contributes ='-')	Goal_D.Sat = max(min(Goal_S.Den, 'P')), Goal_D.Sat)
Den($g_2 \xrightarrow{-} g_1$)	(Contribution.contributes ='-')	Goal_D.Den = max(min(Goal_S.Sat, 'P')), Goal_D.Den)
Sat($g_2 \xrightarrow{--} g_1$)	(Contribution.contributes ='--')	Goal_D.Sat = max(Goal_S.Den, Goal_D.Sat)
Den($g_2 \xrightarrow{--} g_1$)	(Contribution.contributes ='--')	Goal_D.Den = max(Goal_S.Sat, Goal_D.Den)
Sat($g_2 \xrightarrow{required} g_1$)		Goal_D.Sat = min(Goal_S.Sat, Goal_D.Sat)
Den($g_2 \xrightarrow{required} g_1$)		Goal_D.Den = max(Goal_S.Den, Goal_D.Den)

Table 6-7 shows an example of the propagation applying the previous rules. As can be observed, most of the goals are not denied nor satisfied. Most of them do not have associated operationalizations, and they have not been refined to avoid a complex example that could be difficult to follow. A noticeable example is the GOA.1, which is not satisfied neither denied. The reason is that it has been refined by means of an AND relationship into the different working modes. Only the working mode (GOA.2) has been included in the propagation so that the positive propagation ends in this goal.

Table 6-7 Propagation results: values of satisfiability (Sat) and deniability (Sat) before (t_0) and after (t_1) the propagation

Artifact	**Sat**(t_0)	**Den**(t_0)	**Sat**(t_1)	**Den**(t_1)
GOA.1 RDCU be suitable for the user needs	N	N	N	N
GOA.2 RDCU allows working operation	N	N	F	N
GOA.3 RDCU allows safe stop	N	N	N	N
GOA.4 RDCU allows maintenance operation	N	N	N	N
GOA.5 RDCU allows learning	N	N	N	N
GOA.6 RDCU allows calibration	N	N	N	N
GOA.7 RDCU allows diagnosis	N	N	N	N
GOA.8 RDCU allows configuration	N	N	N	N
GOA.16 RDCU allows coating removal by blasting	N	N	N	N

GOA.17 RDCU allow coating removal by pressured water	N	N	N	N
GOA.10 RDCU coordinate positing systems	N	N	F	N
GOA.9 RDCU allows cleaning operations	N	N	F	N
GOA.11 RDCU allows coating removal	N	N	N	N
GOA.12 RDCU allows fresh water washing	N	N	N	N
GOA.13 RDCU allows blasting	N	N	N	N
GOA.14 RDCU allows painting	N	N	N	N
GOA.15 RDCU allows handling objects	N	N	F	N
OPE.1 Operational closing by Teachmover Control accessing RUC-SUC	F	N	N	N
OPE.10 Operational incremental movement by Teachmover Control accessing RUC-MUC-SUC	N	N	N	N
OPE.2 Operational opening by Teachmover Control accessing RUC-SUC	F	N	N	N
OPE.3 Operational opening by Teachmover Control accessing RUC-MUC-SUC	N	N	N	N
OPE.5 Operational closing by Teachmover Control accessing RUC-MUC-SUC	N	N	N	N
REQ.1 RDCU allows tool opened	N	N	F	N
REQ.2 RDCU allows tool closed	N	N	F	N
GOA.18 RDCU allows movement across wide areas	N	N	N	N
GOA.19 RDCU allows movements in a precise way	N	N	F	N
GOA.20 RDCU allows movement across horizontal	N	N	N	N
GOA.21 RDCU allows movement across vertical surface	N	N	N	N
OPE.11 Operational movement of the joint to the start point by Teachmover Control accessing RUC-MUC-SUC	F	N	N	N
OPE.12 Operational movement of the work point from origin by Teachmover Control accessing RUC-MUC-SUC	F	N	N	N
OPE.13 Operational movement of the work point from current position by Teachmover Control accessing RUC-MUC-SUC	F	N	N	N
OPE.7 Operational stop by Teachmover Control accessing RUC-MUC-SUC	F	N	N	N
OPE.8 Operational angular movement of the joint by Teachmover Control accessing RUC-MUC-SUC	F	N	N	N

OPE.9 Operational angular movement by Teachmover Control accessing RUC-MUC-SUC	F	N	N	N
REQ.3 Move joint of the secondary to an angular destination from its current position	N	N	N	N
REQ.4 Move the arm of the secondary to an angular destination from its current position	N	N	N	N
REQ.5 Stop Movement of the secondary	N	N	N	N
REQ.6 Move work point to a target from its current position	N	N	N	N
REQ.7 Move joint of the secondary to the start point	N	N	N	N
REQ.8 Move work point to a target from the origin of the coordinate system	N	N	N	N
REQ.9 Move joint a given delta increment from its current position	N	N	N	N
GOA.24 RDCU is reliable	N	N	F	N
GOA.25 RDCU is mature	N	N	N	N
GOA.26 RDCU is fault tolerance	N	N	N	N
GOA.27 RDCU is recoverable	N	N	F	N
GOA.28 RDCU is available during hundred of hours	N	N	N	N
GOA.29 RDCU repair time is as short as possible	N	N	N	N
GOA.30 RDCU admits degraded modes of operation	N	N	F	N
GOA.31 RDCU admits to operate s	N	N	N	N
REQ.11 RDCU admits to operate separately the devices, if the coordination fails	N	N	F	N
GOA.32 The RDCU is portable	N	N	N	N
GOA.33 The RDCU is adaptable	N	N	N	N
REQ.14 The RDCU supports different maintenance operations	N	N	N	N
REQ.16 The RDCU supports different primary possitioning systems	N	N	N	N
REQ.17 The RDCU supports different secondary possitioning systems	N	N	N	N
GOA.33 Safe[REQ.7]	N	N	F	N
GOA.36 Safe[REQ.5]	N	N	F	N
GOA.37 Safe[REQ.3]	N	N	F	N
GOA.38 Safe[REQ.4]	N	N	F	N

GOA.35 Safe[REQ.9]	N	N	F	N
GOA.40 The RDCU is safe	N	N	F	N
GOA.41 The working mode is safe	N	N	F	N
HZ.1 Tool touches the hull	N	N	F	N
HZ.2 End of range of a joint of the secondary is overrun.	N	N	F	N
HZ.3 Overexposed tool	N	N	F	N
HZ.4 Joint of Secondary touches the hull of ship.	N	N	F	N
HZ.5 Secondary does not stop.	N	N	F	N
OPE.15 Add a safety aspect to the conector of the SUC of the Teachmover to check if the movement of the arm is safe	F	N	F	N
OPE.16 Add a Redundant Safety Node to the SUC of the Teachmover to check if the movement of the arm is safe	N	N	N	N
OPE.18 Add a safety aspect to the conector of the SUC of the Teachmover to check if the Secondary does not stop	F	N	F	N
OPE.19 Add a Redundant Safety Node to the SUC of the Teachmover to check if the movement of the joint a delta increment is safe	N	N	N	N
OPE.20 OPE.24 Add a safety aspect to the conector of the SUC of the Teachmover to check if the movement of the joint a delta increment is safe	F	N	F	N
OPE.21 Add a Redundant Safety Node to the SUC of the Teachmover to check if the movement of the joint to the start point is safe	N	N	N	N
OPE.22 Add a safety aspect to the conector of the SUC of the Teachmover to check if the movement of the joint to the start point is safe	F	N	F	N
OPE.23 Add a Redundant Safety Node to the SUC of the Teachmover to check if the movement of the joint is safe	N	N	N	N
OPE.24 Add a safety aspect to the conector of the SUC of the Teachmover to check if the movement of the joint is safe	F	N	F	N

6.4 CONCLUSIONS

A case study has been presented where every described concept, step and process has been put into practice. It was appreciated that the ISO9126 has been the framework of concerns to guide the elicitation process.

Specially meaningful has been how the safety concern, so relevant for the EFTCoR, has been managed by means of the extension of the process of ATRIUM and the extension of its Metamodel to deal with issues such as hazards, risks, etc. The consideration of very relevant standards and practical approaches (ANSI, Douglass and Leveson) for the domain of safety requirements has been the source to establish some methodological guidelines for safety requirements specification of tele-operated robotic systems.

As was stated above, the specification of safety requirements is a major challenge for the tele-operated systems due to both the combination of hardware/software components and the presence of potential injury to people or equipment. The use of the ATRIUM Goal Model allows the analyst to use a unified proposal for requirement specification in this domain, which provides the integration and derivation of safety requirements with the remaining requirements of the specification. In addition, adopting a goal-oriented approach is convenient due to the obvious benefits for analysis and evaluation of alternatives. Moreover, although it has not been presented in the example, the formal base of the goal-oriented approach allows the verification of several properties when goals and requirements are formalized.

Most of the publication presented in the previous chapter included a case study section where the work presented in this chapter was published. In addition, the work related to the exploitation of ATRIUM for defining safety requirements has been presented in the following publications:

– E. Navarro, P. Sánchez, P. Letelier, J. A. Pastor, I. Ramos, “Sistematizando la Especificación de Requisitos Safety en Aplicaciones Teleoperadas”, Proceedings of X Jornadas de Ingeniería del Software y Bases de Datos (JISBD’2005) Granada, September 14-16, 2005, Toval, A. Hernández, J. (eds). Thomson Paraninfo, Spain, pp. 35-42, ISBN:84-9732-434-X.

– E. Navarro, P. Sánchez, P. Letelier, J.A. Pastor, I. Ramos, “A Goal-Oriented Approach for Safety Requirements Specification”, Proceedings 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS’06), Postdam, Germany, March 27th-30th, 2006, pp. 319-326.

"The worst is not to make a mistake but to seek a justification instead of using it as a providential warning of our flippancy and ignorance."— Santiago Ramón y Cajal.

CHAPTER 7

Scenarios to run Aspect-Oriented Software Architectures

7.1 INTRODUCTION

Scenario-based proposals, such as that presented by (Maiden, 1998) and (Leite et al., 2000), have been used to describe and reason about large-grained behaviour patterns in systems, as well as the coupling of these patterns. For this reason, the specification of scenarios is a proper first step towards the description of the system-to-be. They provide a way to reason about partial views of the architecture, facilitating an improved comprehension of the system-to-be and a way to analyse alternatives architectural descriptions.

Most of Scenario-based proposals needs for an artifact establishing the requirements of the system-to-be to trigger their definition. In this sense, a massive amount of work on linking goals and scenarios together can be found in the literature. The obvious reason is that scenarios and goals have complementary characteristics because, as (Lamsweerde, 2001a) states:

> *"The former are concrete, narrative, procedural, and leave intended properties implicit; the latter are abstract, declarative, and make intended properties explicit."*
>
> *"Goals can be validated by identifying or generating scenarios that are covered by them. One may even think of enacting such scenarios to produce animations."*

In this sense, several proposals can be found in the literature. For instance, GRL (Liu & Yu, 2004) have been defined by using a Goal Model approach along with its traceability, from goals towards `Use Case Maps` (UCM). However, these UCM are close far of the Architectural Scenarios that are used in our work because their abstraction level is much higher. In addition, they made an early splitting of the work between the system-to-be and the environment because the responsibility is assigned at the goals level.

Both GBRAM (Antón, 1996) and CREWS l'Escritorie (Rolland et al., 1999) are also quite similar to our proposal because they use scenario as operationalization of the specified goals. However, as in these proposals they are not architectural scenarios for the higher abstraction level.

(Araujo et al., 2004) have introduced the crosscutting management during the definition of scenarios. However, their motivation has been different because they try to describe scenarios that crosscut other scenarios as interaction patterns using the (Kim et al., 2004)'s proposal. This means that their motivation is quite distant of that proposed in this work. In addition, they do not provide any notation for aspect description nor traceability through the lifecycle of the system-to-be.

In addition, we would like to highlight that none of the mentioned proposals, as far as we know, face an open issue: the separation of concerns and its proper traceability from the very beginning of the software lifecycle. This has been the main motivation for the introduction of ATRIUM, introducing this separation at the requirements, scenarios, architecture and code level.

In our approach, both Goals Model, presented in the previous chapter, and Scenarios Model elaboration are two intertwined processes, in order to overcome some of their deficiencies and limitations when used in isolation. The ATRIUM Goal Model describes the goals and requirements to be met by the system-to-be. Therefore, it is the main input used for the description of the ATRIUM Scenario Model. It facilitates the traceability among them can be rightly established and maintained. In this sense, operationalizations are going to play a key role. As was described in the chapter 5, they are prescriptions of the scenarios to be described during the activity *Define Scenarios*. However, these operationalizations are only textual; they do not identify the necessary collaboration nor the process. It should be taken into account that a software system is highly cooperative, that is, there are software components, connectors, humans, and, even, environmental elements which collaborate in the process of meeting the requirements. For this reason, the employment of scenarios is a meaningful advantage.

The ATRIUM Scenario Model has been defined to capture which elements, both architectural and environmental, are collaborating to satisfy the established requirements in the ATRIUM Goal Model. By specifying the Scenario Model, what their responsibilities are into this task and how their interactions are to achieve them are described. Thus, this model operationalizes the Goal Model by assigning the responsibilities for the elements in the system-to-be.

In addition, ATRIUM follows an Aspect Oriented approach. This means that ATRIUM Scenario Model must provide enough expressiveness for identifying and specifying aspects. It has been one of the key points when the notation for this Model was defined.

The exploitation of *Design Patterns* and *Architectural Styles* is very important because it means the reuse of quality solutions. They represent distilled experience that, through their assimilation, enable expert analysts to convey their knowledge and insight to inexpert ones. It seems obvious that the analyst must be provided with facilities to describe that patterns and instantiate them. For this reason, it seems obvious the introduction of patterns presents a meaningful advantage related to other approaches.

This chapter is structured as followed. Sections 7.2 and 7.3 describe the elements that are necessary for the scenario description and the specified language for this goal, respectively. Section 7.4 presents a proposal for dealing with Architectural Styles and Design patterns. The established process for scenario description is presented in section 7.5. Finally, the last section presents the main conclusions of the chapter.

7.2 ELEMENTS OF THE SCENARIOS MODEL

We should bear in mind that traceability throughout the lifecycle is one of the main concerns of ATRIUM. For this reason, once the Goal Model has been defined (as was described in chapter 5) it is expected that every identified requirement has its scenario/s associated to it, providing us with partial views of the behaviour of the system-to-be. These allow the analyst to reason about the system-to-be with a proper granularity level.

In the ATRIUM Scenario Model, a `scenario` is used to describe the system behaviour associated to one or several requirements and under certain operationalization decision. This decision establishes how the scenario is going to be described in terms of both architectural and environmental elements along with a specific choreography they stick to. Unlike classic scenarios proposals, ATRIUM Scenarios specify architectural elements interaction instead of objects along with the environmental elements which played a role in that scenario. This is due to the fact that ATRIUM aims at describing a proto-architecture which meets the established requirements.

Thus, considering that usually a requirement, which is met by the system-to-be, it is provided by the collaboration of several kinds of elements of both architecture and environment, the following interacting elements must be specified:

- `Shallow components.` They describe computational elements which are candidate to appear as PRISMA software components in the final Software Architecture description. They are called shallow because we do not provide a full description of them but an initial one. They are to be completely described if they finally appear in the final Software Architecture.
- `Shallow connectors.` They are in charge of the coordination between architectural elements and are candidates to be PRISMA software connectors. Its introduction allows a loose coupling between them with the benefits it encompasses in terms of maintainability and reuse. Like shallow components, only a rough description is performed for shallow connectors.
- `Shallow systems.` One of the needs to be satisfied, whenever a system-to-be is being described, is the ability to decompose its description in such a way that several granularity levels can be used. For this reason, shallow systems can be used when a scenario is being defined. A shallow system is a complex component that is internally described by means of components and connectors.
- `Environmental elements.` Usually systems cannot be described by themselves but in the context where they are executed. Especially, because it is becoming very frequent that they are highly hardware intensive. This implies that the ability to identify and specify such elements is almost mandatory whenever a scenario is described. An example of such a demand can be found in the EFTCoR system, our case study, where hardware has a high impact on the final description of the system
- `Humans.` Most of the current systems can be described as interactive systems because they detect and react to the behaviour of users. In addition, it is frequently the case that users can be a meaningful source of information in the process of gathering requirements. For this reason, this kind of elements has been included for the description of scenarios.
- `Commercial off-the-Shelf (COTS).` Actually is becoming almost compulsory the introduction of COTS to reduce development costs and time. Because of that, the identification of those COTS while describing a scenario is needed. This will allow us to identify the interoperability requirements i.e. the requirements that must be satisfied for

the cooperation between different products in order to facilitate their integration (Lauesen, 2006).

As can be observed a distinction has been established between humans and the remainding environment elements. It is due to the fact that interaction is usually quite different and with different results in the final description of the architecture, as will be described in chapter 8.

It is worthy of note that the identification of not only the involved elements but how their interactions are it is highly relevant when a Software Architecture is being defined. These interactions are described by the sequence of interaction `messages` between different elements in accordance with its specific choreography. Every message is in charge of describing which service is required from a specific element.

Taking into account that ATRIUM promotes the separation of `concerns` from the very beginning of the lifecycle, it is important to provide support to their detection and specification from the requirements stage and to the maintenance of their proper traceability throughout the full lifecycle. In this sense, (Rashid et al., 2002) have stated that a concern (*candidate aspect* in their proposal) can have a range of different impacts on later development stages. For this reason, they have identified a set of mappings from these concerns to functions, decisions or aspects which will impact the final system specification. Table 7-1 shows an example of some mappings they have identified.

Table 7-1 Example of mappings specification (extracted from (Rashid et al., 2002))

Candidate Aspect	Influence	Mapping
Compatibility	Specification architecture, design, evolution	Function
Response time	Architecture, design	Aspect
Legal issues	Specification architecture, design	Function
Correctness	Architecture, design	Aspect
Security	Architecture, design	Aspect
Availability	Architecture	Decision
Multi-user system	Architecture, design	Aspect

In ATRIUM this characteristic is also considered, i.e., a crosscutting concern can have a diversity of mappings on the final description of the system. It is up to the analyst to make the final decision about this matter. This means that a crosscutting concern can be realized as a service, an architectural or environmental element or an aspect, depending on the scenario that is described. However, ATRIUM provides the analyst with a set of alternatives, as described in section 7.5, to help him/her during this process.

In addition, we have to consider that ATRIUM generate by applying the *Synthesize and Transform* activity, an architectural specification. In the next chapter, it is shown how it proceeds by using PRISMA (see chapter 4) as AO-ADL. PRISMA model, manages the crosscutting at the architectural level by using *aspects* and employs them to describe the internal view of components and connectors. Figure 7-1 shows an example of the external view and internal view of the SUCConnector. We can observe that a coordination aspect "FSensor", a safety aspect "SMotion", and a distribution aspect "DRobotLocation" integrate it. As can be observed in the figure, every one of these PRISMA aspects is described as a collection of services aggregated from the point of view of a specific concern, concretely, coordination, safety and distribution in the example. Whenever this connector is used, its interacting context does not care whether it aggregates these aspects but only the services it provides.

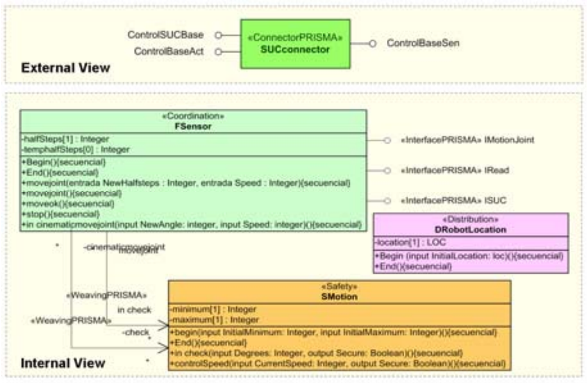

External View
ControlSUCBase
ControlBaseAct
«ConnectorPRISMA» SUCconnector
ControlBaseSen
Internal View
«Coordination» FSensor
-halfSteps[1] : Integer
-temphalfSteps[0] : Integer
+Begin(){secuencial}
+End(){secuencial}
+movejoint(entrada NewHalfsteps : Integer, entrada Speed : Integer){secuencial}
+movejoint(){secuencial}
+moveok(){secuencial}
+stop(){secuencial}
+in cinematicmovejoint(input NewAngle: integer, input Speed: integer)(){secuencial}
«InterfacePRISMA» IMotionJoint
«InterfacePRISMA» IRead
«InterfacePRISMA» ISUC
«Distribution» DRobotLocation
-location[1] : LOC
+Begin (input InitialLocation: loc)(){secuencial}
+End(){secuencial}
-cinematicmovejoint
movejoint
«WeavingPRISMA»
in check
«WeavingPRISMA»
-check
«Safety» SMotion
-minimum[1] : Integer
-maximum[1] : Integer
+begin(input InitialMinimum: Integer, input InitialMaximum: Integer)(){secuencial}
+End(){secuencial}
+in check(input Degrees: Integer, output Secure: Boolean)(){secuencial}
+controlSpeed(input CurrentSpeed: Integer, output Secure: Boolean)(){secuencial}

Figure 7-1 Internal and external view of a PRISMA connector

As described above, ATRIUM scenarios can accomplish the traceability towards architectural or environmental elements, and even services. However, a key issue to address is how the trace from those crosscuttings identified while eliciting the requirements to PRISMA aspects, or aspects in general, can be specified. Due to the fact that ATRIUM scenarios are the joint point between the requirements model and the Software Architecture, they are responsible for incorporating the capability to specify such traceability. Therefore, considering that a message is requiring a service that belongs to a specific aspect, whenever the analyst needs to specify it, he/she can do it by using an aspectual message. This kind of message is in charge of identifying the concern that the required service belongs to. This alternative improves the legibility of the scenarios because it does not include other kind of element for their description as (Cooper et al., 2005) do. This facility is highly meaningful because it is essential for the generation of the proto-architecture as described in chapter 8.

Figure 7-2 summarizes the main elements identified for the scenarios model description. It shows how interacting elements can collaborate by means of messages. In addition, it can be observed that an interaction can be described as an aspectual message as well if it is necessary.

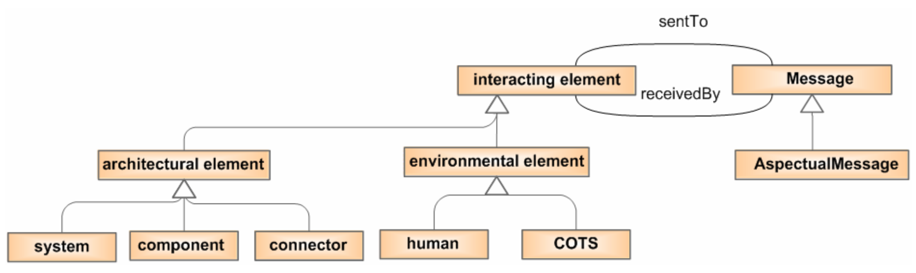


Figure 7-2 Metamodel for ATRIUM Scenarios Model

7.3 GRAPHICAL NOTATION

As graphical language for scenarios, Sequence Diagrams (SD) of UML 2.0 (UML, 2005) has been selected. SDs help to identify the coordination structure of the involved architectural and environmental elements through the process of their construction. This time-based interaction shall help us to define the choreography always necessary while defining the Software Architecture.

An UML profile based on the UML 2.0 Sequence Diagram metamodel has been defined so that it provides us with the needed expressiveness to identify and specify the elements described in section 7.2. For that specialization two steps were performed:

– Stereotyping the UML metamodel classes to describe those entities we need for our purposes.

– Defining those OCL constraints needed to ensure a proper semantics of our proposal.

Figure 7-3 depicts the outcome of first step by including those elements needed for our proposal. It shown in the figure in orange colour which elements of the UML metamodel have been extended with the needed stereotypes and tagged values in order to provide notation to that concepts included in section 7.2. In the following sections, a more detailed description of the most important elements of the metamodel is introduced. It pays special attention to those elements of UML that have been subtyped or are highly relevant for our proposal. The added elements are described along with the elements that have been subtyped. A full description of the UML metamodel is introduced in (UML, 2005). In this work, only those terms and concepts necessary for the comprehension of chapter 8 are introduced.

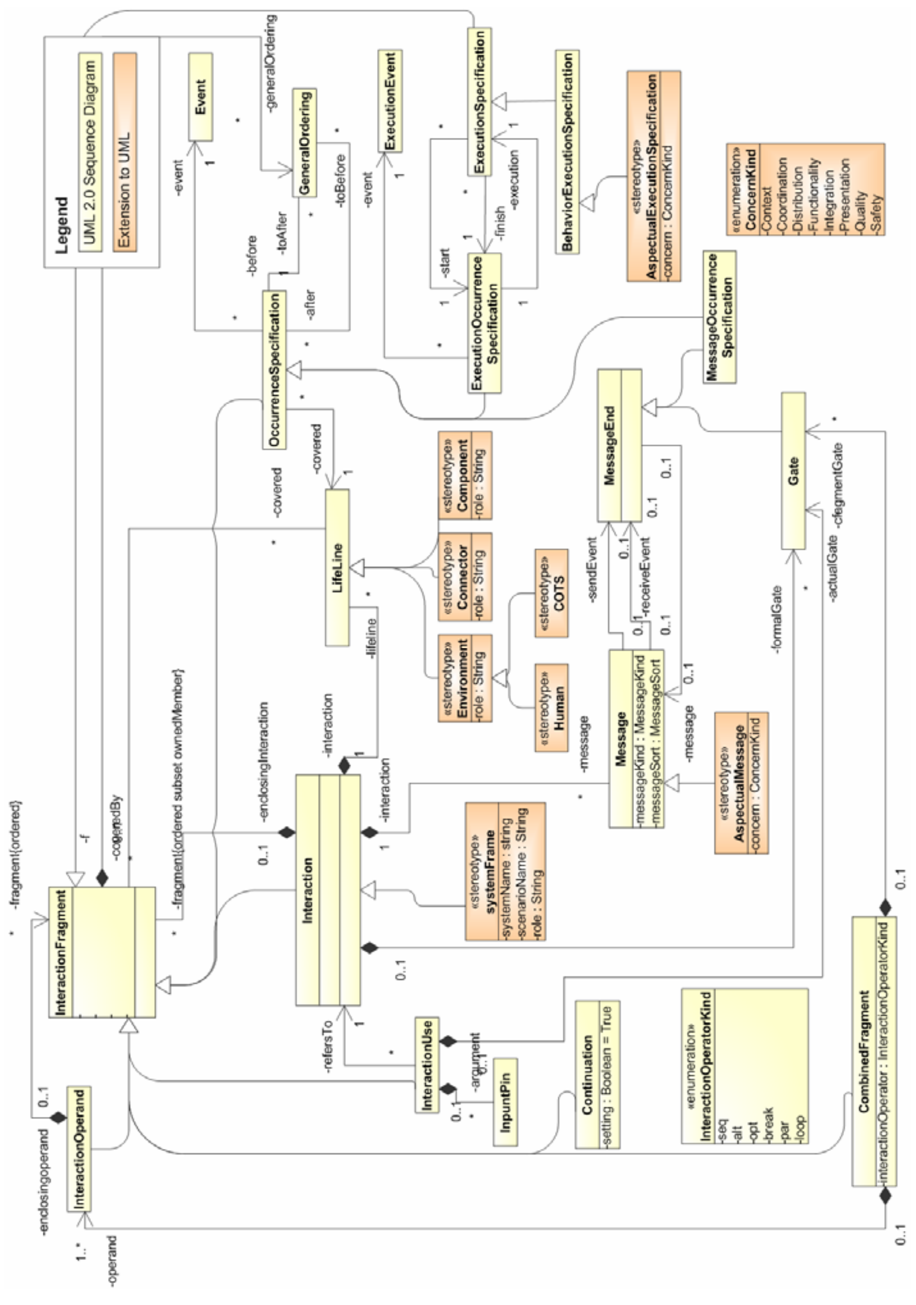


Figure 7-3 Metamodel for Scenarios Model: UML entities and Added Entities

7.3.1 Lifelines

`Lifeline` are used to represent an individual participant either architectural or environmental involved in the scenario being modelled. Each kind of element is represented by means of a different UML stereotype, i.e., that according to the metamodel described in Figure 7-3, five kind of lifelines can be used in a scenario.

- «component» is used to represent shallow components in a scenario, i.e., the main computation elements which are candidates to appear in the final description of the architecture.
- «connector» is employed to describe shallow connectors, i.e., coordination elements which are candidates to appear in the final description of the architecture.
- «environment» is incorporated to the scenario notation to describe elements which are in the environment where the system-to-be has to perform its activity.
- «human» is in charge of describing which operators are going to have interactions with the final system.
- «COTS» is used to describe which components of the self are planed to be incorporated into the system-to-be and which their collaboration is with the others elements.

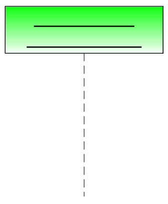
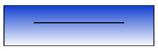
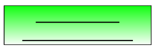
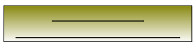
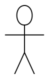
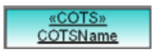


Figure 7-4 Graphical representation for Lifelines

As can be noticed, most of the interacting elements identified in section 7.2 are described by means of lifelines. Figure 7-4 depicts an example of a Lifeline which is stereotyped as a «connector». A specific colour is also used to enhance the legibility of the scenario. Table 7-2 describes the code colour used to specify each kind of element. In addition, every kind of lifeline has an attribute called "role" employed to describe if the element is going to play a specific role, that is, if it is going to have some specific assignment of responsibility. This kind of information can be provided according to the Architectural Style or

Domain-Specific Software Architectures (DSSA) that is being applied during the scenario description (see section 7.4.1).

Table 7-2 Code colour and graphical notation for lifelines

Concern	Code Colour and notation
«component»	
«connector»	
«environment»	
«human»	
«COTS»	

7.3.2 Messages

ATRIUM scenarios use messages to specify a communication between architectural and/or environmental elements. This communication can describe both a service to be provided by the requested element or the creation or destruction of an architectural element. UML 2.0, as observed in Figure 7-5, uses `Messages` to describe a communication between *Lifelines* and/or `InteractionFragments` describing this interaction by means of the `MessageEnds`. If a *Lifeline* is one of the interacting elements a `MessageOccurenceSpecification` appears as the *MessageEnd*. On the contrary, if an *InteractionFragment* is described as interacting then a `Gate` is used to specify such a connection.

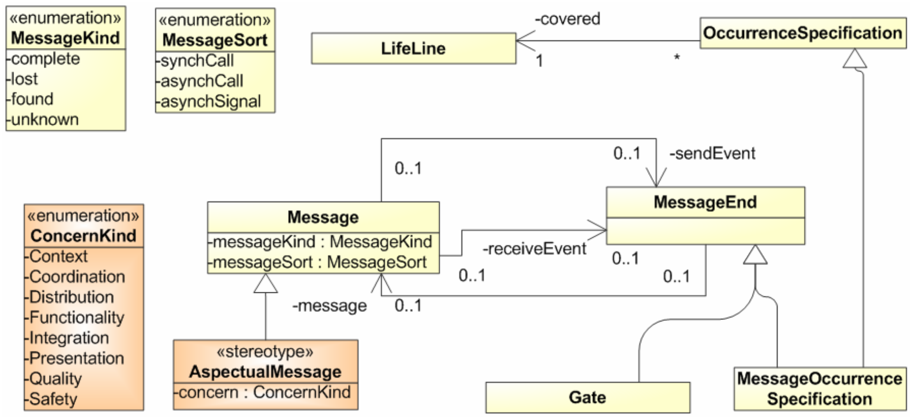


Figure 7-5 Describing Message and its extension as Aspectual Message

Messages are used in ATRIUM in a similar way to that defined in UML 2.0. However, it is necessary to consider two issues. The first is that the identification of aspects is mandatory in our proposal to provide the proper traceability. The second is that it has no sense to introduce aspects as another kind of interacting element in a scenario because they are in the internal view of the architectural elements, with a different abstraction level. To address these issues, a new kind of message has also been included in our metamotel (Figure 7-2). This message is described in UML 2.0 by means of the stereotype «AspectualMessage». It can be observed in Figure 7-5 that the tagged value *concern* has been included in its description, which is typed by *ConcernKind*. It is employed for the description of the concern the messages belongs to. Thus, whenever the analyst considers that a service should be included in the specification of a specific kind of aspect, this service will be pre-fixed with the name of the concern it refers to.

Figure 7-6 shows an example of how the service "secure check()" has been prefixed with "C{Safety}" to indicate that it will be included in a safety aspect which composes the "WristCnct" connector.

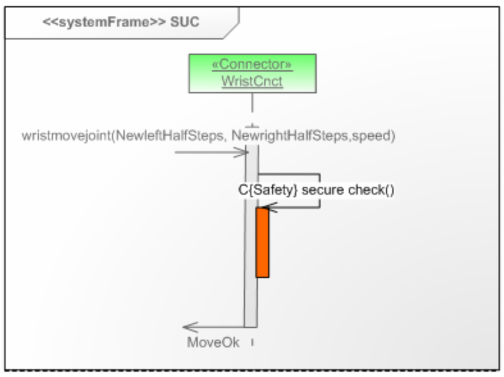


Figure 7-6 Describing a Safety Concern

We have to point out that a constraint has to be satisfied when an «AspectualMessage» is defined. It establishes that the connectable element, which receives the receiveEvent, can only be an *ArchitecturalElement*. This is due to the fact only architectural elements need to describe the concerns they are affected by.

Table 7-3 OCL Constraints for AspectualMessages

Element	**Constraint**

AspectualMessage	**context** AspectualMessage **inv**
	def allowedConnectableElements:Boolean
	Self.receiveEvent.covered.stereotype.name='Component' **or**
	Self.receiveEvent.covered.stereotype.name='Connector'

7.3.3 ExecutionSpecification

An `ExecutionSpecification` is used to illustrate the behaviour of a Lifeline that lasts between two `ExecutionOccurrenceSpecification` (see Figure 7-7): *start* which fires its existence and *finish* which finalizes it.

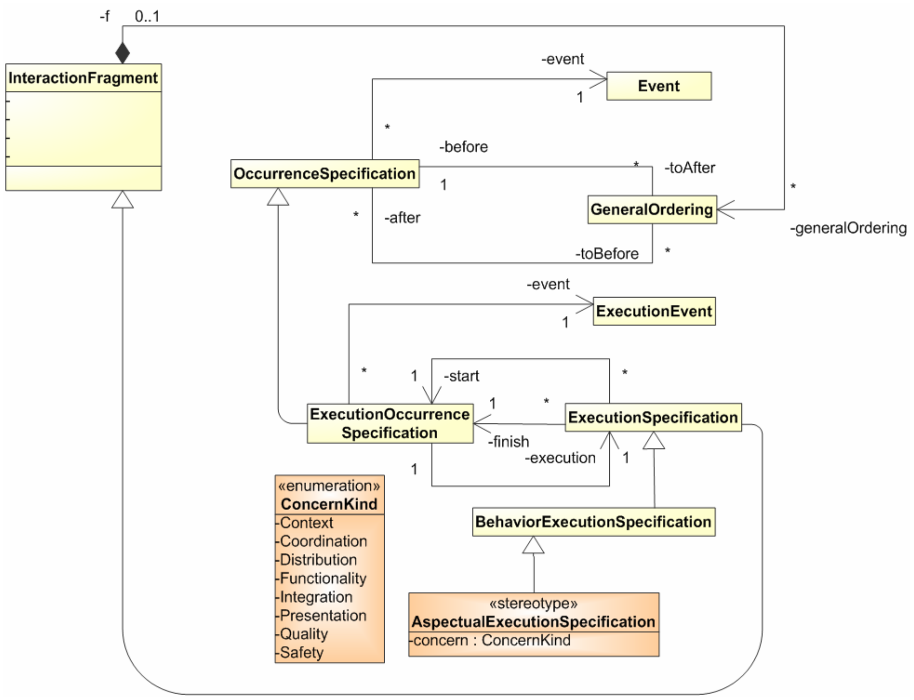


Figure 7-7 Extending BehaviorExecutionSpecification to support concern specification

UML uses *ExecutionSpecification* as an abstraction to describe either an `Action` or a `Behaviour` associated to a Lifeline. The former describes an an atomic change of state of the Lifeline. The second can be used to describe changes of state that take place over time. For ATRIUM Scenarios, the second alternative has been selected since a message is in charge of describing the request of a service that can be a complex one.

In addition, these *BehaviorExecutionSpecification* has been customized in ATRIUM scenarios to facilitate the legibility of the scenario, concretely, to provide more information about system concerns. For this reason the stereotype «AspectualExecutionOccurrence» along with its *concern* tagged value, typed by the enumeration *ConcernKind*, have been included in the metamodel of the Figure 7-3. This permits to describe that a service execution occurrence can be catalogued according to a specific concern. This *AspectualExecutionOccurrence* is graphically described by means of a code colour which is presented in Table 7-4, though it can be customized according to the user preferences by adding new kinds of concerns.

Table 7-4 Concerns for ATRIUM

Concern	Prefix	Code Colour
Context Awareness	C{Context}	⬜
Coordination	C{Coordination}	🟩
Distribution	C{Distribution}	🟪
Functional	C{Functional}	🟨
Presentation	C{Presentation}	🟧
Quality	C{Quality}	🟥
Safety	C{Safety}	🟧

An example of how an *AspectualExecutionOccurrence* is used appears above in Figure 7-6. We can observe how the ExecutionOccurrence, which is fired by the event occurrence of the message "C{Safety} secure check()", is coloured with orange to highlight it is related to a safety concern.

In a similar way to the Messages, «AspectualExecutionOccurrence» can only be used when they describe the execution of a service in an architectural element. This means that the following constraint must be fulfilled:

Table 7-5 OCL Constraints for AspectualMessages

Element	**Constraint**

BehaviorExecutionOccurrence	**context** BehaviorExecutionOccurrence **inv**
	def allowedConnectableElements:Boolean
	Self.receiveEvent.covered.stereotype.name='Component' or
	Self. receiveEvent.covered.stereotype.name='Connector'

7.3.4 Guards

It is frequently the case that some services cannot be executed unless a given condition is satisfied. In order to provide such expressiveness UML 2.0 uses `Guards`. They describe when a condition must be satisfied for a message to be sent.

To describe a *Guard*, the condition must be enclosed among brackets and be stated just as prior to the occurrence event of firing a message. During its description only values local to the interaction scenario can be employed. An example of its use is shown in Figure 7-8. It is part of a scenario where safety features are dealt with. In this case, a "SafetyNode" has been included, in such a way that both "SafetyNode" and "SUCCnct" check if the final position where the robot has moved is different from that is expected. In this case, a

"FAIL" has to be sent. We can observe the guard "[FinalPos<>Estimated]" express such a condition. The full description of Guard syntax can be found in (UML, 2005).

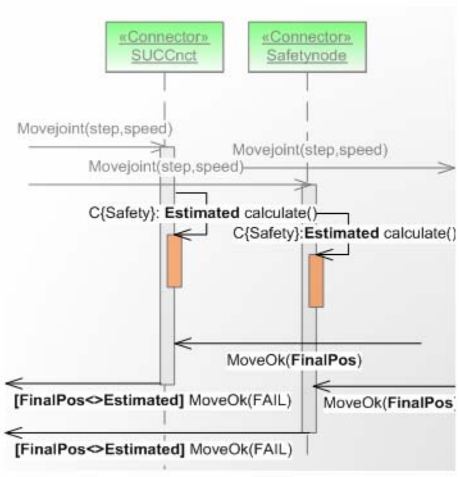


Figure 7-8 Describing Guards in a Scenario

7.3.5 Interaction

An `Interaction` encapsulates a unit of behaviour which is described by means of the connectable elements (*InteractionFragments*, *Lifelines* and *InteractionUse*) along with the set of messages they interchange. Taking into account that every message has associated a pair (*SendEvent*, *ReceiveEvent*) of *OccurrenceEvent*, the set of messages described in an interaction establish a set of *OccurenceEvents*. The set of *OccurenceEvents* is the `trace` of the Interaction. This trace is highly meaningful because it gives semantics to the Interaction. The semantics of the Interactions are compositional, that is, the semantics of an Interaction is mechanically built from the semantics of its constituent *InteractionFragments*. These fragments are described by means of the set of *OccurenceEvents*.

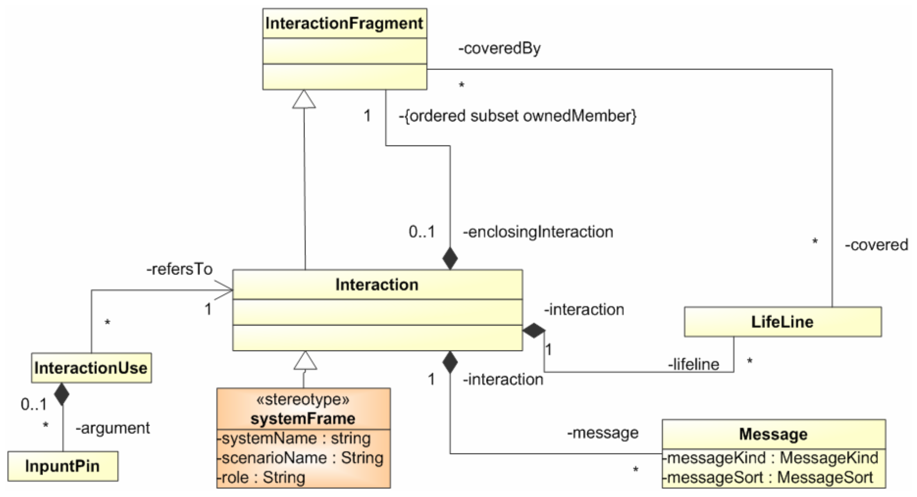


Figure 7-9 Extension of Interaction for specifying systems

As was described in section 7.2, one of the interacting elements necessary for scenarios description are Systems, because they provide the analyst with the ability to decompose the description of a Software Architecture with different granularity levels. It could have been represented like a lifeline in a similar way to components and connectors. However, the main problem is that their description is performed by means of a set of architectural elements that interact to provide the system behaviour. Due to this fact, the most straightforward notation to specify them is by means of a specialization of *Interaction* so as to introduce them as another interacting element.

With this aim, a new stereotype, called «systemFrame», has been included as an extension of *Interaction* that contains in its definition three tagged values: *systemName*, *scenarioName* and *role*. The *systemName* is used to call the system and the *scenarioName* to name the scenario. We have to take into account that a system can have associated several scenarios, in terms of the collection of services it has to provide.

An example of this notation is shown in Figure 7-10. It depicts a system called "MUC", which is composed by the shallow-components "WristActuator" and "WristSensor" and the shallow connector "WristCnct". In addition, Figure 7-10 shows how this system provides the service Move(Joint, step, speed) by means of the interaction of those architectural elements.

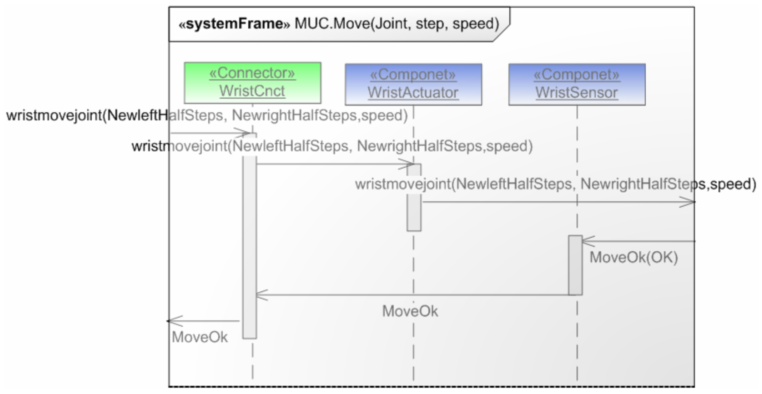

«systemFrame» MUC.Move(Joint, step, speed)
«Connector» WristCnct
«Componet» WristActuator
«Componet» WristSensor
wristmovejoint(NewleftHalfSteps, NewrightHalfSteps,speed)
wristmovejoint(NewleftHalfSteps, NewrightHalfSteps,speed)
wristmovejoint(NewleftHalfSteps, NewrightHalfSteps,speed)
MoveOk(OK)
MoveOk
MoveOk

Figure 7-10 Describing the MUC system for the teachMover

7.3.6 InteractionOccurrence

It is a facility of UML 2.0 that helps the analyst to make an *Interaction* more legible and reusable. In this sense, an Interaction fragment can be factorised as an *Interaction* or *CombinedFragment* facilitating their reuse in several Interactions, i.e., in several ATRIUM scenarios. Whenever its content has to be copied to an Interaction, an *InteractionOccurrence* is used to reference it. The only restriction to be satisfied is that set of actual gates (see section 7.3.8 for more details about gates) of the *InteractionOccurrence* must match the formal gates of the referred *Interaction*.

Figure 7-11 shows an example of an *InteractionUse,* which is depicted by means of a frame labelled with *ref*, has been used to refer the *Interaction* "systemFrame. Base.Move(BaseHalfSteps, speed)". As can be observed the name of the *InteractionUse* has to match the name of the *Interaction*. In addition, we can observe that the actual gates, which appear on the *InteractionUse*, are coincident with the formal gates described in the "systemFrame. Base.Move(BaseHalfSteps, speed)" *Interaction*.

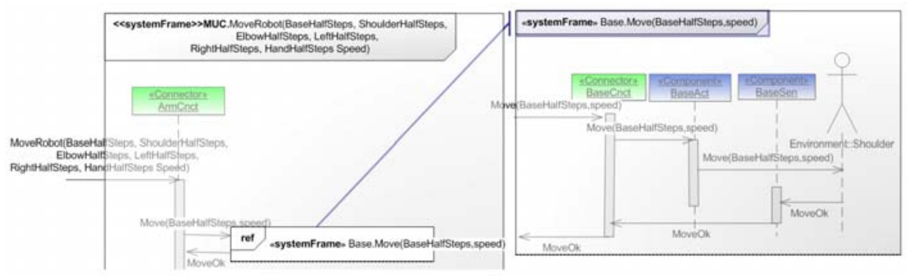

<<systemFrame>>MUC.MoveRobot(BaseHalfSteps, ShoulderHalfSteps, ElbowHalfSteps, LeftHalfSteps, RightHalfSteps, HandHalfSteps Speed)
«Connector» ArmCnct
MoveRobot(BaseHalfSteps, ShoulderHalfSteps, ElbowHalfSteps, LeftHalfSteps, RightHalfSteps, HandHalfSteps Speed)
Move(BaseHalfSteps,speed)
ref
«systemFrame» Base.Move(BaseHalfSteps,speed)
MoveOk
«systemFrame» Base.Move(BaseHalfSteps,speed)
«Connector» BaseCnct
«Component» BaseAct
«Component» BaseSen
Move(BaseHalfSteps,speed)
Move(BaseHalfSteps,speed)
Environment::Shoulder
Move(BaseHalfSteps,speed)
MoveOk
MoveOk
MoveOk

Figure 7-11 Using an InteractionOccurrence to refer the Base.Move(BaseHalfSteps, speed) systemFrame

7.3.7 Combined Fragments

`Combined Fragments` are a new addition of the UML 2.0 notation to describe a graphical boundary for a diagram. Concretely, they are used to provide analysts with notations that help them to simplify the specification of scenarios. They exploit a new graphical element called `frame` to specify this boundary, as Figure 7-12 shows. This notation has different uses that are described below.

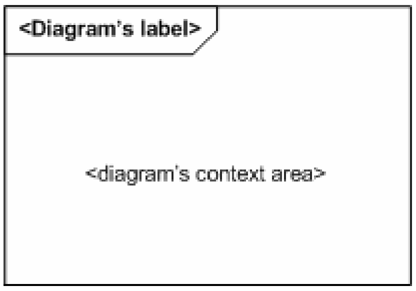


Figure 7-12 Frame description in UML 2.0

Alternative interactions

A notation used with frames is the interaction operator `alt`. It designates that each combined fragment represents a choice of behaviour within the system-to-be where at least one of the services will be executed. Several fragments can be introduced to specify an alternative. Each one must have a guard expression at top that if it is positively evaluated then the corresponding fragment is executed. A fragment guarded by *else* must always be included. It designates a guard which negates the disjunction of all other guards. An example of how *alt* can be used is shown in Figure 7-13. Two fragments are included: the first is guarded by the condition “Joint<> “Tool” so that the service “Move(Joint, step, speed)” of the MUC system is executed if this condition is positively evaluated; the second fragment is guarded by “else” so that the service “Move(step, speed)” of the ToolSuc system is executed if “Joint<> “Tool”“ is negatively evaluated.

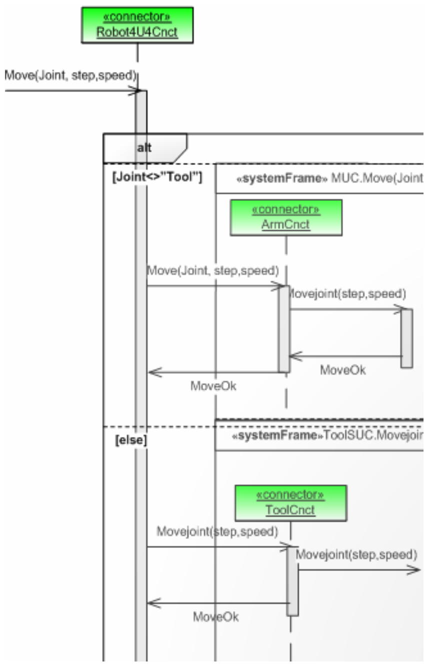


Figure 7-13 Alternative behaviour when defining the movement of a joint

Optionality

In a similar way to alternative interaction optional interactions, defined by means of the interaction operator `opt`, can be used to specify a behaviour which happens only if the condition is positively evaluated; otherwise, there is no alternative behaviour to happen. Therefore, it is used to model a “if then” selection of behaviour with no “else” alternative.

Parallel composition

While describing Software Architecture it is frequent the case that concurrent tasks have to be described. It facilitates that computation to perform a complex task can be split into the processing required to perform several more simple tasks. They are handled in parallel by the system. This means that the behaviour of the system is the merge between the behaviours performed by the set of more simple tasks.

In order to offer notation for parallel composition, UML 2.0 employs the combined fragment element by using `par` as interaction operand. This interaction operand is placed into the sequence diagram's label. In addition, the diagram's context is divided into as many fragment interaction as need. Each fragment represents a thread of execution done in parallel.

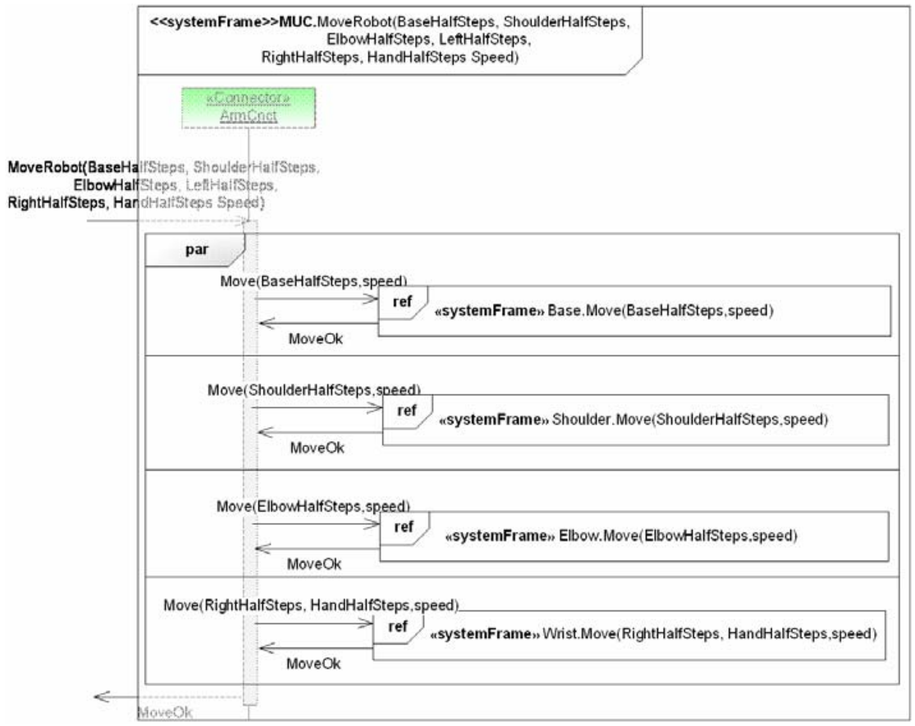

<<systemFrame>>MUC.MoveRobot(BaseHalfSteps, ShoulderHalfSteps, ElbowHalfSteps, LeftHalfSteps, RightHalfSteps, HandHalfSteps Speed)
«Connector» ArmCnct
MoveRobot(BaseHalfSteps, ShoulderHalfSteps, ElbowHalfSteps, LeftHalfSteps, RightHalfSteps, HandHalfSteps Speed)
par
Move(BaseHalfSteps,speed)
ref
«systemFrame» Base.Move(BaseHalfSteps,speed)
MoveOk
Move(ShoulderHalfSteps,speed)
ref
«systemFrame» Shoulder.Move(ShoulderHalfSteps,speed)
MoveOk
Move(ElbowHalfSteps,speed)
ref
«systemFrame» Elbow.Move(ElbowHalfSteps,speed)
MoveOk
Move(RightHalfSteps, HandHalfSteps,speed)
ref
«systemFrame» Wrist.Move(RightHalfSteps, HandHalfSteps,speed)
MoveOk
MoveOk

Figure 7-14 Parallel composition

An example of how this combined fragment can be used is presented in the Figure 7-14. It shows four fragment interactions that execute in parallel the services "Move" provided by the systems "Base", "Shoulder", "Elbow" and "Wrist". As observed in the figure, these systems are referred by means of an Interaction Occurrence.

Iterations

UML 2.0. has introduced a notation to describe when a fragment interaction has to be executed a number of times until a condition is positively evaluated.

In this case, the interaction operand, which is placed in the diagram's label, is `loop`. Inside the diagram's context area, a guard is placed to describe the condition that determines the end of the execution. Under the guard, the fragment iteration to be iterated over is specified.

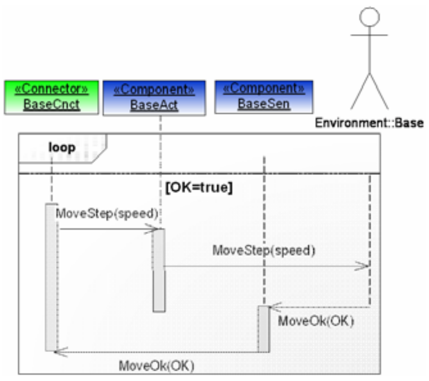


Figure 7-15 Describing a loop for the movement of the Base System

In addition, a loop can have two more special conditions: *minint* and *maxint*. These conditions and the boolean condition are jointly evaluated to determine if the next iteration can be performed. *minint* condition is employed to establish that the interaction fragment has to be executed at least the number of times indicated; whereas *maxint* determines that the number of executions cannot exceeds that number. Once the fragment interaction has been executed *minint* times, if the boolean expression is negatively evaluated the loop terminates; else, the iteration continues until it iterates maxint times or the boolean expression is negatively evaluated. Figure 7-15 describes an example of loop use. As is observed, the service "MoveStep" is requested to the "BaseAct" component, until the response OK is not true.

7.3.8 Gates

When an ATRIUM scenario is being defined is usual that it can not be defined by itself but interacting with its context. This implies that is necessary a mechanism to describe what information is passing between them. With this aim, UML 2.0 has introduced the concept of `Gate`. It is in charge of describing the connection point of a message, concretely, when an end, which is actually the gate, of a message is connected to the sequence diagram frame's edge and its other end is connected to a lifeline.

A gate can have different roles depending on the sequence diagram frame's edge is covering (Figure 7-16). It can be a *formalGate* when it is described on an *Interaction*. This kind of gates is not going to be used, at least not usually. It is because an ATRIUM Scenario can describe several InteractionFragments. The messages they interchange do not use the gates to establish the communication but directly can be connected to the lifelines that the InteractionFragments enclose (see Figure 7-34). It can also be an *actualGate* when it is described on an *InteractionOccurrence*. Its use is appreciated in the Figure 7-14, where the message "Move(BaseHalfSteps,speed)" has the *InteractionOccurrence* named "«systemFrame» Base.Move(BaseHalfSteps,speed)" as one of its ends. Finally, a *cFragmentGate* when it is described on a *CombinedFragment*.

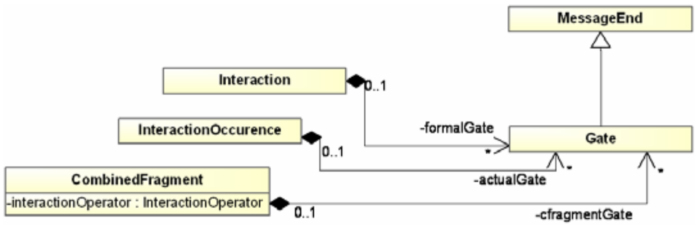


Figure 7-16 Describing Gate in UML 2.0

7.4 ARCHITECTURAL STYLES AND PATTERNS

The exploitation of Software Patterns is a current trend in Software Engineering. It represents a meaningful way to reuse knowledge in software development, especially, when the analyst does not have a proper background in the area. For this reason, their introduction means a meaningful advantage in terms of costs, time and quality of the final product. As (Schmidt et al., 2000) state:

"Patterns expose knowledge about software construction that has been gained by many experts over many years. All work on patterns should therefore focus on making this precious resource widely available. Every software developer should be able to use patterns effectively when building software systems. When this is achieved, we will be able to celebrate the human intelligence that patterns reflect, both in each individual pattern and in all patterns in their entirety.

They are generative. They tell us what to do; they tell us how we shall, or may, generate them; and they tell us too, that under certain circumstances, we must create *them. Each pattern is a rule which describes what you have to do to generate the entity which it defines. (pp. 181-182)"*

Patterns have been defined to several levels. The most well known proposal in this sense has been presented by (Gamma et al., 1995). They describe a set of patterns, at the design level, for managing object creation, composing objects into larger structures, and assigning responsibilities to objects, in the context of object-oriented systems.

However, how does this reuse of the knowledge is applicable while specifying the Software Architecture? Several alternatives state the advantages of its application at this stage. Among them, the use of `Architectural Styles` is a meaningful and widely extended approach. (Shaw & Garlan, 1994) have described an Architectural Style as follows:

"An Architectural Style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an Architectural Style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition."

We can find this concept with a different name: `Architectural Pattern`. They have quite similar as can be observed by the (Douglass, 2003)'s definition:

"An architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them."

Both of them refer to recurring solution described at the architectural level. However, some authors have established some differences among them, mainly in terms of how they have described because *Architectural Patterns* follow a problem-solution approach, similar to (Gamma et al., 1995), whereas the *Architectural Styles* focus on the architectural description. (Avgeriou & Zdun, 2005) give a deeper insight about this topic.

As may be observed, the definitions described above consider the application of an *Architectural Style* as an important decision in terms of the final configuration of the system-to-be. This is because it influences the way the relationships between the subsystems must be established. As (Eden & Kazman, 2003) set out, an Architectural Style *pervades all the parts of* the system because it defines a set of properties that must be satisfied by every element used in its definition. Among these properties, it is especially relevant the ability to assign responsibilities to the subsystems collaborating in the specification. The identification of the employed Architectural Style conveys much information about the system and the made decision to the stakeholders. (Abowd et al, 1995) describe a clear example when given a description, as Figure 7-17 shown, it can have different interpretations respect to the applied Style. They describe that if it were with interpreted respect to a client-server Style, it would mean that there are two kinds of elements: client and server. If it were interpreted using a blackboard style, it would describe a blackboard accessed by three sources of knowledge, etc.

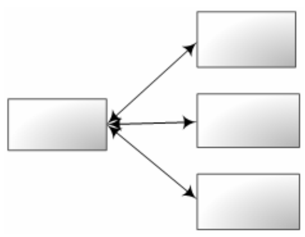


Figure 7-17 Software Architecture Description

However, the reuse of past experience, while defining the Software Architecture, cannot be limited to the application of Architectural Styles but it is also necessary to introduce partial solutions to deal with specific problems. For this reason, the introduction of `Design Patterns` at the architectural level emerges as mandatory to help the analyst in this process. As opposed to (Gamma et al., 1995)'s proposal, the *Design Patterns* here described are oriented to identify which architectural elements can collaborate to offer a solution to a specific problem. Their application does not reorganize the architecture, but rather extends and changes partially its behaviour. In this sense, they usually add one or a few *components* (or *aspects)* that realize that behaviour provided by the patterns. In this sense, these patterns are going to deal with features such as distribution, persistence, concurrency, etc, i.e., those other concerns of the system.

The *Design Patterns* are not predominant in the architectural definition so that it can be merged with *Architectural Styles* without problems. They can emerge when the same architectural elements are affected by more than a *Design Pattern*, especially, if they are dealing with the same concern.

The definition of the *Define Scenarios* activity (section 7.5) of ATRIUM allows for the exploitation of *Architectural Styles* and *Design Patterns*. For this reason, some of the most well known Architectural Styles and some examples are presented in the following sections. It is described its use and impact in the architectural specification along with the problem they face.

7.4.1 Architectural Styles

Several works have tried to define and catalogue the existing Architectural Styles, such as (Garlan et al., 1994) or (Shaw & Clements, 1997). However, they do not provide a common framework for its identification. In this sense, (Bass et al., 2003) have identified the following set of features that can be used to characterize an Architectural Style:

– a set of elements types (e.g., data repository, process, procedure) that perform some function at runtime,

– a topological layout of these elements indicating their interrelationships,

– a set of semantics constraints (for example, two knowledge sources in a blackboard style can not communicate directly),

– a set of interaction mechanisms (e.g., subroutine call or remote procedure call (Mehta et al., 2000)) that describes how the elements communicate, coordinate, or cooperate through the established topology.

Considering these features, they have described five types of *Architectural Styles*. They are introduced below along with their most representative example, described more exhaustively in the following sections.

– `Data-Centred`. This type of Style is related to those systems exploiting a central repository of data to facilitate the communication and synchronization of their multiple components. An example of this type is the `Blackboard Style`.

– `Data Flow.` This type of Style deals with those systems focused on how streams of data are successively processed or transformed by components. `Pipe-and-Filter` is a clear example of this type.

– `Call-and-Return` deals with how complex and heterogeneous systems must be decomposed into interacting parts to facilitate their comprehension and implementation. The most well known and widely used example of this type is the `Layered Style.`

– `Independent Components` focuses on systems whose individual components exchange messages to perform the main computation but keep their independence. `Event Style` is an example of this type.

– `Virtual Machine.` This type of Style is concerned with how systems offer an abstraction layer that is exploited by the computation infrastructure. `Interpreter Style` is a realization of this type.

These types of Styles have been defined in a general way, without considering a particular domain or application. However, lately another type of Style is getting more and more attention, called `Domain-Specific Software Architectures (DSSA)` (Mettala & Graham, 1992). This kind of Style is based on the idea of identifying common elements that are shared by a family of systems so that new systems can be created by instantiating them. This means that a pre-assignment of responsibilities, in terms of the constraints they have to preserve, is given to the architectural elements they identify. This type of Style has also been considered in the description of ATRIUM. For this reason, an example is introduced. Specifically, the ACROSET style, described by (Ortiz et al., 2005), is presented, because it has been put into practice for the Teachmover and the EFTCoR definition.

Blackboard Style

Some research fields, as for instance the Artificial Intelligence in the context of problem solvers, are demanding mechanisms of coordination able to deal with the heterogeneity of the coordinated processes and the high load due to the volume of processes. In this context, the *Blackboard Style* offers a solution by exploiting a common structure for the insertion and removal of information that facilitates an inter-process communication and a decoupling between the communicating elements. (Pfleger and Hayes-Roth, 1997) provides a good overview of this Style from the point of view of the Software Engineering.

Typically, an Architecture realizing this style is separated into a central and fully reachable data structure that is called *blackboard*. This is a global data structure employed as communication medium. It can store a wide range of different kinds of information, although information related to the solution or a partial solution is the most usual. A variable number of architectural elements, named *knowledge sources* (KS), access to this structure to perform their computation based on its content and modifying it as needed. In this way, the solution is build by modifying the information store in the *blackboard*. These *KSs* can operate on the *blackboard* only when they are enabled. The *control* is in charge of this task. It decides which element or elements are enabled to be executed depending on the kind of strategy to apply, that is, if several KSs can operate in parallel or only one KSs can operate at a time.

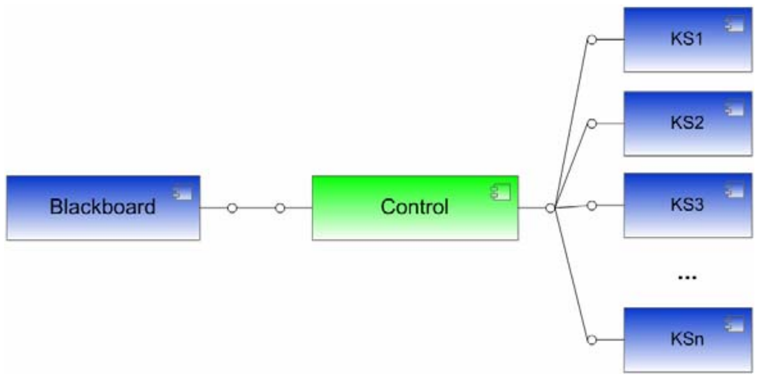


Figure 7-18 A common structure for the Blackboard Style

Figure 7-18 shows what the typical layout looks like. It can be observed that the *knowledge sources* are directly connected to the *Control*. This element is in charge of deciding which ones are enabled to monitorize the access to the *Blackboard*. Taking into account that layout, the Figure 7-19 shows an archetype of an ATRIUM Scenario while instantiating this Style. It shows how the architectural elements interact and which roles are playing each one. In this example, each *KS* maintains its own state and evaluates the precondition using it. The *Control* determines which KS must be enabled taking into account such preconditions and the current state of the blackboard.

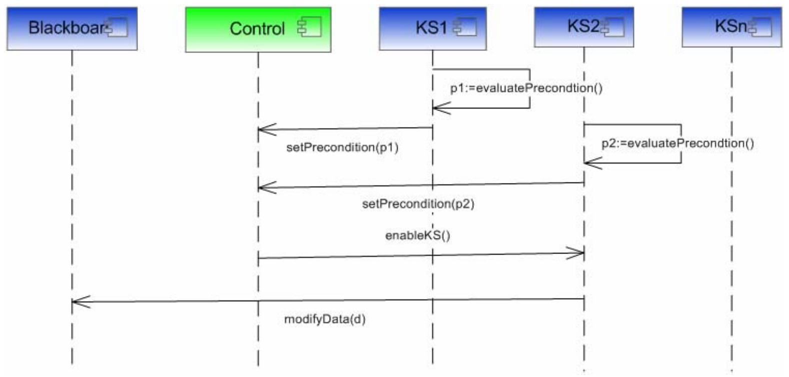


Figure 7-19 ATRIUM Scenario applying a Blackboard Style

Several constraints must be applied when this Style is applied:

- KSs cannot communicate directly but by means of the blackboard. This means a high decoupling between the implied elements, without any information about how they have defined.

- KSs operate by self-invocation when there is a specific state of the solution that enables them to work. This means that they are not explicitly called but they wait for a set of conditions to operate.
- There is not an explicit order of execution of the KSs nor the examination or manipulation of the items in the blackboard. This implies that it is up to the run-time to decide which execution order to apply using the current state established in the blackboard.

Taking into account how this Style has been described, the used communication mechanism is obviously by means of direct data access. It is because all the communication and coordination of the KSs is supported by means of the *blackboard*.

Pipe and Filter Style

This Style is exploited when there is a need in terms of the abstraction of the route between two elements with a dependency relationship between them, that is, one of them has an attribute whose value must be bound to the value of an attribute of the other element. The main goal of this abstraction is to establish the appropriate synchronization mechanisms. It also provides a low decoupling between the elements because if one of the elements suffers any modification their surrounded elements are not affected by it. (Abowd et al, 1995) offers a good description and formalization of this Style.

When this style is used, two kinds of elements are defined: *pipes* and *filters*. The *source* is a filter element forwarding the data *sourceData* to be processed, i.e., the required values. The *destination* is a filter element that processes the received data *destData*. Both elements are connected by means of a binary relationship 1:1 called *pipe* that abstracts the transportation mechanism.

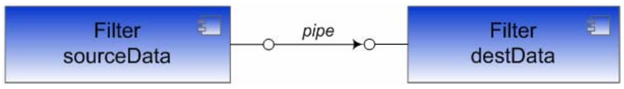


Figure 7-20 Main structure of the Pipe and Filter Style

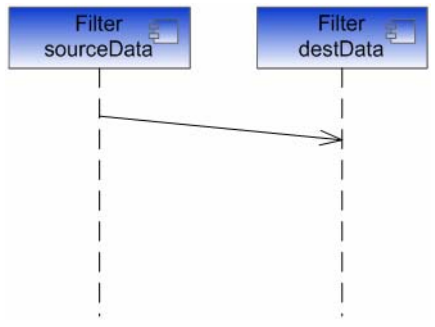


Figure 7-21 Applying the Pipe and Filter Style in an ATRIUM Scenario

This Style does not impose any topological constraint, that is, every *filter* can be connected to any other but always using a *filter* as communication medium, as the Figure 7-20 shows. For this reason, its instantiation in an ATRIUM Scenario is quite simple. Figure 7-22 shows that only the interacting *filters* must appear in the scenario description. The *pipe* is implicitly represented by the sequence of messages that both filters interchange.

Several constraints have been defined that any Software Architecture applying this Style must satisfy:

- *Pipes* cannot process the data that are being transferred from one element to the other.
- *Pipes* do not allow (by default) full duplex communication between filters, only one-way.
- *Filters* are in charge of transforming the input data by a single computation step. This processing is performed following the specific order of arrival of the data.
- Their control mechanism is asynchronous facilitating that pipes can work in parallel in an independent way.

There is no a specific interaction mechanism for this Style. However, the most well know approach is to reuse Unix operating system primitives to handle task scheduling, synchronization, and communication through Unix pipes.

Layered Style

This Style provides an abstraction for systems having specific needs in terms of dependencies between components. Specifically, when high-level components are dependent on lower-level components to perform their main computation. Thus, a vertical decoupling between these components is mandatory to provide these systems with the necessary modifiability, portability and reusability. Usually, a horizontal structure, in addition to the vertical one, is employed to describe specific assignment of responsibilities as well.

In order to provide these vertical and horizontal structures *Layers* are used. Each *layer* is composed by a set of *components*, having all of them the same abstraction level, that interact by means of *connectors*. Each *layer* can use the services of the layers immediately above and below it through their clearly defined interfaces.

The main layout of this Style is shown in the Figure 7-22. It depicts the horizontal structure that is integrated by several components and connectors. It also illustrates that each layer access to the layer beneath through a clear interface. Figure 7-23 shows what an ATRIUM Scenario looks like when a Layered Style is applied. It can be observed that architectural elements beloging to different layers can interact between them. It means that interfaces must be described between those layers to facilitate that interaction.

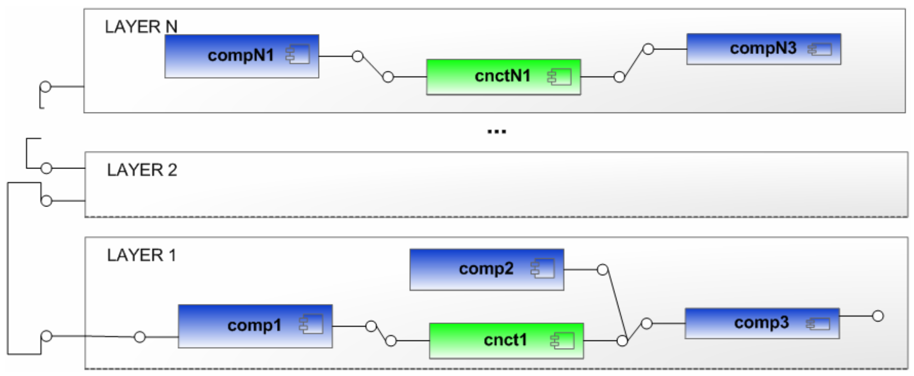


Figure 7-22 Layout for the Layered Style

The main constraint exhibited by this Style is related to the access between layers. Each layer should access to the layer just bellow it. If this constraint is not satisfied the main benefits of this Style, as its modifiability, would not be achieved.

Several communication mechanisms can be used for this Style. However, the most widely used is the procedure call to perform the transfer of data by means of parameters. This mechanism also provides facilities for the coordination of the interacting components.

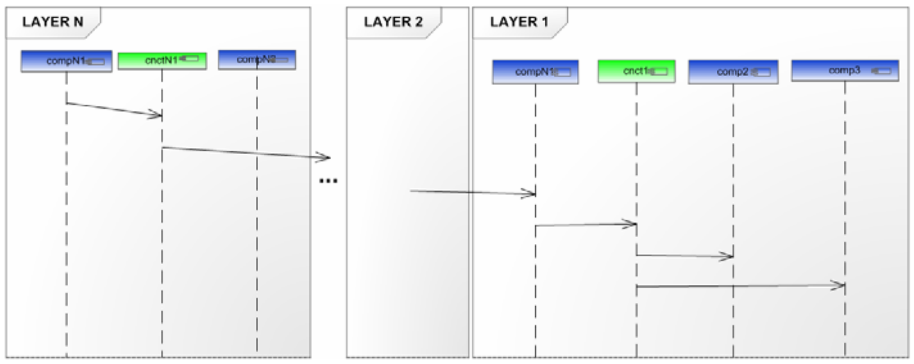


Figure 7-23 Layer Style applied to an ATRIUM Scenario

Event-based Style

Event-based Style has been widely accepted for the academia and industry and extensively used in the design of distributed systems. This has been mainly motivated for the low coupling it provides. In an Event-based Style, the components communicate by generating and receiving event notifications. The powerful of this Style is that neither the generation nor the reception of events is directed to a specific component. This means that components are not designed to work with other specific components, which facilitates a low decoupling between them. It also provides a high degree of adaptability because any specific component must be introduced to generate or receive the produced events. In addition, this Style facilitates the integration activities by using the implicit invocation of tools, subsystems or components as response to the notification of an event. Finally, it also can provide meaningful advantages in terms of scalability, but depending on how the notification service is designed. (Fiege, 2005) give a good overview of systems applying this Style.

A connector named *EventService* is one of the main elements of this Style. This element is in charge of dispatching the event notification, that is, when an event has been generated. The interacting components can play two different roles. If they play as *subscribers* then they advise the *EventService* their interest in the reception of event notification by sending a subscribe request. If they play as *publishers* then they notify the occurrence of an event by sending a publish request to the *EventService*. As response, the *EventService* is in charge of dispatching the event notification to all the *subscribers* to that event.

Figure 7-24 depicts the typical layout for an Event-based Style. It can be observed how the publish request flows to the *EventService* and how it is notified to every subscriber. Figure 7-25 illustrates an ATRIUM Scenario applying this

Style. It can be observed that *subscribers* do not interact directly but by means of the *Event Service*. It notifies to the subscriber when a subscribed event has been published.

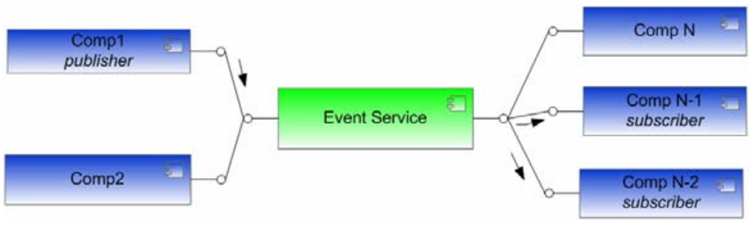


Figure 7-24 Typical layout for an Event-based Style

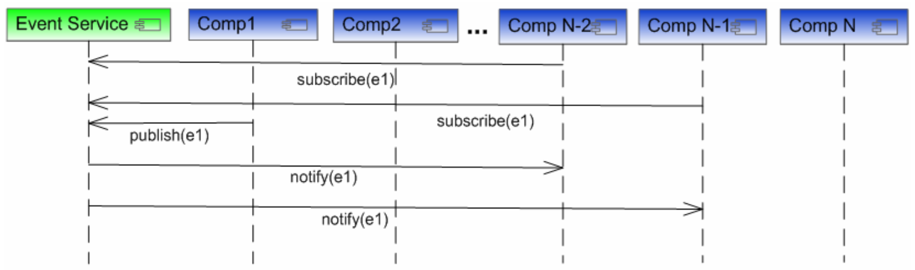


Figure 7-25 Event-based Style applied to an ATRIUM scenario

This Style has as main constraint how the components are specified. They cannot have any reference to any existing or predetermined component in their definition. Otherwise, the adaptability of this Style would not be achieved. Other constraint to be satisfied is the dispatching assurance, i.e., every subscriber must have an instantaneous notification of those events it has subscribed for.

Events are used in systems applying this Style as a communication medium. It is because they can content a wide range of information such as time, source, or any other domain-related information. They are also the coordination mechanism used for these systems because they are the base to model the control flow in these systems.

Interpreter Style

Whenever the configurability is a concern of the system, this Style offers an appropriate solution facilitating the adaptation of the system to unexpected situations. The main idea exploited by this Style is to convert the functionality of the system into data, usually, in a Metamodel. This Metamodel can be interpreted in a way that can be changed as needed and thus, the behaviour of the system, by modifying the Metamodel. However, this Style exhibits a severe performance penalty for its use.

The typical structure of this style is described by means of one *interpreter*. It is the execution engine in charge of reading and modifying the Metamodel as needed. Three memories are also used in their description: *program* that describes the Metamodel being interpreted; *program state* that represents the current state of the program; and, the *engine state* that describes the current state of the *interpreter*.

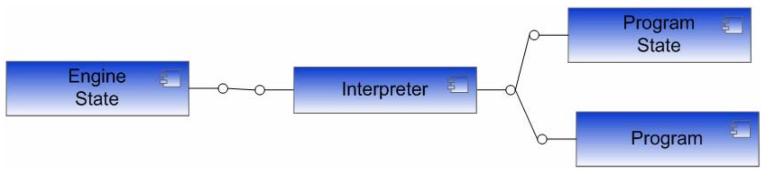


Figure 7-26 Layout for an Interpreter Style

There is not a typical layout for this Style. Figure 7-26 identifies the main elements previously described, but there is no constraint about how this Style must be decomposed. In this sense, there are not specific constraints for its application either. This is also applicable to the communication and coordination mechanisms. An ATRIUM Scenario applying this Style is presented in Figure 7-27. It describes how the interaction proceeds by evaluating the *next Expression* and determining the next state of the *Interpreter* based on its current state and the state of the program being evaluated.

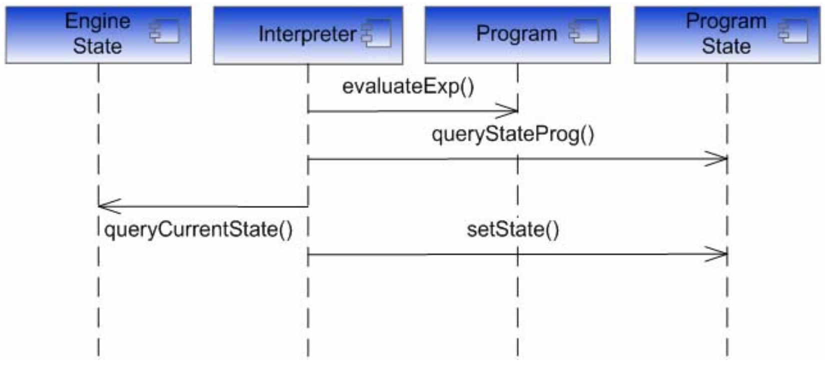


Figure 7-27 An ATRIUM Scenario with the Interpreter Style

ACROSET DSSA

Tele-operated systems cover a broad range of mechanisms and missions, such as hull ships cleaning, bomb deactivation, etc. However, they share some common characteristics as, for instance, a behaviour controlled by an operator; or working areas fixed and well known, etc. These characteristics facilitate the description of a Domain Specific Software Architecture. An example of this approach is ACROSET (Ortiz et al., 2005). It has been described as a specialization of a Layered Style where an assignment of responsibilities to each layer has been established.

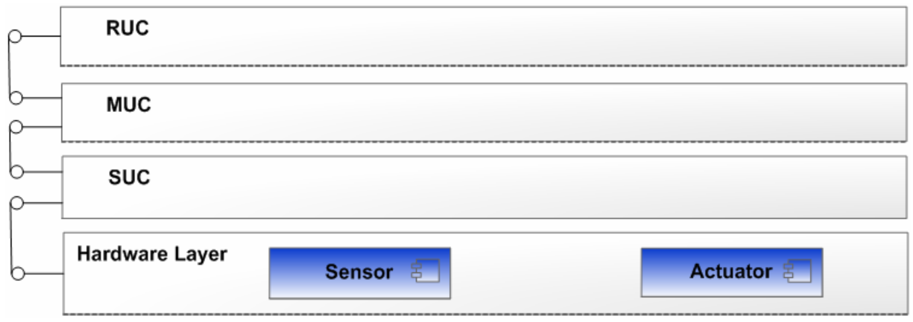


Figure 7-28 Main layout of ACROSET

ACROSET identifies four layers that Figure 7-28 depicts. The lower abstraction layer is the *Hardware Layer* that is integrated by a set of sensors and actuators, usually implemented by means of hardware components. The sensors are components that provide information that is required for controlling the active elements of the system. The actuators are components in charge of modelling these active elements, for instance, one of the joints of the Teachmover (see section 4.2.2).

The next layer is called *Simple Unit Controller* (SUC). It controls the actuator and processes the data received from the sensors that are in the layer beneath it. For instance, in the Teachmover there will be a SUC for every joint generating the commands for the actuator according to the information it receives from the sensors. This layer must describe clearly the control policy it implements.

The *Mechanism Unit Controller* (MUC) is at the third level. This layer is defined by means of an aggregation of SUCs that it must control according to the information they provide and the control policy it implements. It provides a control for a whole mechanism. For instance, in the Teachmover, it is in charged of controlling the arm of the robot.

Finally, the last layer is called Robot Unit Controller (RUC) and controls the whole behaviour of the robotic unit by managing the set of MUCs below it. For instance, it controls the wrist and the arm in the Teachmover example. It is also in charge of implementing the control policy of the robotic unit.

Figure 7-29 shows a typical scenario while applying this DSSA. It can be observed that an environment element it is included to represent the active elements that must be controlled. In addition, it typically uses two software components, *actuator* and *sensor*, to establish the communication with it.

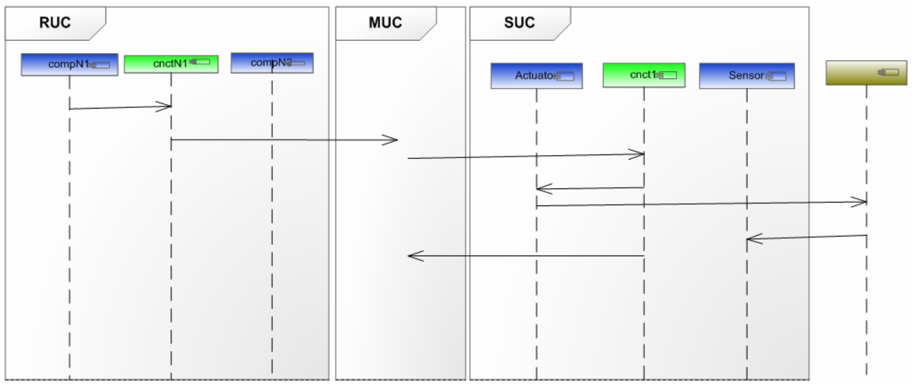


Figure 7-29 ACROSET Style in an ATRIUM Scenario

7.4.2 Design Patterns

They address smaller reusable designs than Architectural Styles, such as the structure of subsystems within a system. For this reason, they are referred to as *microarchitectures* sometimes. They differ from Architectural Styles in they do not affect the fundamental structure of the application, although they can have a strong influence on the structure of a subsystem. Their description is usally in terms of communicating elements that play a customized role to solve a specific problem.

In the next section, some design patterns are introduced in order to provide an overview about how their definition must be to be incorporated in ATRIUM. However, we should emphasize that the design patterns introduced in the next section are not similar to that presented by (Gamma et al., 1995). On the contrary, the following design patterns are described in terms of shallow components and connectors because they are the elements used in an ATRIUM Scenario. The described patterns also incorporate the exploitation of the Aspect-Oriented techniques as a lightweight solution. They are specially indicated when some concerns of the system-to-be, such as performance or safety, must be faced.

ATRIUM is aimed at the need of using patterns as a way to describe quality solutions for the scenarios description. However, ATRIUM does not propose an elaborated catalogue of patterns because we have detected that they are highly dependent of the specific domain. For this reason, the described patterns were those detected in the context of the tele-operated systems, thanks to both the development of the EFTCoR and the collaboration of the UPCT. We can call them patterns because they have been used and re-used during the EFTCoR project. They are introduced in the following section to illustrate how these patterns must be described for its applicability in ATRIUM. In addition, one of these patterns is instantiated in section 7.5 to illustrate how it can be exploited.

For the description of these patterns, we have used an adaptation of the Role-Based Metamodeling Language (RBML) for patterns specification, proposed by (Kim et al., 2004). It defines a sub-language for several types of UML diagrams. Specifically, we have exploited the Interaction Pattern Specifications (IPSs) that is used to characterize Interaction Diagrams. An IPS is employed when it is necessary to constraint the allowed interaction between the interacting elements. The two key elements of this notation that have been used are:

– The expression |*s*:|*Subject* represents an instance *s* of an architectural element that conforms to the classifier role *Subject*.

– The expression |*message()* represents a message that conforms to the feature role *message*.

If messages or lifelines do not employ that kind of expression, they describe concrete elements that must be present when the pattern is instantiated.

Safety Patterns

As was presented in the chapter 6, Safety is one of the main concerns of the teleoperated systems. The Industrial Robots and Robot Systems - Safety Requirements standard (ANSI/RIA, 1999) was used as guidance for the elicitation of this kind of requirements. However, what happens when scenarios meeting these requirements must be described? (ANSI/RIA, 1999) tries to give some guidelines to help in this process.

Table 7-6 describes the strategy to apply according to the Risk Reduction Category R2A. As can be observed, this recommendation is not enough to develop concrete systems such as EFTCoR and, in consequence, more detail is needed in order to facilitate this process. Thus, we propose to establish a catalogue of patterns for the reduction or elimination of risks, classified by the ANSI RRC. In order to facilitate its comprehension, we give two concrete examples of patterns considered in the context of the EFTCoR system.

Table 7-6 Description of the ANSI Risk Reduction Category R2A

R2A: Control reliable safety circuitry (based on hardware or software controller or firmware)
The monitoring shall generate a stop signal if a fault is detected. A warning shall be provided if a hazard remains after cessation of motion
Safe state shall be maintained until the fault is cleared.
Common mode failures shall be taken into account.
The single fault should be detected at time of failure.

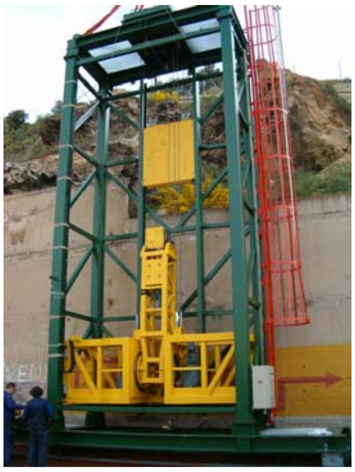


Figure 7-30. Primary Positioning System with both arm joint (yellow) and joint on tracks (green) of the EFTCoR system

The primary positioning system (see Figure 7-30) has a height of twelve meters and a weight of twenty tons that make inevitable the movement of the robot without the consideration of safety requirements. The crane has, in its central zone, an articulated arm of two tons with a secondary positioning system at its end (an XYZ-table that includes a cleaning tool). It is mandatory that the system ensures a safe movement of the arm according to the received commands from the operator. A detailed analysis of the hazard HZ.7 ("the arm of the primary system does not stop") leads us to associate the following sources of error:

- Any sensor integrated with motors, which move the secondary, fails.
- The electrical power is off.
- The control unit does not run correctly (a hardware fail or a software error).

The hazard HZ.7 may imply the breakage of mechanical parts, the precipitation of components to the floor or damages to the human operator (risks R6, R7, R8, respectively). Taking into account the severity of the injury, the frequency of the exposure and the probability of avoidance, the RRC is R2A, which establishes the risk must *cease* if it arises because of its severity.

A pattern was identified to deal with this kind of RRC whose main idea is to introduce a Safety Node to check the movement. It has been defined by means of three ATRIUM Scenarios (Appendix C, section C.1). Figure 7-31 shows one of them. The following description gives a good idea of the way the hazard has been considered:

– When a movement command is received, the "cnct", which is the connector to be "Monitored", forwards it simultaneously to the redundant node dedicated to monitor possible hazards ("SafetyNode").

– The "cnct" node reads from a sensor the current position of "acE", which play the role of the "activeElement", and controls directly its functioning by means "acE", which plays the role of the "actuator". The "SafetyNode" will stop the motor if it detects a malfunction of the motor.

– Just before the execution of any command, the "Monitored"connector sends a message to the "SafetyNode" to notify the starting of the movement. From this time, the "Monitored" connector sends to the "SafetyNode" the current value just read from the "Sensor". The "SafetyNode" computes the curve of the discrete positions that must be reached by the "activeElement". This timed discrete calculus is done by taking into consideration the initial value of the sensor and the command to be executed. Any difference between the calculated values implies an anomaly in the function of the "activeElement" movement. Whenever the "SafetyNode" detects a discrepancy in this value, with respect to the estimation of the position values, an emergency signal is generated and the movement is stopped.

– The "Monitored" connector whose behaviour is similar also does the checking performed by the "SafetyNode". That is, it also stops the movement and generates an emergency signal (see Figure C. 2 in Appendix C, section C.1).

– If there is no error, the movement goes on until the "activeElement" gets the position (see Figure C. 3 in Appendix C, section C.1).

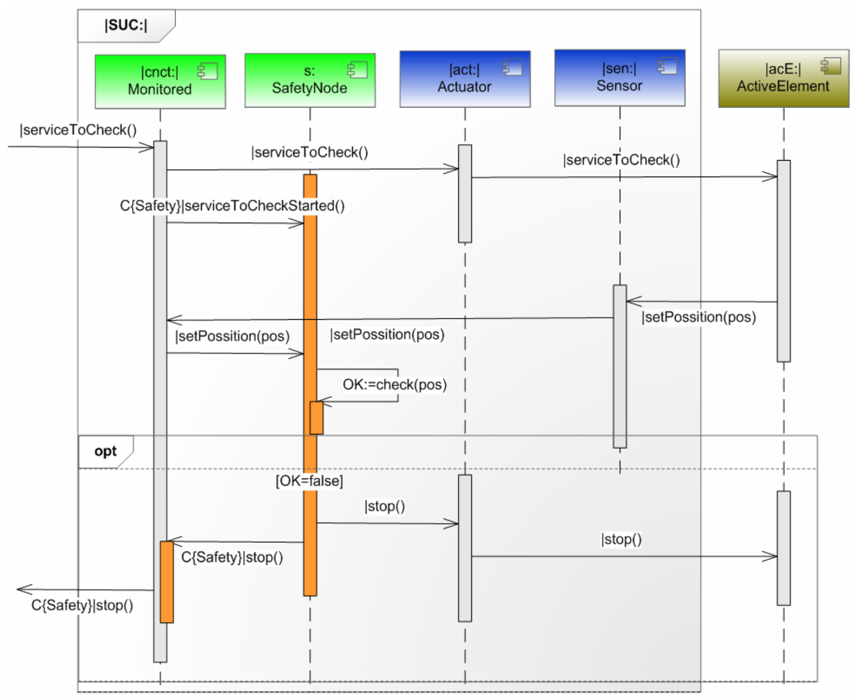


Figure 7-31. Partial description of a safety pattern for R2A Risk Reduction Category

The different error conditions, which could lead to a safe stopping of the robot in the previous example, are the following:

a) Both the design and construction of the robot are done in such a way that, if a global fail of the system occurs then the robot will be mechanically fixed and returned to a safe mechanical state;

b) If any computing node does not work well (due to software or hardware errors) or the communication link fails then the other one will detect the discrepancy in the values.

In this last case, it is essential that the "Monitored" connector periodically reads the sensor data, although there was no current "Monitored" command in execution.

In the chapter 6, when the Safety specification was presented, it was detected that the Risk Reduction Category R2C must be applied. This category deals with those hazards that must be prevented (or ceased) but whose severity is not very high. This means that the system should introduce some mechanism able to manage properly that hazard but it does not need to introduce so strong constraints as the previous one. A pattern that was detected, which can be associated with that category, is the introduction of a Safety Aspect. The main idea is that this aspect would check if the movement is safe, before it is performed.

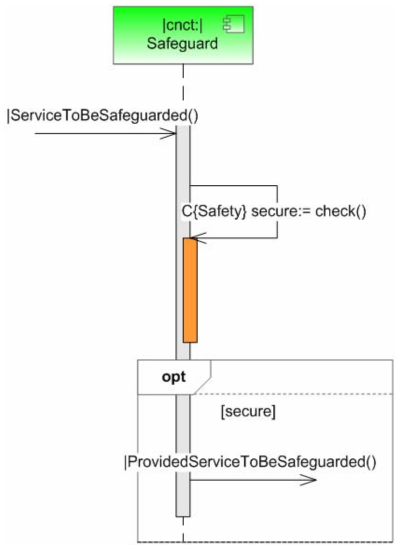

|cnct:| Safeguard
|ServiceToBeSafeguarded()
C{Safety} secure:= check()
opt
[secure]
|ProvidedServiceToBeSafeguarded()

Figure 7-32 An aspect for managing Safety concerns

Figure 7-32 shows the interaction described by this pattern. As can be observed, whenever it must be checked if the movement associated to a service "|ServiceToBeSafeguarded()" must be safe, then the architectural element "Safeguard" must contain a Safety Aspect with a service called *check()* in charge of checking the movement. The movement, "|ProvidedServiceToBeSafeguarded()", will proceed only if it is safe, that is the guard "secure" is possitively evaluated.

7.5 Process for Scenarios Modelling

The objective of the activity *Define Scenarios* is mainly related to the description of the ATRIUM Scenario Model, that is, a set of scenarios described by means of our profile based on UML Sequence Diagrams that describe the main behaviour of the system-to-be. Each scenario is going to provide a partial view of the system.

Figure 7-33 depicts the main tasks to be performed. As can be appreciated, the ATRIUM Goal Model is the main input for the process. Specifically, the operationalizations that were selected by means of the analysis performed in the previous activity (see section 5.4.2). These operationalizations describe textually how these scenarios must be defined.

The process is carried out in two different phases. The first takes into account those operationalizations described for those operationalization contributing to requirements without crosscutting relationships and the second for the remainders. These requirements are going to be, usually, those described as refinements of the *Functionality* concern. This is because the ATRIUM Goal Model specifies when a crosscutting exists between two requirements. Usually, this crosscutting arises between functional (*Functionality*) and quality requirement, as (Moreira et al., 2002) set out. For this reason, one of the alternatives to provide a solution for these requirements is transforming the Functional scenarios, i.e., to modify these scenarios by introducing architectural elements, messages, etc, to meet the quality requirements.

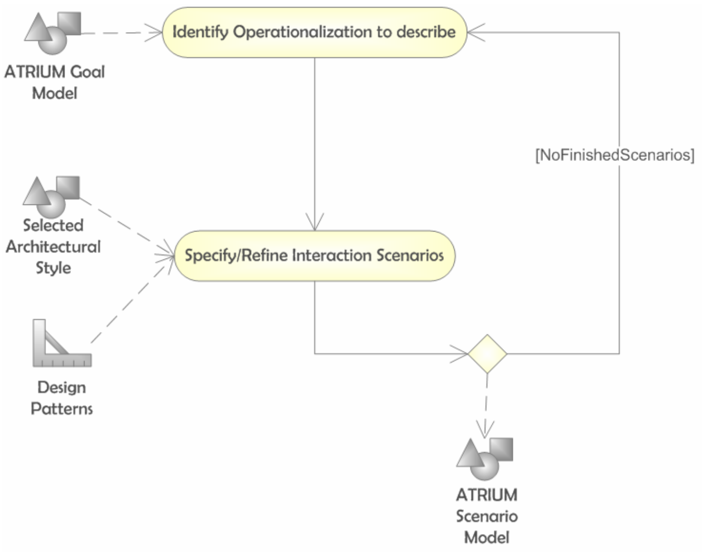

ATRIUM Goal Model
Identify Operationalization to describe
[NoFinishedScenarios]
Selected Architectural Style
Specify/Refine Interaction Scenarios
Design Patterns
ATRIUM Scenario Model

Figure 7-33 Process for Scenarios Description

Therefore, during the first phase, the analyst selects one operationalization, related to one or several functional requirement, and describes its scenario/s. This step has the *Design Patterns* as one of the inputs. They are going to describe solutions that can be reused, as those already presented in section 7.4.2. It must be taken into account that the operationalization already describes that a specific pattern must be introduced, if it was the decision taken, for the definition of the scenarios because the analysis of the alternatives was performed in the previous activity. For this reason, in the step *Specify/Refine Interaction Scenarios* the analyst uses the template that describes the pattern to be reused in order to instantiate it properly.

As can be observed, the *Selected Architectural Style* is another input for the step *Specify/Refine Interaction Scenarios*. This is due to the fact that it is going to give us a sketched view of the system and how the scenarios must be defined. In addition, it gives us a proper detail about the semantics associated to the Software Architecture description.

During the second phase, for each requirement having a crosscutting relationship with a quality requirement we have to enrich the established scenario. The operationalization selected for the non-functional requirement will extends the scenario, adding lifelines (architectural elements), systemFrames and/or messages, as needed. This strategy provides the necessary traceability because it guarantees that every crosscutting relationship is considered and explicitly integrated in the scenarios.

For instance, taking the Teachmover example, it can be observed in the Figure 6-15, the REQ.1 "RDCU allows tool opened" has associated the operationalization OPE.2 "Operational opening by Teachmover Control accessing RUC-SUC". Associated to this operationalization, the following scenario has been described where the architectural elements and systemFrames are identified.

This scenario is used to describe the opening of the tool by sending the appropriate parameters. It can be observed that two systemFrames were described, following the prescription established by the ACROSET DSSA (section 7.4.1). "ToolRUC" plays the role of the *RUC* layer, that is, the layer in charge of controlling all the joints of the Teachmover. "ToolSUC" plays the role of the *SUC* layer, i.e., the layer that has direct access to the *Hardware Layer*. Two components "ToolActuator" and "ToolSensor" were introduced, as the Style recommends, to control the mechanical active element "Tool". It is described in the scenario as an *environment lifeline* because it is an external element that the system-to-be must collaborate with. The set of suitability scenarios, that have been defined for the TeachMover, were described in a similar manner (see Appendix C, section C.2).

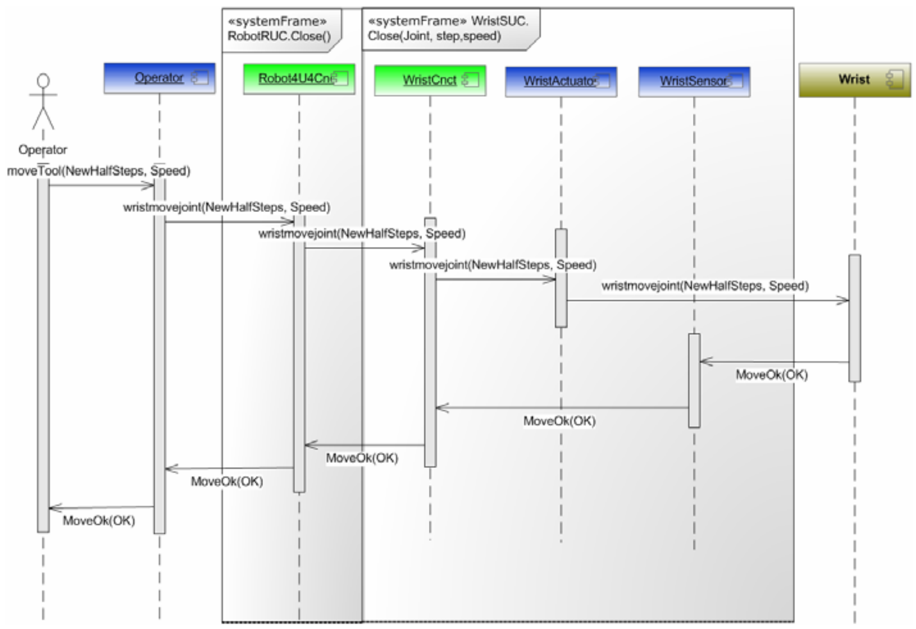

«systemFrame» RobotRUC.Close()
«systemFrame» WristSUC. Close(Joint, step,speed)
Operator
Robot4U4Cnt
WristCnct
WristActuator
WristSensor
Wrist
Operator
moveTool(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)

Figure 7-34 Describing the operationalization for requirement MoveWrist

It can be appreciated that describing the scenario as a functional scene, where every interacting shallow system along with its shallow architectural elements is identified and specified, provides a better comprehension of how the system-to-be is structured, and how every element collaborates to provide the expected behaviour. If a different scenario were described for every shallow system, the analyst would not have an overall idea of the behaviour of the system-to-be.

During the specification of the Safety concern, a crosscutting relationship was established between the REQ.1 and the safety requirement "REQ.85 Safe[REQ.1]". The analysis of this Safety requirement determined R2B as the RRC for this requirement. For this reason, it was introduced as an operationalization of "REQ.85" the use of the pattern described in the previous section, that is, the introduction of a Safety Aspect to check in advance whether the movement is going to be safe. Figure 7-35 shows how the scenario was modified to introduce the Safety Aspect. The set of safety scenarios described by the Teachmover were described in a similar way (see Appendix C, section C.2).

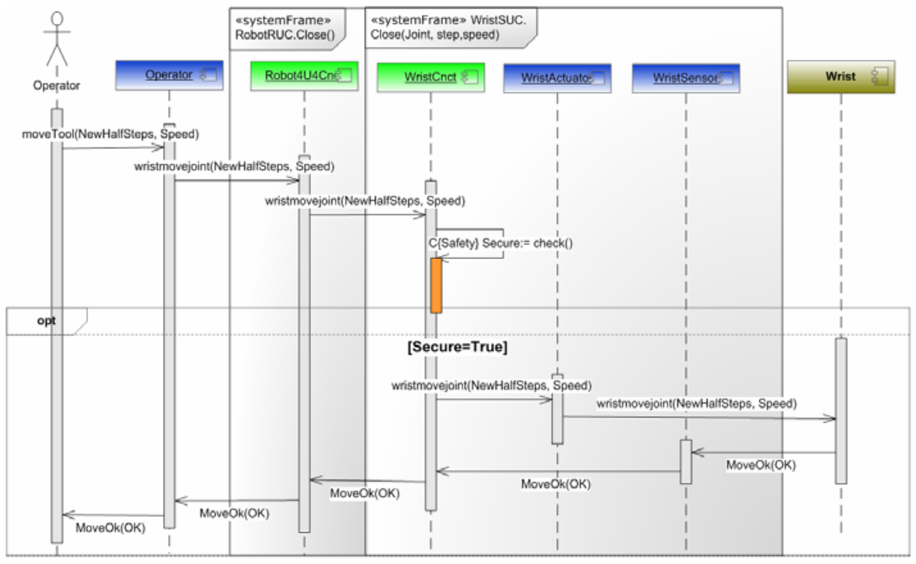

«systemFrame» RobotRUC.Close()
«systemFrame» WristSUC. Close(Joint, step,speed)
Operator
Operator
Robot4U4Cnt
WristCnct
WristActuator
WristSensor
Wrist
moveTool(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
C{Safety} Secure:= check()
opt
[Secure=True]
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)

Figure 7-35 Applying a Safety Pattern

Once the set of scenarios has been described, the next activity of ATRIUM, *Generate proto-architecture* (chapter 8), can proceed. It must be taken into account that a whole description of the set of scenario is not necessary to carry out the next activity, but a partial description is enough to obtain a draft of the proto-architecture. In this manner, the analyst can obtain a view of the proto-architecture and perform the evaluation he/she considers more appropriate. This is thanks to the automation supported in the next activity, facilitating it can be performed without additional effort.

7.6 CONCLUSIONS

In this chapter, another model has been included in the definition of ATRIUM: ATRIUM Scenario Model. The main idea behind its exploitation is to facilitate the analyst a mechanism to study and analyze the main behaviour of the system-to-be. For this reason, each ATRIUM scenario is going to depict a partial view of the system-to be coping with an operationalization decision. This means that each scenario is traced from an operationalization (see section 5.3.1) and, thus, from a set of specific requirements. This facilitates the maintenance of the traceability throughout the lifecycle.

In order to provide support for the definition of the ATRIUM Scenario Model, its main elements have been identified and described. These architectural and environmental elements interact according to a specific choreography. Regarding their description, it should be emphasized that in our proposal they are specified at enough detail level as to be used for the proto-architecture generation by synthesising the behaviour from the set of scenarios. In order to facilitate their description, a graphical notation has been specified by defining a Profile based on the UML 2.0 Sequence Diagrams.

In addition, it is worth noting that the operationalization decisions consider both functional and non-functional requirement along with their identified crosscutting. For this reason, the notation had to be adapted to manage properly this constraint. In this sense, the Aspect-Oriented approach has meant an improved advantage. A notation for the enrichment of scenarios has been defined, using this technique, has been introduced. This alternative allows the analyst to introduce lightweight solutions for specific problems, as was presented for the Safety concern in section 7.4.2. It should be also highlighted that the proposed alternative does not overload the expressiveness of the ATRIUM Scenario because it does not describe Aspects by means of Lifelines but by means of annotations of the messages. Otherwise, elements with different granularity levels would have the same representation and it could lead to confusion and misunderstandings. In addition, it provides a systematic way of dealing with early aspects and their traceability to Software Architecture.

Another advantage of the proposed ATRIUM Scenarios is that they provide a transversal view of the system-to-be. This allows the analyst to obtain the overall idea of the behaviour of the system-to-be and its structure. In this sense, the introduction of Architectural Styles, especially DSSAs, are very useful because they give the analyst some templates of how to define the scenario because they describe elements to use, allowed interactions, etc.

A process for the description of the ATRIUM Scenarios has also been described. It is worthy of note that, in addition to the Architectural Elements, the Design Patterns are another inputs. They have been recognized as a valuable asset for the description of quality solutions. Some Safety patterns, which have been detected and used during the project, have been presented for explanation purposes. In this way, how these patterns could be described in a catalogue, and how they could be exploited has been presented.

The work related to the definition of ATRIUM Scenarios model and the study of the Architectural Styles has been presented in the following publications:

– E. Navarro, P. Letelier, I. Ramos, “Requirements and Scenarios: playing Aspect Oriented Software Architectures”, Proceedings Sixth Working

IEEE/IFIP Conference on Software Architecture (WICSA 2007), Mumbai, India, January 6 - 9 2007 (short paper).

- E. Navarro, P. Sánchez, P. Letelier, J.A. Pastor, I. Ramos, “A Goal-Oriented Approach for Safety Requirements Specification”, 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06), Postdam, Germany, March 27th-30th, 2006, pp. 319-326.
- J. Jaén, J. H. Canos, E. Navarro, “A Web-Based Coordination Infrastructure for Grid Collective Services”, 5th International Conference on Web-Age Information Management (WAIM 2004), Dalian, China, July15 - 17, 2004, Proceedings in Lecture Notes in Computer Science 3129 Springer 2004, ISBN 3-540-21044-X, pp. 449-458.
- J. Jaén, E. Navarro, "An Infrastructure to Build Secure Shared Grid Spaces", VI International Conference on Coordination Models and Languages (COORDINATION 2004), Pisa, Italy, February 24-27, 2004, Proceedings in Lecture Notes in Computer Science 2949 Springer 2004, ISBN 3-540-21044-X, pp. 170-182.

"The whole history of science has been the gradual realization that events do not happen in an arbitrary manner, but that they reflect a certain underlying order, which may or may not be divinely inspired." — *Stephen Hawking*

CHAPTER 8

Towards a first view of the Architecture

8.1 INTRODUCTION

One of the main outcomes of ATRIUM is the description of a proto-architecture using the set of scenarios description as input of the process. This is the aim of the *Synthesize and Transform* activity of ATRIUM. In order to provide a proper solution to carry out this activity, we have to take into account several facts:

– The solution to be obtained has to support the whole set of the defined scenarios. The definition of scenarios, performed by the previous activity, provides us with traceability to the set of requirements to be met by the system-to-be. Thus, if a partial set of scenarios were used to obtain the proto-architecture then only a partial fulfilment of the requirements would be achieved.

– The use of Architectural Style has an high impact throughout the final description of the architecture. It is inherently different from a design pattern because it is going to affect across the whole definition of the architecture. This point is highly relevant because the Architectural Style selection, performed during the scenario description process, cannot be just an initial help to browse in the catalogue of patterns but it must be considered and used to obtain the proto-architecture. Thus, the process to be defined must employ the Architectural Style as one of its inputs.

– Considering that ATRIUM aims at analyzing alternatives to describe the Software Architecture of the system-to-be, is it possible to identify any mechanism that allows the analyst to evaluate how appropriate is the proto-architecture being obtained? Could this mechanism be automated to obtain several alternatives along with their related evaluation?

These were the challenges that were considered when a solution to describe the *Synthesize and Transform* activity of ATRIUM was established. In the following section, it will be shown how they were addressed.

8.2 CONTEXT AND ALTERNATIVES FOR OUR PROPOSAL

Before thinking of a proposal to the *Synthesize and Transform* activity of ATRIUM, it must be taken into account that ATRIUM has been defined as a Model-Driven Architecture (MDA) proposal according to the two (Mellor et al., 2004) considerations. First, ATRIUM clearly specifies its objectives, the set of artifacts to be produced (Goal Model, Scenario Model and PRISMA model), along with guidelines and techniques to build them. In addition, ATRIUM is supported by a CASE tool called MORPHEUS, presented in chapter 9, which provides high levels of automation for different tasks and activities. Figure 8-1 shows how ATRIUM can be viewed from this point of view. In this context, a remaining question to be solved is: how a PRISMA model can be generated from the described scenarios?

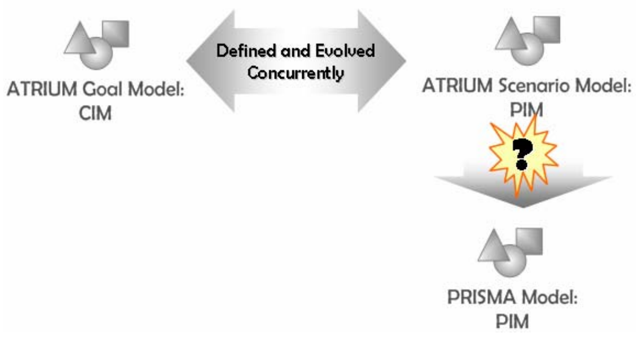


Figure 8-1 ATRIUM following the MDD approach: where a solution has to be described

In this context, transformation languages have been described as a solution to increase the productivity, improve traceability relationships between models, improve the quality because patterns can be specified as transformations, improve maintainability because traceability throughout the lifecycle is consistently described, etc. Several surveys, such as (Gerber et al., 2002), (Sendall & Kozaczynski, 2003) (Czarnecki & Helsen, 2006), identify the requirements that a transformation language model-to-model has to satisfy in order to realize the MDA initiative. However, as far as we know, it is the latter the one that has a more detailed study and has been more widely accepted. This is why we have used this work to help us in the selection of a solution for the *Synthesize and Transform* activity of ATRIUM.

(Czarnecki & Helsen, 2006) have presented a bi-dimensional taxonomy of approaches and proposals for model transformation in the MDA context. In the first dimension, their work describes which features to support are desirable for a transformation proposal. In the second dimension, they identify which approaches are currently used. We have exploited this taxonomy according to the ATRIUM specific needs. The first step we have performed has been to select which features are necessary according to our needs. In the following, it is described why these features are necessary in the context of ATRIUM:

– *Transformation rules*. The language to be used must provide some ability of transformation either by means of rewriting rules or functions of transformation.

– *Rule application control*. It is necessary to provide the analyst with mechanisms to select which instances of the model are selectable to apply the rules. This is highly relevant because, as will be described in the following section, there are transformations that have to be applied if some conditions are satisfied.

– *Rule organization*. The facilities to reuse the transformations are also desirable for our purpose because they decrease the definition efforts and enhance the maintainability of the transformations.

– *Source-target relationship*. The ability to describe if more than a model can be used as target model is decidedly pertinent. We must take into account that one of the challenges to be faced is to provide mechanism that help in the evaluation of the generated proto-architecture. For this reason, it would be desirable that not only an instantiated PRISMA model was generated during the transformation but also a model that specifies if there are faults in the specification. It would facilitate the analysis of the resulting architecture.

– *Incrementality*. Since changes in the source model can arise due to changes in the requirements, it is necessary to provide the ability to update existing target models based on those changes.

– *Directionality and Tracing.* If the transformation could be executed in both directions and information of the transformation could be recorded, it would be a first step towards the traceability top-down and bottom-up between the Scenario Model and the PRISMA model.

(Czarnecki & Helsen, 2006) have identified eight approaches to model-to-model transformation. In the following, a brief description of those approaches (reader is referred to that work for more details) along with the language we have selected in each category, that meets more of the established features. Another requirement that was used to make such a selection was the existence of a supporting environment for the language. The current approaches are:

– *Direct-manipulation approaches.* They are usually object-oriented frameworks that provide facilities for model representation and its manipulation. The main problem is that they are not oriented to model transformation so that most of the tasks have to be developed from scratch with any support of the framework. (JAMDA, 2006) is a framework following this approach. It takes as input an UML model that is added with more classes to generate code.

– *Structure driven approach.* This kind of languages is oriented to generate the target model into two phases: first, its structure, and second, its attributes and references. (OptimalJ, 2005) is a framework that provides an environment for UML modelling where business models can be described and next transformed into the structure of a working application.

– *Relational approaches.* This category refers to declaratives languages that express the transformations usually by means of constraints. (QVT, 2005) is the most popular proposals following the approach. It proposes a bi-directional framework whose specifications are non-executable.

– *Graph-Transformation based approaches.* They are based on the theory of graph transformations, where the left and right side of each transformation rule is described by means of a graph. According to (Mens et al., 2005) evaluation of graph transformation technology, GReAT (Sprinkle et al., 2003) is the proposal with more facilities for model transformation such as vertical transformations to generate more than a model, control structure and OCL to describe restrictions on the transformation, etc.

– *Template-based approach.* This approach proposes to compute the transformation by using an annotated syntax of the target model to address how the instantiation of the template is performed. Although most of the proposals, such as MoMoT (Schippers et al., 2004), are more oriented to code generation, there are some proposals as that given by (Czarnecki &

Antkiewicz, 2005), that provide the analyst with facilities for model transformation.

- *Operational approach*. This approach is quite similar to the structure driven approach because it follows an imperative proposal to describe the transformation. However, they are more oriented to model transformation. (QVT, 2005), by means of the operational mappings, provides an imperative language that uni-directionally describes transformations.
- *Hybrid approach*. There are proposals that combine some of the previous approaches as, for instance, (QVT, 2005). It describes a proposal exploiting both operational and relational transformations.
- *Other approaches*. There are other approaches as Extensible Stylesheet Language Transformation (XSLT, 1999) and meta-programming for model transformation (Tratt, 2006).

Table 8-1 shows the evaluation that we have performed using two dimensions: requirements and approaches. As can be observed, the relational approach is the one that has more facilities for our purposes, unlike the other approaches, because:

- It facilitates that we can generate an error model during the transformation, if it was required, by using the *source-target relationship* feature.
- It permits that a target model can be updated after an initial execution has been performed, i.e., the *incrementality*.
- It smoothes the progress of defining traceability top-down and bottom-up between the Scenario Model and the PRISMA model by means of the *directionality* and *traceability* features.

All the previous considerations have led us to its selection to describe the last activity of ATRIUM, concretely, by using MOF 2.0 Query/View/Transformation (QVT, 2005). This language along with its capabilities and the decisions that were made regarding its use are described in the following section.

Table 8-1 A framework for selecting the most proper approach for ATRIUM models transformation

	Transformation rules	**Rule application control**	**Rule organization**	**Source-target relationship**	**Incrementality**	**Directionality**	**Tracing**
Direct-manipuation approach	✓	✓	✓				
Structure driven approach	✓	✓					
Relational approach	✓	✓	✓	✓	✓	✓	✓
Graph-Transformation based approach	✓	✓	✓				
Template-based approach	✓	✓	✓	✓			
Operational approach	✓	✓	✓	✓		✓	
Hybrid approach	✓	✓	✓				
Other approach	✓	✓	✓				

8.2.1 QVT: a proposal for model transformation in ATRIUM

QVT has been defined as a proposal to the request launched by the Object Management Group (OMG) to describe a standard transformation language. This transformation language comes out to solve a recurrent requirement in MDA, that is, to define deterministic mapping and transformation mechanisms between elements belonging to different meta-models.

QVT is intended to provide mainly three competences for model transformation: Query, View and Transformation. Their meanings along with their specific use in ATRIUM are described in the following:

– `Query`: A query is an expression that is evaluated over a model. The result of a query is one or more instances of types defined in the source model, or defined by the query language. As described in the following section, transformations cannot be described generically but elements to be transformed have to be selected according to some specific characteristics they have. For instance, section 8.1 describes that the shallow-components interacting with humans, have to be selected to generate PRISMA components having an interaction aspect in its definition.

– `View`: A view is a model that is completely derived from another model in such a way that changes to the base model cause corresponding changes to the view. These Views are generated via transformations. This is mandatory for our purposes because our last goal is to have a view of the architecture generated from the Scenario Model.

– `Transformation` A transformation generates a target model from a source model. Depending on how transformations were described, they may cause independent or dependent models. If models are independent, it means that the relationship between them is not maintained once the target model has been generated. If models are dependent, the source and target models are coupled, so that every change on any of them affects the other. The second case is highly relevant for our purpose since it could be a first step to maintain the traceability, both top-down and bottom-up, between the Scenario Model and the Software Architecture.

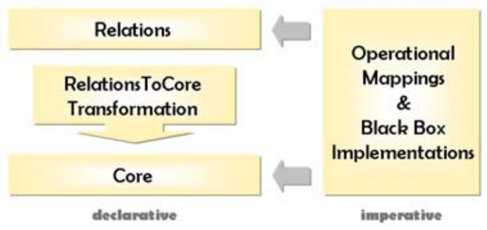

Relations
RelationsToCore Transformation
Core
Operational Mappings & Black Box Implementations
declarative
imperative

Figure 8-2 Describing QVT

QVT has been defined by means of three languages (Figure 8-2):

– `Relations` is a declarative language. One of its main characteristics is its ability to describe `object template expressions`. A template expression is used to describe patterns of domain variables whose matching is the result of binding elements from typed model to variable declared in such a domain. A relation can define several object template expressions to match patterns in the candidate models. It provides facilities for the creation and deletion of objects in order to generate those necessary elements in the target model if they do not exist. The trace management is automatically performed because a trace class is derived from each Relation. It is annotated with as many properties as domains are used in the relation. In addition, a graphical syntax has been introduced to facilitate the legibility of the transformations.

– `Core` is a declarative language that, unlike Relations, does not allow one to describe object patterns. It limits the expressiveness of the proposal because expressions that are more verbose have to be introduced to describe the same transformation. It is based on EMOF and OCL, in such a way that transformations and trace information are introduced as a MOF metamodel.

– `Operational Mappings` is an imperative Domain Specific Language that has OCL as query language. OCL has been extended with imperative features to provide the necessary capabilities to perform the transformations. As described in Table 8-1, this language does not provide support for features as needed as directionality and tracing.

Taking into account the characteristics supported by the three languages, the Relations language was eventually selected. This is because it is more expressive and user-friendly than the core language and it has more capabilities than those supported by the Operational Mapping. For instance, Operational Mapping does not provide support for directionality and tracing that does provide Relations.

8.3 DESCRIBING QVT TRANSFORMATIONS

Using the Scenarios Model specification, the generation of the PRISMA proto-architecture is carried out. It has to be emphasized that the Scenarios Model provides us with partial views of the architecture, where only shallow-components, shallow-connectors and shallow-systems have been identified along with their behaviour expressed through interaction. They are called shallow because we do not need their complete definition but an initial one that can be refined for their later compilation to code if needed. Therefore, before proceeding to such a compilation a synthesis process must be performed to create these shallow-architectural elements from their collaboration throughout the set of scenarios.

In order to describe the needed transformations QVT Relations have been used, as was presented previously. The set of transformation rules has been catalogued as `architectural transformation patterns`, `Architectural Style-oriented transformations` and `idioms-oriented transformations`. The first are used to describe those transformations applicable to most of the existing ADLs because they are focused on the generation of components, connectors and systems. The second are defined to take into account the Architectural Style to be satisfied by the proto-architecture to be generated. The third are following the (Schmidt et al., 2000) definition:

"An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language."

For this reason, those transformation that are oriented to generate a PRISMA specification will be described as idioms. However, if the transformation is applicable to any architectural model, it will be considered a generation pattern. This distinction will help in the process of generating any necessary architectural model only by means of the description of its specific idioms. With this aim, and thanks to the *Rules Organization* feature of QVT, several transformations have been defined that can be imported and used depending on the expected outcome of the generation process. For instance, a transformation has been described for architectural transformation patterns, other for the PRISMA idioms and another for the Architectural Style. It facilitates that using the same set of scenarios, different proto-architectures could be generated depending on the selected Architectural Model and its specific idioms. Table 8-2 describes how the transformation has been declared specifying two typed models to be transformed: *scenarios* that represents a model that conforms the ATRIUM Scenarios meta-model (described in chapter 7) and *archModel* that represents a model that conforms the Architectural meta-model. Specifically, the PRISMA Metamodel (described in chapter 4) has been used for executing the transformations. We have described the transformations using the ArchitecturalModel domain thinking of their use with other architectural models.

Table 8-2 Declaration of the transformation

```
transformation ScenariosToArchModel (scenarios: ATRIUMScenarios,
archModel: ArchitecturalModel)
```

Every transformation is defined by means of a set of relations that must hold if the transformation is successfully applied. Every relation declares several domains. For instance, the relation "FragmentToSystems" (Table 8-3) specifies the *scenarios* and the *archModel* domain, which where previously specified in the transformation (Table 8-2). The relation specifies how the elements belonging to one domain are matched to the elements belonging to the other domains. In the relation, FragmentToSystems a matching is performed between the elements belonging to the *scenarios* domain and those belonging to the *archModel* domain. For this reason, every relation specifies in each domain a pattern that must be satisfied by every element that is going to be used for the matching. In the examples described in the following sections, a pattern is described in the *scenarios* domain and another one in the *archModel* domain. In the relation "FragmentToSystems", the *scenarios* domain describes a pattern consisting of a "systemFrame" with an attribute "fragment" that must refer to a "systemFrame" having an attribute "systemName". It can be observed that an OCL expression can also be defined to filter out those elements of the Scenarios model that do not satisfy a condition. In the *archModel* domain, the pattern describes a "System" that has an attribute "name". It can be observed that the matching between the elements belonging to the each domain is established by means of the variable "cn", because both "systemName" and "name" are bound to this variable.

When a relation is described, additionally to the domains, two clauses more can be included that define OCL expressions or other relations: *when* and *where.* The former specifies the conditions that must be hold to the relation be satisfied, i.e., the matching established by the relation will not be performed if the *when* clause is not hold without reporting any inconsistency. The latter specifies the conditions that must be hold by all the elements participating in the relation. For instance, the relation "FragmentToSystems" describes that three relations more must be satisfied.

Thanks to the *Directionality* feature of QVT, a transformation can be executed for enforcement selecting the target domain. This means that selecting *archModel* as target model the proto-architecture is generated. The execution of the transformation checks, whether there are elements in the target model that satisfy the relations and if that was not the case then the elements will be created, deleted or modified in the target model to enforce the relations. With this aim, each pattern can be evaluated using two different modes: `checkonly` that only checks if the pattern is not satisfied reporting an inconsistency in this case; and, `enforce` that checks whether the pattern is satisfied and modifies, deletes or creates elements, as necessary, in the target model if that was not the case. This allows the analyst to both generate the proto-architecture and check if inconsistencies between the generated proto-architecture and the scenario model emerge.

It is worthy of note that it is not necessary to provide as input a full set of scenarios required to describe the system behaviour. On the contrary, with only one scenario the generation can proceed. However, the *Incrementality* feature of QVT facilitates that as new scenarios are defined or modified, the proto-architecture can be automatically updated.

In the following sections, the most relevant relations are described along with a graphical example to improve its comprehension. Their full description is presented in the Appendix C. Before presenting them, it must be highlighted that every ATRIUM Scenario is defined as an *Interaction* enclosing one or several *SystemFrames*.

8.3.1 Architectural transformation patterns

In order to describe the transformation between the two models, the first relation to be described is that oriented to the creation of Systems in the architectural model, called "FragmentToSystems". For this reason, it has been declared as a top relation. This relation is checked for every SystemFrame existing in the ATRIUM Scenarios Model. If it fails, that is, there is no a system whose "name" bounds to the same variable "cn" that is bounded by "systemName" of the SystemFrame, then a new System is created. In the Figure 8-3, it can be observed that a new System called "WristSuc" will be created because a SystemFrame called "WristSuc" exists. It must be taken into account that "System.name" has been defined as an identifying property to avoid creating duplicated Systems.

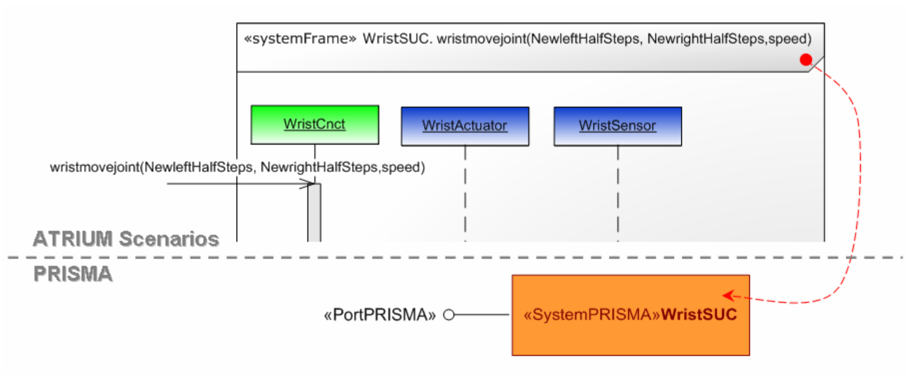

«systemFrame» WristSUC. wristmovejoint(NewleftHalfSteps, NewrightHalfSteps,speed)
WristCnct
WristActuator
WristSensor
wristmovejoint(NewleftHalfSteps, NewrightHalfSteps,speed)
ATRIUM Scenarios
PRISMA
«PortPRISMA»
«SystemPRISMA»WristSUC

Figure 8-3 Establishing mappings between an ATRIUM Scenarios systemFrame and a System

Table 8-3 Describing the transformation from an ATRIUM systemFrame to a System

```
top relation FragmentToSystems
{
   cn: String;
   c: ArchitecturaModel::Component;

   checkonly domain scenarios p:SystemFrame{
         fragment=sf:SystemFrame {systemName=cn}

   }{p.fragment->notEmpty()};

   enforce domain archModel s:System
   {
      name=cn
   };
   where{
      MessageToArchElements(sf, s);
      MessageBetweenComponentsToArchElements(sf, s);
      GeneralOrderingToWeaving(sf, s);
   }
}
```

The relation “FragmentToSystems” is constrained by the *where* clause specifying that whenever it holds, other three relations must hold. One of these relations is “MessageToArchElements” (see Table 8-4) that specifies a pattern for the scenarios domain that retrieves all the objects of type Message connecting two named Lifelines. It is constrained by an OCL expression to filter out those Messages that are not connecting a Component Lifeline and a Connector Lifeline. The template expression valuates “l1” and “l2” with the Lifelines that act as Sender and Receiver respectively on each Message. Then, the *where* clause determines which of these Lifelines, “l1” and “l2”, is a Component (“NameComponent”) and which is a Connector (“NameConnector”). Once this is done, a Component and a Connector are created by evaluating the relations "LifelineComponentToComponent" and "LifelineConnectorToConnector" respectively. Thus, the "MessageToArchElements" relation enforces the creation of Components and Connectors. The template expression associated to the *archModel* domain determines that both domain variables, "comp" and "con", are bound to the properties "containsComp" and "containsCnct". This determines that both architectural elements will be referred by the system "s".

For instance, Figure 8-4 depicts how the message "wristmovejoint" connecting two lifelines "wristCnct" and "wristActuator" and belonging to the systemFrame "WristSUC" will be transformed in the architectural model into two components with the same name and referred by the System "WristSUC". The *where* clause also evaluates other relations that must hold. "MessageToAttachmentComponent" and "MessageToAttachmentConnector" are described below. "LifelineToServiceAspectComponent" and "LifelineToServiceAspectConnector" are detailed in section 8.1.

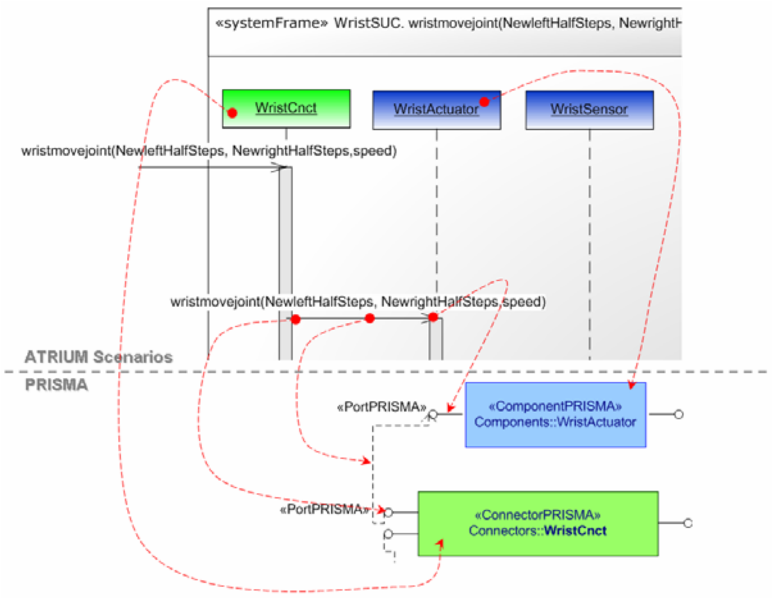

«systemFrame» WristSUC. wristmovejoint(NewleftHalfSteps, Newright
WristCnct
WristActuator
WristSensor
wristmovejoint(NewleftHalfSteps, NewrightHalfSteps,speed)
wristmovejoint(NewleftHalfSteps, NewrightHalfSteps,speed)
ATRIUM Scenarios
PRISMA
«PortPRISMA»
«ComponentPRISMA» Components::WristActuator
«PortPRISMA»
«ConnectorPRISMA» Connectors::WristCnct

Figure 8-4 Establishing mappings between ATRIUM Scenarios Messages and Architectural Elements

Table 8-4 Describing the transformation from an ATRIUM Message to an architectural element

```
relation MessageToArchElements
{
   cn: String; cn1: String; cn2: String;
   comp:ARCHMODEL::Component;
   con:ARCHMODEL::Connector;
   lcon:ATRIUMScenarios::Lifeline;
   lcom:ATRIUMScenarios::Lifeline;

   checkonly domain scenarios sf:SystemFrame{
      message=m:Message
      {
         name=cn,
         sendEvent=m1:MessageOccurrenceSpecification{
            covered=l1:Lifeline{name=cn1}
         },
         receiveEvent=m2:MessageOccurrenceSpecification{
            covered=l2:Lifeline{name=cn2}}
         }
      }{(l1->oclIsKindOf(Connector)
               and l2->oclIsKindOf(Component))
               or (l1->oclIsKindOf(Component)
               and l2->oclIsKindOf(Connector))};

   enforce domain archModel s:System
   {
      containsComps= comp ,
      containsCnct= con
   };
   where
   {
      lcom=NameComponent(l1, l2);//which is the Component Lifeline
      lcon=NameConnector(l1, l2); //which is the Connector Lifeline
      LifelineComponentToComponent(lcom, comp);
      LifelineConnectorToConnector(lcon, con);
      MessageToAttachmentComponent(lcom, s, lcon.name, lcom.name);
      MessageToAttachmentConnector(lcon, s, lcon.name, lcom.name);
      LifelineToServiceAspectComponent(lcom, comp);
      LifelineToServiceAspectConnector(lcon, con);
   }
}

relation LifelineComponentToComponent
{
   cn: String;

   checkonly domain scenarios l:Lifeline{
      name=cn
   }{l->oclIsKindOf(Component)};

   enforce domain archModel con :Component{
      name=cn
   };
}
```

The relation "MessageToAttachmentComponent” has been defined to generate Attachments, i.e., the connections between Components and Connectors. In

Figure 8-4, it can be observed that the message “wristmovejoint” is mapped to a connection between “wristCnct” and “wristActuator”. An Attachment is defined by means of the connection between two ports belonging to two different architectural elements. In addition, the Attachments are defined in the context of Systems. Therefore, the relation will pass the evaluation if both the “name” property of the Lifeline and the “name” property of the Component are bound to the same domain variable “cn”. Considering this matching, an attachment will be enforced that has a name property bound to the domain variable “attName”. This domain variable is resolved in the *where* clause, where is set by the concatenation of the Component and the Connector name. The relation establishes the connection of the Component to the attachment. For this reason the property “linkPort” is bound to the domain variable “p” that defines the Port of the Component being attached. In a similar way, the relation “MessageToAttachmentConnector” proceeds for the connection of the Connector. It must be highlighted that an identifying property has been defined for the Attachment, as the name of the Attachment and the System where it is being defined. This facilitates that if the same Lifelines are connected by different messages there are no new attachments defined between them.

Table 8-5 Describing the transformation from an ATRIUM Message to an Attachment

```
relation MessageToAttachmentComponent
{

   cn:String;
   nComp:String;
   attName:String;

   checkonly domain scenarios l:Lifeline{name=cn};

   enforce domain archModel s:System{
      connect=a:Attachment{
         name=attName,
         linkPort=p:Port{
            name=nComp,
            ArchitecturalElement=c:Component{name=cn}
         }
      }
   };
   primitive domain portComp:String;
   primitive domain portCon:String;
   where{
      attName=portComp+portCon;
      nComp=portComp;
   }
}
```

8.3.2 Applying the Architectural Style

The selected Architectural Style, which is applied in the process of the scenarios specification, is a deciding factor for the configuration of the proto-architecture. For this reason, transformation rules must be described that exploit this decision for its generation. With this aim, these rules use Bindings and Attachment because they determine the compositionality of the system-to-be, i.e., which architectural elements are composed by others or which architectural elements are connected to others.

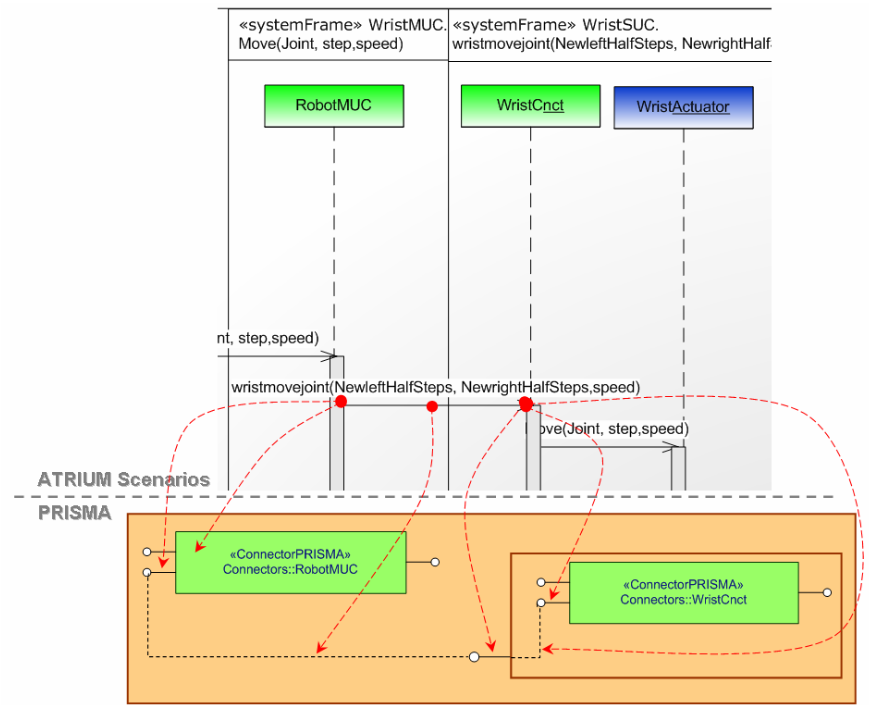


Figure 8-5 Establishing mapping between ATRIUM Scenarios Interaction connecting Connectors in different Systems and Bindings

An example in this sense is the ACROSET, a layered style that clearly describes an assignment of responsibilities and a configuration. In this style, the System MUC is composed of several SUC Systems and, similarly, the RUC System is composed of several MUC Systems. This means that attachments must be established between the architectural elements owned by the composed System and the component System. Figure 8-5 illustrates this situation. Two connectors, “RobotMUC” and “WristCnct”, have a message “wristmovejoint” connecting them. Both Connector Lifelines belongs to different SystemFrames. The mapping to the architectural model determines that:

– a Port is created for "WristCnct" to facilitate the communication; this port is connected, by means of a Binding, to the Port of the System "WristSUC".

– an Attachment must be established between the port of "WristSUC" and the port of "RobotMUC" because "WristSUC" is contained in the definition of the System "WristMUC".

In order to apply these mappings, several relations have been defined, as can be observed in the following. The first relation is "Transformation ApplyingACROSET" described as a top relation because it is in charge of retrieving all the messages that are going to be used for the application of the ACROSET Style. These ATRIUM Scenarios Messages do not belong to any of the interacting SystemFrames but to the Interaction that encloses them. For this reason, the template expression in the scenarios domain retrieves those messages "m" defined in the Interaction "i". This template expression also obtains the Lifelines ("l1" and "l2") connected by means of these messages along with the SystemFrames ("i1" and "i2") they belong to. It was described in the previous chapter that a SystemFrame has an attribute called "role" that determines its played role regarding the employed Architectural Style. This attribute is used by the OCL expression, in the scenarios domain, to make sure that the relation is only applied between MUCs and SUCs or RUCs and MUCs.

In the *archModel* domain, the template expression determines that the domain variable "s" of type System, will bind its property "name" to the name of the Architectural Element having the port "ps1". This variable will have a binding because of the *where* clause expression "LifelineToArchitecturalElementBinding", that is described below. In addition, this template also describes that the property "name" is bound to the domain variable "att" of kind Attachment. This domain variable has a property "name" which is bound to "attName" that is evaluated in the *where* clause by the concatenation of the Systems and Lifelines names. It avoids the creation of different Bindings connecting the same Ports because it has been defined as an identifying property. The property "linkPort" is also bound to the variable "ps1", resolved in the *where* clause by "LifelineToArchitecturalElementBinding". In the example of the Figure 8-5, this means the connection of the Port owned by "SystemMUC".

Table 8-6 Describing the transformation for dealing with the ACROSET

```
top relation TransformationApplyingACROSET
{
   cn: String;
   ln1: String;
   sn1: String;
   ln2: String;
   sn2: String;
   attName:String;
   s1: ARCHMODEL::System;
   s2: ARCHMODEL::System;
   ps1: ARCHMODEL::Port;

   checkonly domain scenarios i:Interaction
   {
      message= m:Message{
      name=cn,
      sendEvent=m1:MessageOccurrenceSpecification{
         covered=l1:Lifeline{
            name=ln1,
            interaction=i1:SystemFrame{
               systemName=sn1
            }
         }
      },
      receiveEvent=m2:MessageOccurrenceSpecification{
         covered=l2:Lifeline{
            name=ln2,
            interaction=i2:SystemFrame{systemName=sn2}
         }
      }
   }{i.message->notEmpty() and
   ((i1.role='MUC' and i2.role ='SUC') or
      (i1.role ='RUC' and i2.role='MUC'))};



   enforce domain archModel s:System{
      name=ps1.ArchitecturalElement.name,
      connect=att:Attachment{
         name=attName,
         linkPort=ps1
      }
   };
   where{
      attName=sn1+ln1+sn2+ln2;
      LifelineToArchitecturalElementBinding(l1, ps1, sn2+ln2);
      MessageToBinding(l2, s, sn1+ln1, sn2+ln2);
   }
}
```

The relation “MessageToBinding” establishes, in the *archModel* domain, a template expression that specifies a property “connect” for the domain variable “s”. This is bound to a domain variable “att” of kind Attachment to describe an Attachment that is internal to the System. Its property "linkPort" is bound to "ps2" that is resolved in the *where* clause by the relation "LifelineToArchitecturalElementBinding". The property "name" of the Attachment filters the proper Attachment because "attName" is a variable bound by the evaluation performed in the *where* clause. This means the establishment of the attachment of the Port owned by "RobotCnct" in the example of the Figure 8-5.

Table 8-7 Transforming a Message to a Binding

```
relation MessageToBinding
{
   attName:String;
   ps2: ARCHMODEL::Port;

   checkonly domain scenarios l2:Lifeline{};


   enforce domain archModel s:System{
      connect=att:Attachment{
         name=attName,
         linkPort=ps2
      }

   };
   primitive domain portName1:String;
   primitive domain portName2:String;
   where{

       attName= portName1 + portName2;
       LifelineToArchitecturalElementBinding(l2, ps2, portName1);
   }
}
```

The relation "LifelineToArchitecturalElementBinding", because of its application to the example of the Figure 8-5, determines the "WristSUC" has a Port employed for the connection to the same attachment as "RobotCnct". With this aim, the property "name" of the Port is bound to a variable "pn" that is assigned in the *where* clause to "portName", a variable of the primitive domain of the relation. This relation will have a successful evaluation if the *where* clause is positively evaluated, that is, "LifelineToComponentBinding" or "LifelineToConnectorBinding" are positively evaluated, as described in the following.

Table 8-8 Transforming a Lifeline to an Architectural Element

```
relation LifelineToArchitecturalElementBinding
{
   cn: String;
   ln: String;
   pn:String;

   checkonly domain scenarios l:Lifeline{
      name=ln,
      interaction=i:SystemFrame{systemName=cn}
   };
   enforce domain archModel p:Port{
      name=pn,
      ArchitecturalElement=s:System{name=cn}
   };

   primitive domain portName:String;

   where
   {
      pn=portName;
      LifelineToComponentBinding(l, p, portName);
      LifelineToConnectorBinding(l, p, portName);
   }
}
```

The relation “LifelineToConnectorBinding” is in charge of creating the binding that, in the example of the Figure 8-5, connects the Port of the “WristSUC” and the port of the “WristCnct”. For this reason, the *archModel* domain defines a template with a domain variable “p”, of type Port. This is not free but bound to the port created in the relation “LifelineToArchitecturalElementBinding”. Thus, the property “name” of the Port and the domain variable “s”, of type System, are also bound. The mapping between *scenarios* domain and *archModel* domain is established for the Lifeline “l” and the Connector “c” because both have a property “name” bound to the same variable “ln”. It is also described that they are composing a SystemFrame “sf” and a System “s”, respectively, whose properties “name” are bound to the same variable “sn”. The Binding relation is established in the *archModel* domain because both the property “isComposed” of “s” and the property “isComponent” of “c” are bound to the same domain variable “b”. This variable is resolved in the *where* clause by means of the relation “LifelineToBinding” that creates a new Binding if it does not exist already with that name. The relation “LifelineToComponentBinding” proceeds in a similar way, but in this case a Binding with a Component will be created.

Table 8-9 Transforming a Lifeline to an Architectural Element

```
relation LifelineToConnectorBinding
{
   sn: String;
   pn: String;
   ln: String;
   a: archModel::ArchitecturalElement;
   b: archModel::Binding;
   checkonly domain scenarios l:Lifeline{
      name=ln,
      interaction=sf:SystemFrame{systemName=sn}
   }{l->oclIsKindOf(Connector)};

   enforce domain archModel p:Port{
      name=pn,
      ArchitecturalElement=s:System{
         name=sn,
         containsCnct=c:Connector{
            name=ln,
            has=pc1:Port{
               name=pn,
               isComponent=b
            }
         },
         has=pc2:Port{
            name=pn,
            isComposed=b
         }
      }
   };
   primitive domain portName:String;
   where
   {
      pn=portName;
      LifelineToBinding(l, b, portName+l.name);
   }
}

relation LifelineToBinding
{

   cn: String;
   checkonly domain scenarios l:Lifeline{};

enforce domain archModel b:Binding{
      name=cn
   };

   primitive domain bindingName:String;

   where{
      cn=bindingName;
   }
}
```

8.3.3 PRISMA idioms

As was introduced in section 8.3, a set of idioms has been described to deal with the mappings that are inherent to the PRISMA model. Two kinds of idioms are described. One of them is related to the constraint described in

PRISMA for the relation between architectural elements. The other kinds of idioms are related to the way the architectural elements are described by a gluing of aspects.

Idioms for Architectural Elements

Figure 8-6 shows an example of an ATRIUM Scenario where two Component Lifelines are interacting. One of the constraints of PRISMA is that two Components cannot be directly attached. For this reason in the figure, the mapping of this scenario to a PRISMA specification means that a Connector must be created. To achieve this goal, the relation "MessageBetweenComponentsToArchElements" has been defined. This relation is evaluated in the context of the relation "FragmentToSystems", that is, when a SystemFrame is being mapped to a PRISMA System.

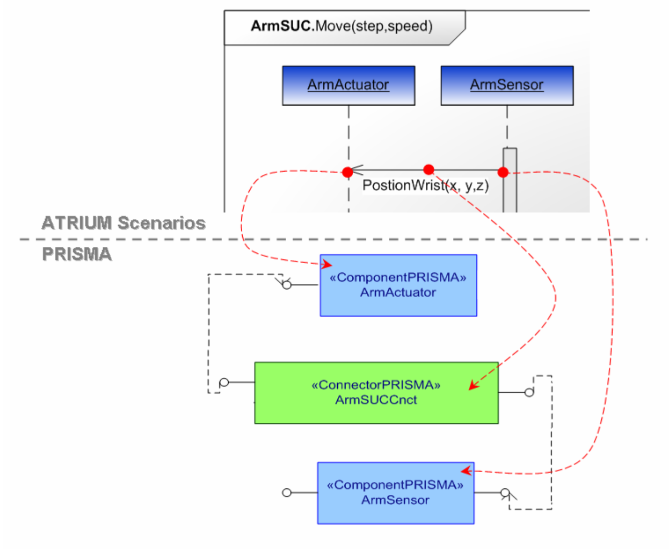


Figure 8-6 Establishing mapping between ATRIUM Scenarios Component-Component Interaction and Components and Connector

As can be observed, an OCL expression is defined to apply this relation whenever both Lifelines "l1" and "l2" are Component Lifelines. A template expression is defined in the scenarios domain that filters every Message "m", defined in the context of the SystemFrame "sf", connecting two Components Lifelines "l1" and "l2". In the *archModel* domain, a pattern describes that for the domain variable "s", of type System, its property "containsComps" and "containsCnct" will be enforced to the variables "comp" and "con" resolved in the *where* clause. The variable "comp" is resolved by the relation "LifelineComponentToComponent", explained in the previous section, to determine if a PRISMA Component exists as mapped from the Lifeline "l1". In Figure 8-6, it is checked that the PRISMA Component "ArmSensor" exists (otherwise, it will be created). The variable "con" is similarly resolved, as is represented by the "ArmSUCCnct" in Figure 8-6. The attachment between both "comp" and "con" is resolved by means of the relations "MessageToAttachmentConnectorBetweenComponents" (explained in the previous section) and "MessageToAttachmentComponent". It can be observed that the relation "MessageReceiveBetweenComponentsToArchElements" must be also positively evaluated, as described below.

Table 8-10 Transforming a Message between Component Lifelines to two Components and a Connector

```
relation MessageBetweenComponentsToArchElements
{
   cn: String;
   cn1: String;
   cn2: String;
   comp: archModel::Component;
   con: archModel::Connector;

   checkonly domain scenarios sf:SystemFrame
   {
      message=m:Message
      {
         name=cn,
         sendEvent=m1:MessageOccurrenceSpecification{
            covered=l1:Lifeline{name=cn1}},
      receiveEvent=m2:MessageOccurrenceSpecification{
            covered=l2:Lifeline{name=cn2}}
      }
   }{(l1->oclIsKindOf(Component) and l2->oclIsKindOf(Component))};

   enforce domain archModel s:System{
      containsComps= comp,
      containsCnct=con

   };
   where
   {
      LifelineComponentToComponent(l1, comp);
      LifelineConnectorToConnectorBetweenComponents(l1, con);
      MessageToAttachmentComponent(l1, s, con.name, comp.name);
      MessageToAttachmentConnectorBetweenComponents(l1, s,
               con.name, comp.name);
      MessageReceiveBetweenComponentsToArchElements(l2, s, cn1);
   }
}
```

The relation "MessageReceiveBetweenComponentsToArchElements" is in charge of enforcing the mapping, in the example of the Figure 8-6, between the Lifeline "ArmActuator" and the Component "ArmActuator" as well as its connection to the "ArmSUCCnct". With this aim, the domain scenario describes a pattern where the domain variable "l" is bound because of the relation "MessageBetweenComponentsToArchElements", i.e., it is bound to the Lifeline that is receiving the message. A template expression is defined in the *archModel* domain that specifies a domain variable "s", of kind System and bound by the relation "MessageBetweenComponentsToArchElements", in order to check (or enforce otherwise) that its properties "containsComps" and "containsCnct" are bound to "comp" and "con". Both of them are resolved in the *where* clause by means of "LifelineComponentToComponent" and "LifelineConnectorToConnectorBetweenComponents". Both "com" and "con" are attached by resolving the last two relations, as was explained for "MessageBetweenComponentsToArchElements" in the previous section.

Table 8-11 Transforming a Lifeline to a Component and a Connector

```
relation MessageReceiveBetweenComponentsToArchElements
{
   comp: archModel::Component;
   con: archModel::Connector;

   checkonly domain scenarios l:Lifeline{};
   enforce domain archModel s:System{
      containsComps= comp,
      containsCnct= con

   };
   primitive domain compName:String;
   where
   {
      LifelineComponentToComponent(l, comp);
      LifelineConnectorToConnectorBetweenComponents(l, con);
      MessageToAttachmentComponent(l, s, con.name, comp.name);
      MessageToAttachmentConnectorBetweenComponents(l, s,
            con.name,comp.name);
   }
}
```

Idioms for Coordination Aspect identification

One of the characteristics that PRISMA exhibits is the description of Components and Connectors by means of a gluing of Aspects. This not only determines how they are internally specified but also how they behave according to services specified in those aspects. For this reason, in this section some idioms are described to identify what aspects appear during the architectural synthesis process.

One of the constraints imposed by PRISMA is that every Connector must be described with a Coordination Aspect that facilitates the coordination process between the connected Components. For this reason, whenever a message is received by a Connector Lifeline, it will be mapped to a service belonging to the Coordination Aspect of a Connector. Figure 8-7 shows a typical example where the "WristCnct" Connector Lifeline sends a message "writsmovejoint" to the "WristActuator". The MessageOccurrenceSpecification, covered by "WristCnct", is mapped to a service ""wristmovejoint" in the Aspect "CoordWristCnct".

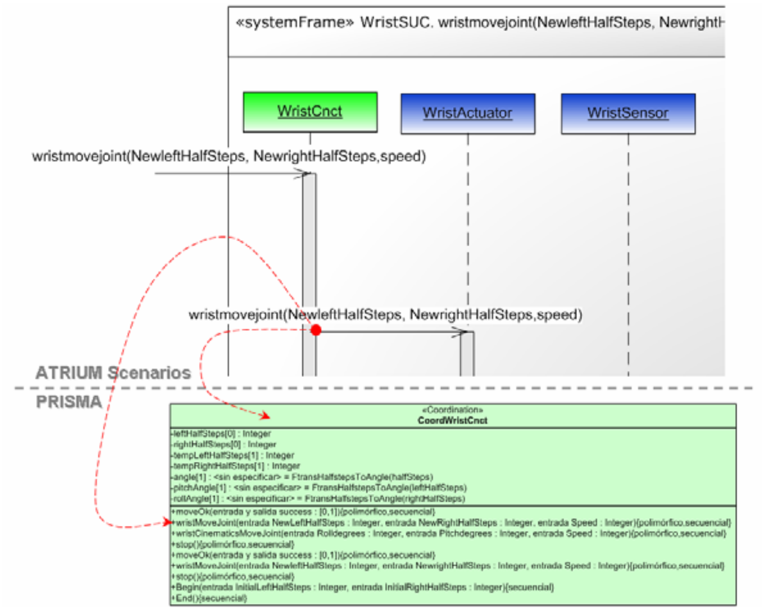

«systemFrame» WristSUC. wristmovejoint(NewleftHalfSteps, NewrightH
WristCnct
WristActuator
WristSensor
wristmovejoint(NewleftHalfSteps, NewrightHalfSteps,speed)
wristmovejoint(NewleftHalfSteps, NewrightHalfSteps,speed)
ATRIUM Scenarios
PRISMA
«Coordination»
CoordWristCnct
-leftHalfSteps[0] : Integer
-rightHalfSteps[0] : Integer
-tempLeftHalfSteps[1] : Integer
-tempRightHalfSteps[1] : Integer
-angle[1] : <sin especificar> = FtransHalfstepsToAngle(halfSteps)
-pitchAngle[1] : <sin especificar> = FtransHalfstepsToAngle(leftHalfSteps)
-rollAngle[1] : <sin especificar> = FtransHalfstepsToAngle(rightHalfSteps)
+moveOk(entrada y salida success : [0,1]){polimórfico,secuencial}
+wristMoveJoint(entrada NewLeftHalfSteps : Integer, entrada NewRightHalfSteps : Integer, entrada Speed : Integer){polimórfico,secuencial}
+wristCinematicsMoveJoint(entrada Rolldegrees : Integer, entrada Pitchdegrees : Integer, entrada Speed : Integer){polimórfico,secuencial}
+stop(){polimórfico,secuencial}
+moveOk(entrada y salida success : [0,1]){polimórfico,secuencial}
+wristMoveJoint(entrada NewleftHalfSteps : Integer, entrada NewrightHalfSteps : Integer, entrada Speed : Integer){polimórfico,secuencial}
+stop(){polimórfico,secuencial}
+Begin(entrada InitialLeftHalfSteps : Integer, entrada InitialRightHalfSteps : Integer){secuencial}
+End(){secuencial}

Figure 8-7 Establishing mapping between an ATRIUM Scenarios Message and a PRISMA Aspect

The relation "LifelineToServiceAspectConnector" is evaluated in the context of the relation "MessageToArchElements". This means that, when the relation is evaluated, "l" and "c" are not free but they are bound to objects of its respective types. In the scenarios domain, an OCL expression determines that the relation will be applied only when the type of "m" is Message, that is, the AspectualMessages are dealt with in a different way. In the *archModel* domain, the variable "c" of type Connector binds its property "imports" to "as". It is a

Coordination Aspect that is resolved in the *when* clause by the relation "LifelineToAspect". In the *where* clause, it is determined the kind of service (in or out) for its use in the relation "LifelineToService". This relation maps the message to a service of the aspect "as".

Table 8-12 Transforming a Lifeline to a Service of an Aspect

```
relation LifelineToServiceAspectConnector
{
   cn: String;
   cn1:String;
   cn2:String;
   as: archModel::Aspect;

   checkonly domain scenarios l:Lifeline{
      coveredBy=mo:MessageOccurrenceSpecification {
         message=m:Message{}
      }
   }{m->oclIsTypeOf(Message)};

   enforce domain archModel c:Connector{
      imports=as
   };
   when{
      LifelineToAspect(l,as, 'Coordination');
   }
   where{
      cn1=mo.event.name.substring(1,2);
      cn2=KindService(cn1);
      LifelineToService(m, as, cn2);
   }
}
```

The relation "LifelineToAspect", in the *archModel* domain, defines a template expression where the variable "as" has the properties "name" and "concern" bound to "an" and "conc" that are resolved in the *where* clause. Variable "conc" is assigned to the kind of Concern the Aspect is. Variable "an" is assigned to the concatenation of part of the name of the concern and the name of the Lifeline. Two relations are evaluated in the *where* clause as well. "LifelineToBegin" and "LifelineToEnd" that determine the creation of these services in the PRISMA Aspect to assure its appropriate initialization and destruction.

Table 8-13 Transforming a Lifeline to an Aspect

```
relation LifelineToAspect
{
   cn: String;
   an: String;
   conc: String;
   checkonly domain scenarios l:Lifeline{
      name=cn
   };

   enforce domain archModel a:Aspect{
      name=an,
      concern=conc
   };

   primitive domain parConcern:String;

   where{
      conc=parConcern;
      an= parConcern.substring(1,4)+ l.name;
      LifelineToBegin(l,a);
      LifelineToEnd(l,a);
   }
}

relation LifelineToBegin
{
   cn: String;

   checkonly domain scenarios l:Lifeline{};

   enforce domain archModel a:Aspect{
      belongsTo= b1:Service{name='begin()'}
   };
}
```

The relation "LifelineToService" describes a pattern in the *archModel* domain where the variable "as" of kind Aspect has a property "belongsTo" that is used to refer to the Service to be created. In addition, it is also checked (or enforced if it does not exist already) that the name of the message and the name of the service being added are equal by means of the bound to the variable "cn". The property "type" is bound to "ks", a variable resolved in the *where* clause by its assignment with "parType". This variable determines the kind of service that is being checked (or enforced).

Table 8-14 Transforming a Lifeline to a Service

```
relation LifelineToService
{
   cn: String;
   ks: String;

   checkonly domain scenarios m:Message{
        name=cn
   };

   enforce domain archModel as:Aspect{
      belongsTo=s:Service{
         type=ks,
         name=cn
      }
   };

   primitive domain parType:String;

   where{
      ks=parType;
   }
}
```

Idioms for Presentation Aspect identification

PRISMA provides support for human interaction by means of the use of Presentation Aspects. The ATRIUM Scenarios Model provides the analyst with specific notation for this aim by means of the use of Human Lifelines(7.3.1). Figure 8-8 depicts an example, where the Component Lifeline "UIRobot" receives a message "Move" from the Human Lifeline "Operator". By means of the relation "MessageFromHumanToComponent", explained below, the MessageOccurrenceSpecification, that is the end of the Message "Move" and is covered by "UIRobot", is mapped to a service "Move" owned by the Aspect "PresUIRobot".

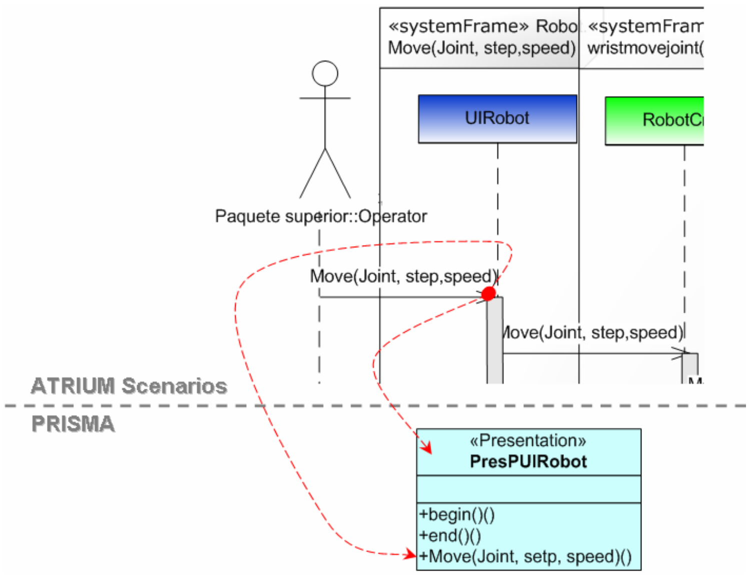


Figure 8-8 Establishing mapping between an ATRIUM Scenarios Operator-System Interaction and a PRISMA Presentation Aspect

The relation "MessageFromHumanToComponent" describes, in the scenarios domain, a template expression that filters every two Lifelines "l1" and "l2" connected by a message whenever the OCL expression is satisfied, i.e., if one of them is a Component Lifeline and the other a Human Lifeline. In the *archModel* domain, the pattern checks (or enforce if it does not exist already) that the domain variable "s" of type System will be bound to those Systems having a property "name" bound to the variable "sn". This facilitates the mapping of the SystemFrames containing the Lifeline Component with the Systems. In addition, the property "containsComps" is bound to "comp", a variable which is resolved in the *where* clause by the relation "LifelineComponentToComponent", explained in the previous sections. The *where* clause also specifies that another relation must be positively evaluated: "LifelineToServicePresentationAspectComponent", detailed below.

Table 8-15 Transforming a Messages received from Human Lifeline to a Component

```
top relation MessageFromHumanToComponent
{
   cn: String;
   ln1: String;
   sn: String;
   ln2: String;
   sn2: String;
   comp: archModel::Component;
   lcom: ATRIUMScenarios::Lifeline;

   checkonly domain scenarios p:Package
   {
      packagedElement= i:Interaction{
         message= m:Message{
            name=cn,
            sendEvent=m1:MessageOccurrenceSpecification{
               covered=l1:Lifeline{
                  name=ln1
               }
            },
            receiveEvent=m2:MessageOccurrenceSpecification{
               covered=l2:Lifeline{
                  name=ln2,
                  interaction=sf:SystemFrame{
                     systemName=sn
                  }
               }
            }
         }
      }
   }{i.message->notEmpty()
      and m->oclIsKindOf(Message)
      and ((l1->oclIsKindOf(Human)
            and l2->oclIsKindOf(Component)) };
   enforce domain archModel s:System{
      name=sn,
      containsComps=comp
   };

   where{
      LifelineComponentToComponent(l2, comp);
      LifelineToServicePresentationAspectComponent(m2, comp);
   }
}
```

The following relation is in charge of describing the mapping from the Message, in the scenarios domain, to a service of a Presentation Aspect in the *archModel* domain. As can be observed, it proceeds similarly to "LifelineToServiceAspectConnector", by evaluating the same relations. However, in this case it is specified that the kind of Aspect is Presentation.

Table 8-16 Transforming a Lifeline to a Presentation Aspect

```
relation LifelineToServicePresentationAspectComponent
{
   cn: String;
   ks: String;
   as: archModel::Aspect;

   checkonly domain scenarios  mo:MessageOccurrenceSpecification{
      covered=l:Lifeline {},
      message=m:Message{}

   };

   enforce domain archModel c:Component{
      imports=as
   };
   when{
      LifelineToAspect(l,as, 'Presentation');
   }

   where{
      ks=KindService(m);
      LifelineToService(m, as, ks);
   }
}
```

Other relevant Idioms related to Aspects

In the Appendix B more idioms, similar to the previous ones, have been defined. The reader is referred to that Appendix to have more details about them. However, it is necessary to sketch some ideas of these idioms to have an insight of how detailed is the generated model. Some of these idioms are described in the following:

– By default, every service received by a Component Lifeline is mapped to a service in a Functional Aspect. This Aspect is imported by the Component resulting from mapping the Component Lifeline.

– If an Aspectual Message is received by a Lifeline, it will be mapped to a service belonged by an Aspect that will have the same concern as that specified in the Aspectual Message. Similarly to the previous one, this Aspect will be imported by the Architectural Element generated from the Lifeline.

– If there is an interaction with a COTS Lifeline, a Component will be enforced with the same name as the Lifeline. In addition, every send/receive message will be mapped to an Integration Aspect owned by this Component.

– The sequence of the messages is also analysed in order to determine the Weaving relations. If two MessageOccurrenceSpecifications are covered by means of the same BehaviourExecutionOccurrence, they are executed one after the other and belong to two different aspects, a Weaving relationship is established between them.

8.4 PROCESS FOR SYNTHESIS AND TRANSFORMATION

In a similar way to the previous activities of ATRIUM a process has been defined for the Synthesis and Transformation activity that is depicted in Figure 8-9. As can be observed, its definition is straightforward, where only three steps have been identified.

The first activity is related to the selection or definition of the transformations to be applied. Because different Architectural Models could be applied for the generation of the proto-architecture, the analyst should select the transformations to be applied among that defined. In this sense, we should take into account that three kinds of transformations where defined: architectural transformation patterns, Architectural Style-oriented transformations and idioms-oriented transformations. The first kind of transformation should be selected by default because it facilitates the generation of those architectural elements support by most of the existing architectural models. The second kind of transformation should be selected according to the Architectural Style to be applied in the system-to-be. Finally, the third kind of transformation should be selected to take into account the constraints and properties of the target architectural model that must be defined by means of those idioms.

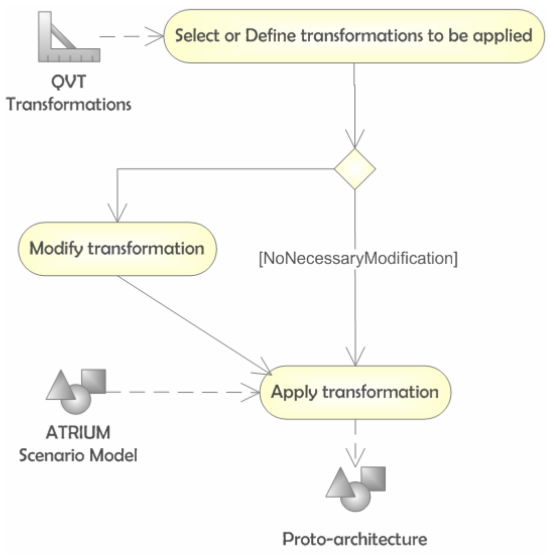


Figure 8-9 Describing the process for the Synthesis and Transformation activity

It is also possible that if none of the defined transformations is appropriate for his/her purposes, he/she could define the necessary transformation by using QVT Relations. It is recommended that analysts define the Architectural Generative transformation at the beginning, followed by the Architectural Style-oriented transformations and, finally, the idioms-oriented transformations. In this sense, the analyst would have more facilities not to introduce overlapping transformations. In addition, the common Relations should be included in the Architectural Generative transformation to be reused by the idioms and Architectural Style-oriented transformations.

The second step is optional, and it must be applied only if the transformations should be modified. For instance, the analyst could decide to introduce new idioms he/she has detected, the application of different styles, etc.

The last step of the process is fully automated. Once the analyst has selected the appropriate transformations, they are applied on the ATRIUM Scenarios Model to generate the proto-architecture. In this sense, the tool MORPHEUS plays an important role as is described in the following chapter.

8.5 CONCLUSIONS

During this chapter, a process has been defined for the last activity of ATRIUM: Synthesis and Transformation. The main aim of this activity is to generate a proto-architecture. It is used as a first draft of the SA for the system-to-be to be refined in a later stage of the software development. It facilitates that the analysis performed in the previous activity by defining the ATRIUM Scenarios Model to obtain an improved comprehension of the system, can be traced to a later stage with an automatic process.

In order to select the best approach for this activity, several applicable alternatives were studied and evaluated according to a set of defined needs. Among them, QVT emerged as the most proper solution because it satisfies most of the established goals.

By using QVT Relations, a set of transformations rules has been defined that faces the challenges described in the introduction of the chapter. The first was related to its applicability to the whole set of scenarios. As was observed, QVT supports the definition of keys to be applied in the process. This support facilitates the synthesis process by avoiding that objects with duplicated keys can be created. In addition, the set of transformation has been defined to deal with a set of scenarios because no constraint in this sense has been included.

Another challenge faced with this alternative was the introduction of styles during the transformation. As can be observed in section 8.3.2, it was straightforward the introduction of some restriction imposed by the ACROSET style by introducing some specific relations. This set of rules was defined in a different transformation. The main idea is that different transformations can be described that generate the proto-architecture applying different Architectural Styles. This means that different proto-architectures could be generated using the same set of scenarios and taking into account different Architectural Styles.

It is worthy of note that it is not necessary to provide as input a full set of scenarios required to describe the system behaviour. On the contrary, with only one scenario the generation can proceed. However, the *Incrementality* feature of QVT facilitates that as new scenarios are defined or modified or selected for the transformation, the proto-architecture can be automatically updated.

This facility of QVT for using more than two domains rises to the challenge described in the Introduction: how to establish mechanisms that are able to evaluate the proto-architecture being obtained. We are focused on the definition of `fault` stated by (Munson, et al 2006):

> *"A fault, by definition, is a structural imperfection in a software system that may lead to the system's eventually failing."*

They have stated the need of specifying well-defined methods of faults identification that are repeatable. The main idea we are focusing on is how to detect such faults at the specification level, that is, while the proto-architecture is being generated. An early detection of these faults will made meaningful strides to improve the development in terms of both quality and costs of the final product. The definition of a `Fault Model` that describes the faults that can appear at the specification level would mean a first step in this sense. It could be used not only as a guide for the analysts but also for the evaluation of the generated proto-architecture. In order to generate this model, it could be declared as another input domain in the transformations and, consequently, to obtain automatically a first evaluation. It would facilitate not only the detection of faults in the proto-architecture but also which scenarios are contributing to those faults by means of the *Traceability* feature of QVT. As far as we know there are not works copying with this issue and, it is currently our first challenge to be faced.

It must be also pointed out that this activity of ATRIUM has been defined to provide as much flexibility as possible in terms of the Architectural Model used for the generation of the proto-architecture. The main goal was to provide the analyst with facilities to generate it using that Architectural Model he/she considers more appropriate for his/her aims. For this reason, the set of Relations have been catalogued as architectural generative patterns, Architectural Style-oriented transformations and idioms. This facilitates that the same set of scenarios could be transformed to different proto-architectures by selecting the idioms that are specific to the desired Architectural Model.

Finally, one of the main challenges not only related to this activity but also to the whole definition of ATRIUM, is the traceability. This issue was the main motivation for the definition of ATRIUM and it has been faced in this activity thanks to the use of QVT. Its use plays a significant role for traceability top-down and bottom-up. The former is provided because the proto-architecture is generated automatically by establishing the appropriate transformations. Although it has not been dealt with in this work, the bottom-up traceability could be easily achieved as well. It is because QVT Relations derive a Trace Class from each used Relation in order to generate traceability maps. This ability is highly meaningful because a mapping is established between every element in the proto-architecture and its related element/s in the ATRIUM

Scenarios Model. It must be taken into account that a proto-architecture is generated from the set of scenarios, not a full specification. This means that it should be refined during the next stage of the software development. During this activity, if any change is required it could be traced-back to detect which scenario/s should be modified, helping to maintain the models up-to-date.

The work related to the transformation of the ATRIUM Scenarios model has been presented in the following publication:

– E. Navarro, P. Letelier, J. Jaén, I. Ramos, "A generative proposal for proto-architectures exploiting Architectural Styles", First European Conference on Software Architecture (ECSA'07), September 24-26, 2007, Aranjuez (Madrid), Spain, (submitted).

"Anyone who has lost track of time when using a computer knows the propensity to dream, the urge to make dreams come true and the tendency to miss lunch." —
Tim Berners-Lee

CHAPTER 9
MORPHEUS: A Tool for ATRIUM

9.1 INTRODUCTION

Nowadays, automation is becoming one of the principal means to achieve a greater productivity and a higher quality product. This is due to the trend of increasing acquirer's satisfaction of the developed and delivered software product. For this reason, its introduction in this proposal was compulsory to provide support to the whole set of defined models and assist as much as possible in the process. In its definition, several facts were considered as is described in the following.

The main idea behind this tool is to offer a graphical environment for the description of the different models because it provides the analysts with an improved legibility and comprehension. It must also be considered that three models are used in the description of ATRIUM. For this reason, the tool should provide the facilities proper of each model.

However, the support of the tool would be quite limited if it only provides graphical notation. For this reason, several mechanisms should also be included in their definition to facilitate their exploitation. In this sense, the tool should provide facilitates for analyzing architectural alternatives because this is one of the main goals of ATRIUM. In addition, it must also provide automatic support for model transformation.

Considering these facts, a tool called MORPHEUS has been developed. Due to the fact that it has to manage each model used by ATRIUM three different environments are provided:

– *Requirements Environment* provides user with a requirements metamodelling tool for both describing requirements model customized according to project's specific needs and its later exploitation and analysis. More details about this environment are presented in section 9.3.

– *Scenarios Environment* has been expressly developed to describe the ATRIUM Scenarios. It also supports the generation of proto-architectures using the defined QVT rules. Section 9.4 presents more details about the environment.

– *Software Architecture Environment* makes available a complete graphical environment for the PRISMA AO-ADL so that the proto-architecture obtained from the Scenarios Model can be refined. This environment is introduced in section 9.5.

In the next section, the selected technology for the graphical support is briefly presented. The main conclusions obtained round up the chapter.

9.2 TECHNOLOGIC DECISSIONS FOR MORPHEUS

In order to provide tool support for this proposal, several alternatives were studied. The first one was to exploit the use of UML profiles to describe the established metamodels. The corresponding models could then be elaborated in a straightforward way by using any CASE tool that supports UML. However, this alternative exhibits limitations because most of the existing tools provides a poor support for UML profiles.

For this reason, other analysed alternative was to follow the trends associated to Meta-CASE and Domain Specific Modelling. They are a promising emergent technology thrust by current interest into Model Driven Development (Mellor et al., 2004). In this sense, these Meta-CASE tools, such as MetaEdit+ (Kelly et al., 1996), have been developed to provide the user with immediate modeling support according to a customizable metamodel. However, it was detected that this alternative presents problems when more than a metamodel must be exploited. This is highly relevant in ATRIUM because three metamodels, with their corresponding models, are used. In addition, they do not provide support for the transformation needed for the *Synthesize and Transform* activity of ATRIUM.

For this reason, the selected alternative was to develop a tool specific to ATRIUM. There are several tools oriented to the visual modelling such as Visio (Visio, 2003). Most of them provide mechanisms to extend their functionality. This means that there is no need to develop a tool from scratch but extending any of the existing ones. We considered and reviewed several of these tools as support for ATRIUM modelling, but we finally selected Visio. It is due to the fact that Visio allows straightforward management, both for using and modifying, shapes. This characteristic is highly relevant for our purposes because all the kinds of concepts that are included in our metamodels can easily have different shapes that facilitate the legibility of the model. It also provides an additional advantage because Visio can be integrated in other tools facilitating they can be customized according to the specific needs. With this aim, the Microsoft Office Visio Drawing Control 2003 has been used. It is a Microsoft ActiveX control that provides full access to the Visio object model (API) and its user interface. This drawing control has been embedded in MORPHEUS to provide it with a drawing surface for displaying, editing and removing shapes. In addition to this drawing surface, the user is provided with all the functionalities that Visio has, that is, she/he can manage different diagrams to properly organize the specification, make zoom to see more clearly details, print the active diagram, etc.

The basic architecture of MORPHEUS, related to the use of Visio is depicted in Figure 9-1. MORPHEUS acts a container for the three environments described in the Introduction. Each one of these environments is a container that references an instance of the Visio Drawing Control for the graphical definition of its corresponding model. It must be mention that each instance loads a Visio document. For this reason, each MORPHEUS project integrates three Visio documents to define the requirements, scenarios and Software Architecture of the system-to-be.

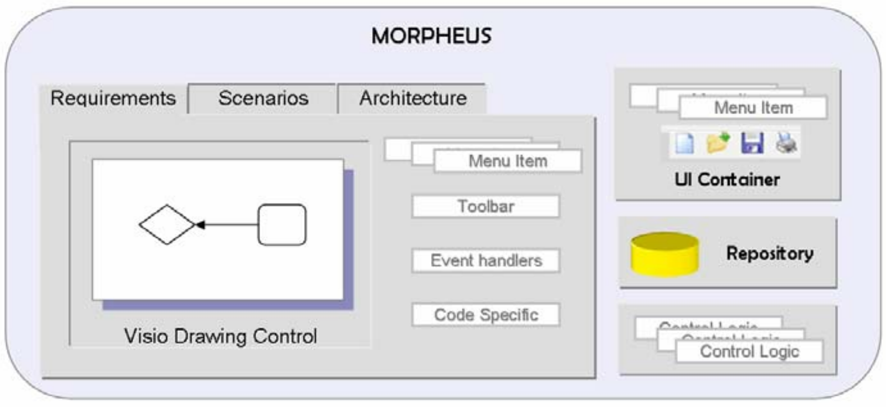


Figure 9-1 Sketching the MORPHEUS Architecture

Each environment introduces a component in charge of handling the events triggered when shapes, and diagrams are managed. It also contains its specific menu items and toolbar to provide an easy access to its functionality. Some components are also contained in each environment to support its management.

MORPHEUS also provides several menus and toolbars that are shared for all the environments that facilitate access to the printing, zoom, etc. It also encapsulates several components to control the whole tool and provide access to the repository shared for the three environments.

9.3 REQUIREMENTS ENVIRONMENT

With the aim of supporting the metamodeling proposal and its capabilities for tailoring to the specific project needs described in the chapter 5, the Requirements environment in MORPHEUS has been developed. It is not only able to define both new kinds of artifacts and relationships but also to instantiate and exploit them. For this reasons, its environment has been split into two different contexts, as shown in Figure 9-2. The first one allows analysts to establish the metamodel to be used; and, the second one provides analysts with facilities for modelling according to the defined metamodel. In this sense, the developed elements are described in the following sections.

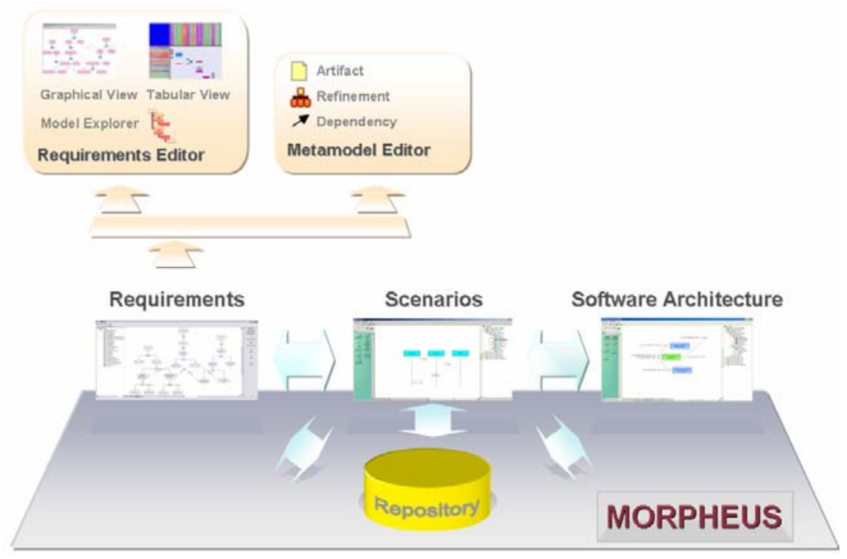
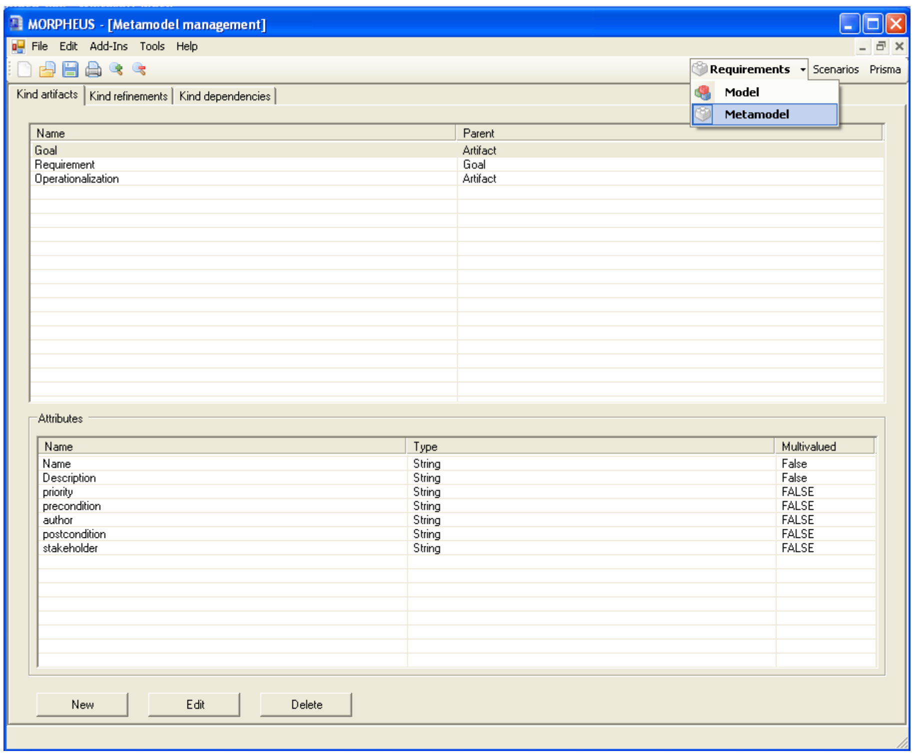


Figure 9-2 Requirements Environment

9.3.1 Requirements Metamodel Editor

It allows the definition of types of artifacts and relationships according to the specific needs of expressiveness. With this purpose, Figure 9-3 shows what MORPHEUS looks like when this context is activated. It can be observed that the analyst can access to this context by means of a combo box situated in the Environments toolbar at the top of MORPHEUS. Depending on the active context, a different bitmap is shown to facilitate its distinction.



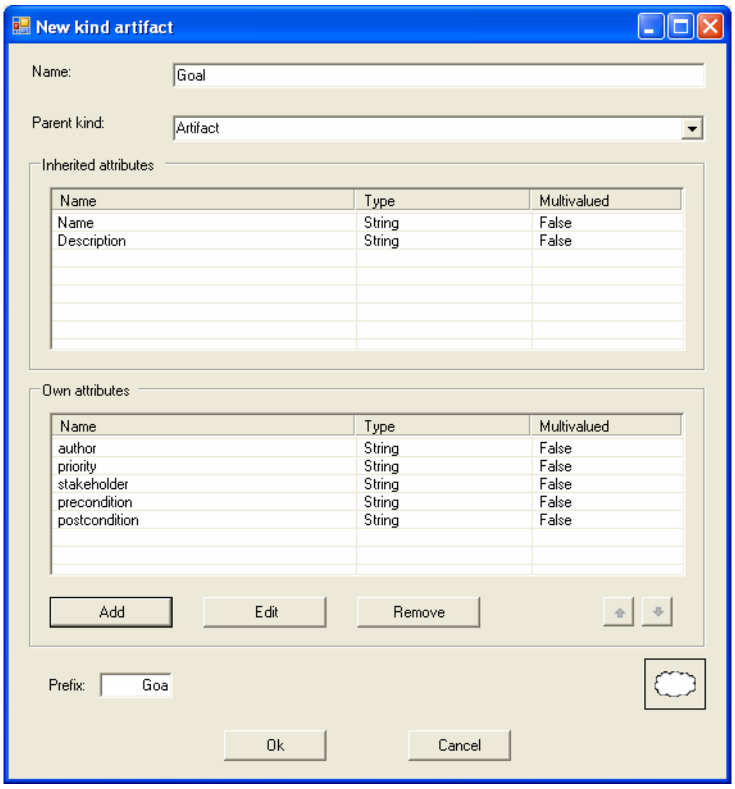
Name	Parent
Goal	Artifact
Requirement	Goal
Operationalization	Artifact

Name	Type	Multivalued
Name	String	False
Description	String	False
priority	String	FALSE
precondition	String	FALSE
author	String	FALSE
postcondition	String	FALSE
stakeholder	String	FALSE

Figure 9-3 Metamodel Editor

Whenever the Requirements Metamodel is being defined, the analyst can define new kinds of *artifacts*, *refinements* and *dependencies*. In the Figure 9-3, it can be observed that three kinds of artifacts have been defined. *Goal* and *Operationalization* have been defined as child of *Artifact*. The environment provides this kind by default in order to follow the guidelines for meta-modelling described in chapter 5. If *Artifact* is defined as parent the attributes *Name* and *Description* are inherited automatically (see section Attributes in Figure 9-3). However, *Requirement* has been defined as child of *Goal* inheriting all the attributes defined by it. These facilities are also provided to describe new kinds of refinements and dependencies by means of the types *Refinement* and *Dependency*, respectively.

It can be observed that the Requirements Metamodel Editor environment supports three operations to specify the Requirements metamodel: *New*, *Edit* and *Delete*. The first one allows the analyst to create new kinds of *Artifacts*, *Refinements*, and *Dependencies*. When *New* is pressed the form depicted in Figure 9-4 is shown. It facilitates the definition of attributes owned by the Artifact being defined and the inspection of the inherited attributes. The owned attributes can be created, modified or eliminated. Whenever an attribute is being created or modified, the form depicted in the Figure 9-5 is shown, to describe its type and select if it is enumerated and/or multi-valued. It can be observed that two buttons are situated at the bottom of the form. They are used to assign a weight to each enumerated value regarding its position in the list. This facility is highly relevant to facilitate the analysis process presented in section 9.3.3.


New kind artifact

Name:	Goal
Parent kind:	Artifact

Inherited attributes

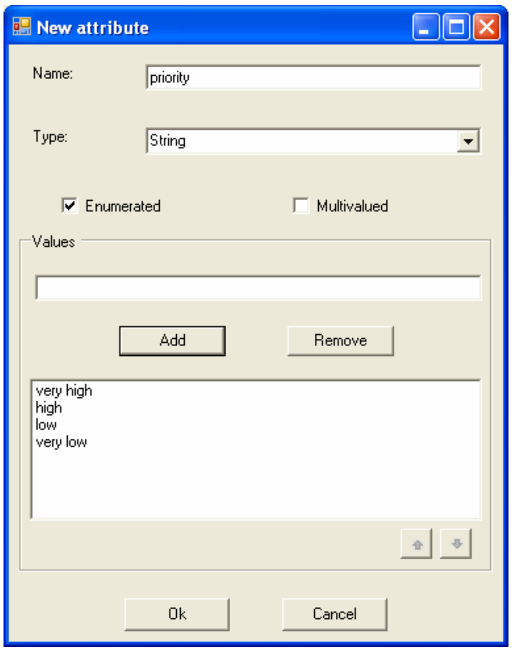
Name	Type	Multivalued
Name	String	False
Description	String	False

Own attributes

Name	Type	Multivalued
author	String	False
priority	String	False
stakeholder	String	False
precondition	String	False
postcondition	String	False

Prefix: Goa

Figure 9-4 Describing Meta-Artifacts for a kind of Artifact


New attribute

Name: priority

Type: String

[✓] Enumerated

[ ] Multivalued

Values

very high
high
low
very low

Figure 9-5 Describing a new attribute

When a kind of *Artifact* is being defined, the analyst must select as well the shape that will represent it graphically by means of button situated at the bottom in the Figure 9-4. Whenever it is pressed, the form shown in Figure 9-6 is presented to facilitate their selection.

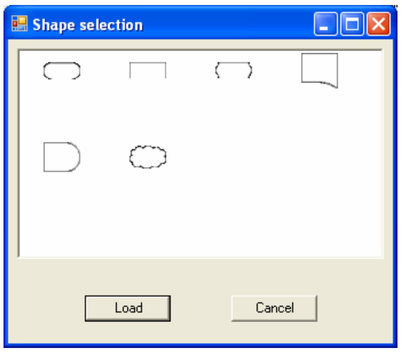
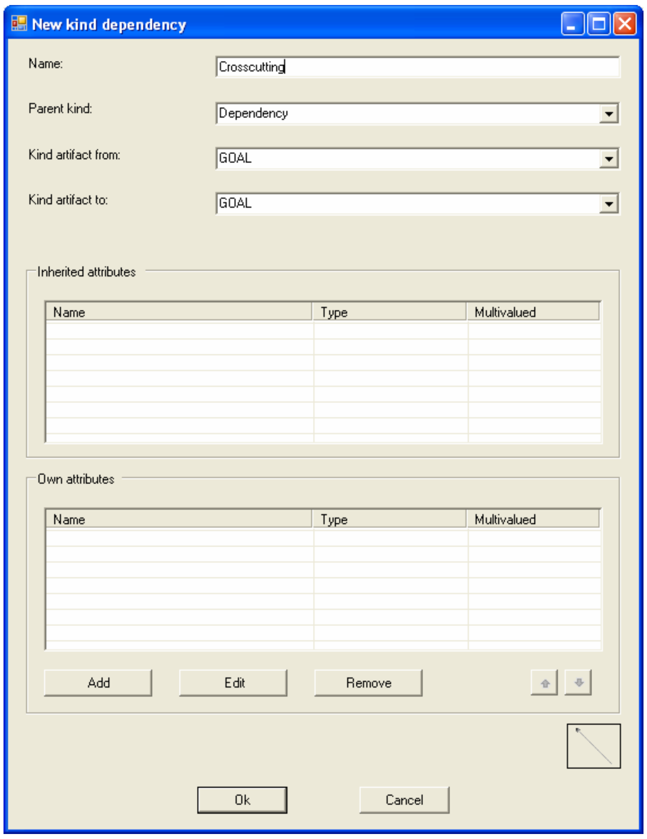


Figure 9-6 Selecting the shape for the kind of Artifact



Name:	Crosscutting
Parent kind:	Dependency
Kind artifact from:	GOAL
Kind artifact to:	GOAL

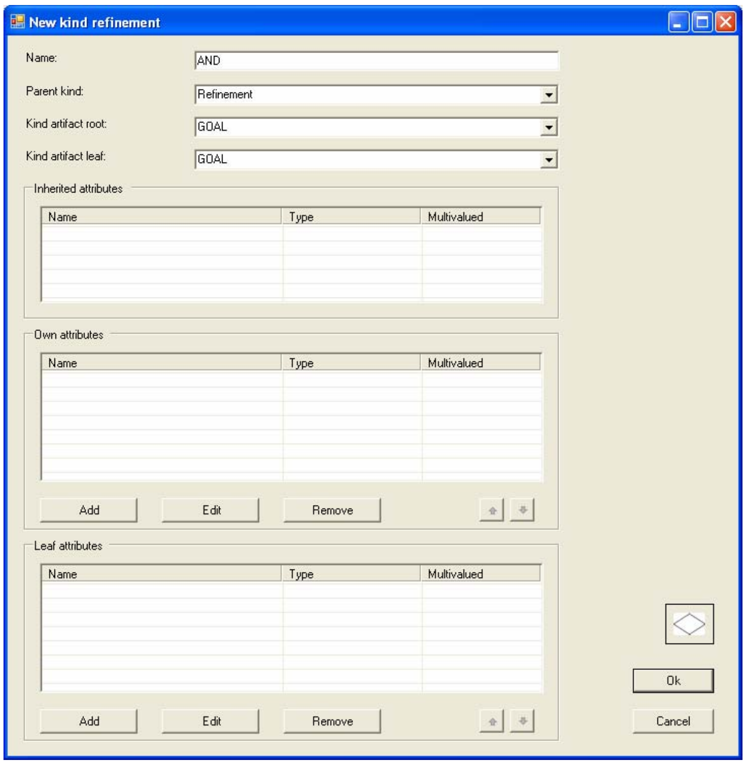
Name	Type	Multivalued

Name	Type	Multivalued

Figure 9-7 Describing a kind of Dependency

Figure 9-7 shows how new kinds of *dependencies* can be specified. It can be observed that when a new kind of *dependency* is being defined another kind of *dependency* is selected as parent. It means that the new kind will inherit all the attributes the parent kind has. New attributes can be also defined. The kind of artifacts that this new kind is going to relate must be also selected. Finally, the shape to describe the relationship must be selected by means of the button situated at the bottom.

Figure 9-8 shows how new kinds of *refinements* can be defined in a similar way to the previous one. It can be observed that attributes can be in the leaves of the new kind of *refinement*. It gives more facilities to include any necessary information in the leaves of the relation.


New kind refinement

Name:	AND
Parent kind:	Refinement
Kind artifact root:	GOAL
Kind artifact leaf:	GOAL

Inherited attributes

Name	Type	Multivalued

Own attributes

Name	Type	Multivalued

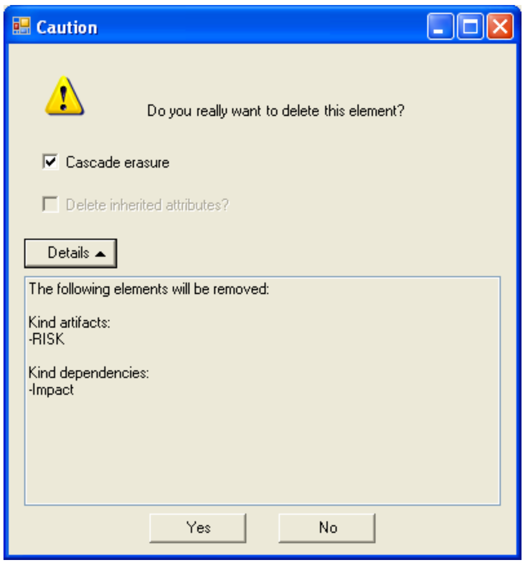
Leaf attributes

Name	Type	Multivalued

Figure 9-8 Describing a kind of Refinement

As soon a new kind of *artifact*, *dependency*, or *refinement* is created, it is automatically made available for the analyst in the Requirements Editor environment.

In a similar way, when the operation *Delete* of the context is applied to a kind of *artifact*, *dependency*, or *refinement*, both it and all its corresponding instances are automatically eliminated from Metamodel and the model being defined, respectively. In this case, the form illustrated in the Figure 9-9 is shown. It is used not only to confirm if the user wants to eliminate it but also to check if a cascade erasure must be applied. This means that all the kinds defined as child of that being erased will be also deleted. Alternatively, the analyst can select that the inherited attributes of all these kinds only will be erased as well. These changes are also propagated to the instances of the child kinds. If a kind of artifact is being eliminated then every kind of dependency and/or refinement connecting that kind will be also deleted propagating that elimination to their instances. This avoids that un-connected relationships can appear in the model.



Do you really want to delete this element?

☑ Cascade erasure

☐ Delete inherited attributes?

The following elements will be removed:

Kind artifacts:
-RISK

Kind dependencies:
-Impact

Figure 9-9 Eliminating a kind of Artifact

If the operation *Edit* is performed on one of the defined kinds then the form of the Figure 9-4, Figure 9-7 or Figure 9-8 is again shown to edit, delete or create attributes of the kind of *artifact*, *dependency* or *refinement*, respectively. The introduced changes are automatically applied to its instances. It is worthy of note that if the kind of an attribute, of a kind of *artifact*, *dependency* or *refinement*, is changed then all the instances of that kind reset their value to null unless a casting between the types can be applied.

9.3.2 Requirements Editor

Figure 9-10 shows how MORPHEUS looks like when this environment is loaded. It can be observed that it is defined mainly by means of three components. One of them is the *Model Explorer* that provides facilities to navigate through the Requirements model being defined in an easy an intuitive way. The *Model Explorer* is shown on the left in the Figure 9-10 where the different defined artifacts of the EFTCoR project can be examined. Several elements are identified in this component: *dimensions*, *artifacts*, *diagrams*, and the *project* on top of the tree. *Dimensions* are used to classify the *Artifacts* that are being specified as instances of the kinds of *artifacts* defined in the metamodel. It can be observed that the defined *dimensions* for the EFTCoR correspond to the ISO 9126 taxonomy. The *diagrams* help to group those artifacts that have some kind of relationship among them. Each element has a different icon to improve the legibility of the model. In addition to the navigation through the model, the *Model Explorer* allows the management (creation, modification and deletion) of each kind of element. It also facilitates the modification of the preferences of the project by selecting the item on top of the tree and showing the form on Figure 9-11. It facilitates that the name of the project can be changed, the *views* can be customized, and the instances of kinds *artifacts, dependencies and refinements* can be hidden or make visible in the diagrams depending on their selection.

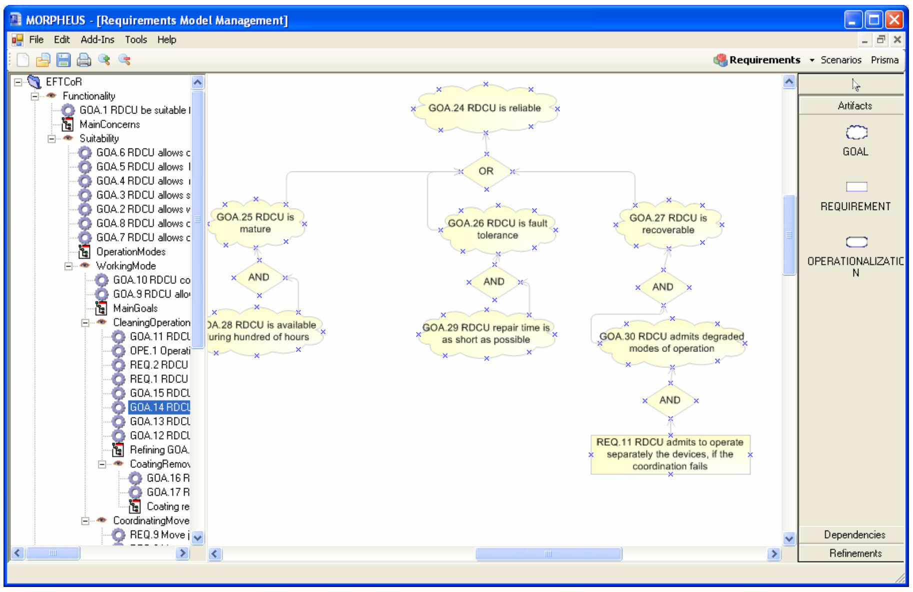
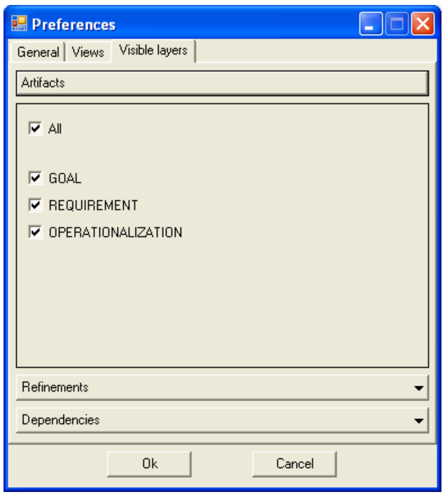


Figure 9-10 MORPHEUS while loading the Requirements Model Management environment

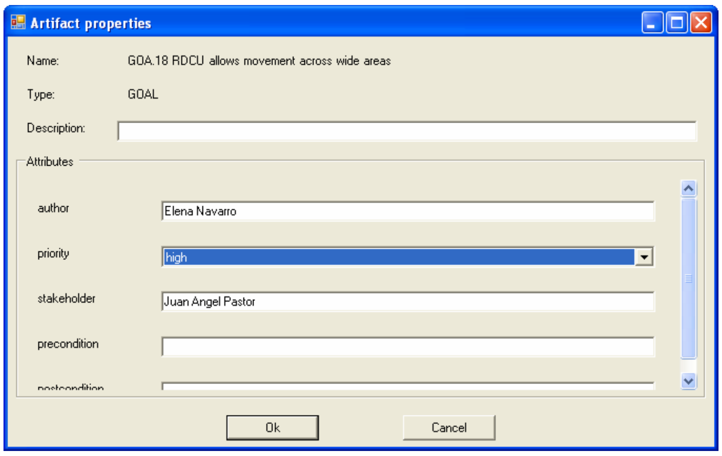
Other component included in this environment is the *Graphical View* that is situated in the middle of the Figure 9-10. This component is used for graphical modelling according to the loaded metamodel. For this reason, any defined relationship or artifact can be graphically created, modified or deleted. By selecting any element in the graphical view, the form depicted in Figure 9-12 is activated to facilitate its specification. This form is automatically customized according to the kind of the instance being specified. This customization takes into account the kind of the attribute as well. For instance, it can be observed in the Figure 9-12 that the attribute *priority* is shown to the user as a combobox because it was defined as an enumerated attribute.



☑ All

☑ GOAL
☑ REQUIREMENT
☑ OPERATIONALIZATION

Figure 9-11 Customizing the preferences of the project



Name: GOA.18 RDCU allows movement across wide areas

Type: GOAL

Description:

author	Elena Navarro
priority	high
stakeholder	Juan Angel Pastor
precondition	
postcondition	

Figure 9-12 Describing the properties of an artifact

Finally, the component situated on the right of the figure makes available the specified Requirements metamodel to the analyst. It can be observed that each kind of *artifact*, *dependency* and *refinement* is shown in this component according to the selected shape that is going to describe them. It facilitates that the analyst just selects the needed kind and click on the Graphical View to create any instance he/she needs. It must be highlighted that this component is automatically updated as soon as the Requirements metamodel is changed.

The Requirements Model Management environment also provides an alternative way for analysing the model by means of *Tabular Views*. It can be observed in the Figure 9-13 that the view called “Tabular View” selected on the *Model Explorer* is shown in the middle of the form as a bi-dimensional table where *artifacts* are situated on the first row and column, and the relationships among them in the corresponding cells. This alternative is highly useful when the number of artifacts and relationships is high, facilitating the analyst a proper way of analysing the specification. The analyst can create as many Views as she/he needs only by selecting the project in the Model Explorer. These views can be customized by showing only those *dimensions*, *artifacts* and *relationships* selected. Figure 9-14 shows that the analyst can also select the colour of the dimension to improve the legibility of the View.

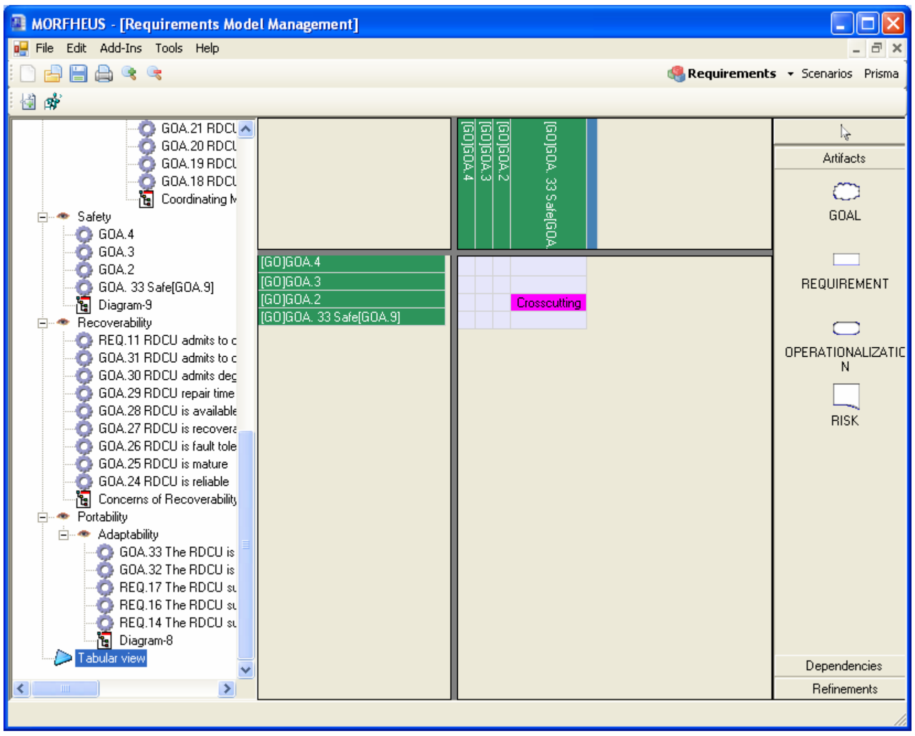
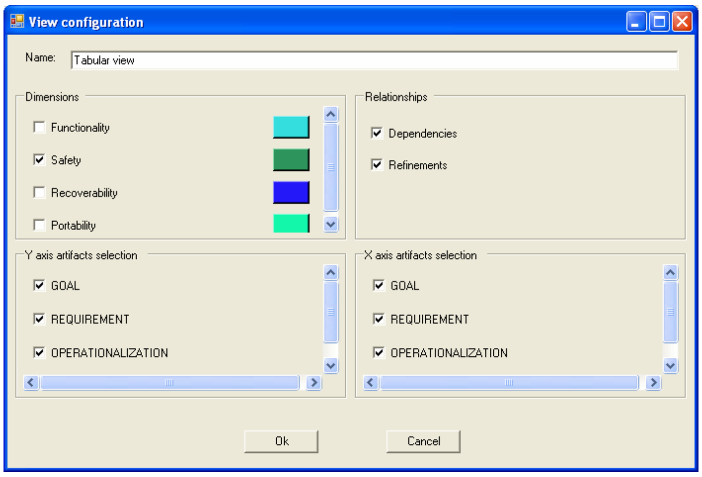


Figure 9-13 MORPHEUS while loading a Tabular View in the Requirements Model Manager


View configuration

Name: Tabular view

Dimensions

- [ ] Functionality
- [x] Safety
- [ ] Recoverability
- [ ] Portability

Relationships

- [x] Dependencies
- [x] Refinements

Y axis artifacts selection

- [x] GOAL
- [x] REQUIREMENT
- [x] OPERATIONALIZATION

X axis artifacts selection

- [x] GOAL
- [x] REQUIREMENT
- [x] OPERATIONALIZATION

Figure 9-14 Configuring the tabular view

In addition, MORPHEUS has been developed with capabilities to extend its functionality with techniques of analysis and exploitation. The main aim is that as new metamodels are defined, their related techniques can also be included and exploited in MORPHEUS. An example of how this capability has been used is presented in the following section.

9.3.3 An add-in for customizing the analysis process

An add-in for Goal Model analysis based on satisfiability propagation has been developed. It is based on the recommendations described in chapter 5. It has to be pointed out that this extension facility is mainly supported by the MORPHEUS API, an interface that facilitates access to the repository for its management. Figure 9-15 shows how this add-in has been designed. It has been split into four main components: a Rules Editor, a Rule Compiler, a Code Compiler and a Propagation Processor.

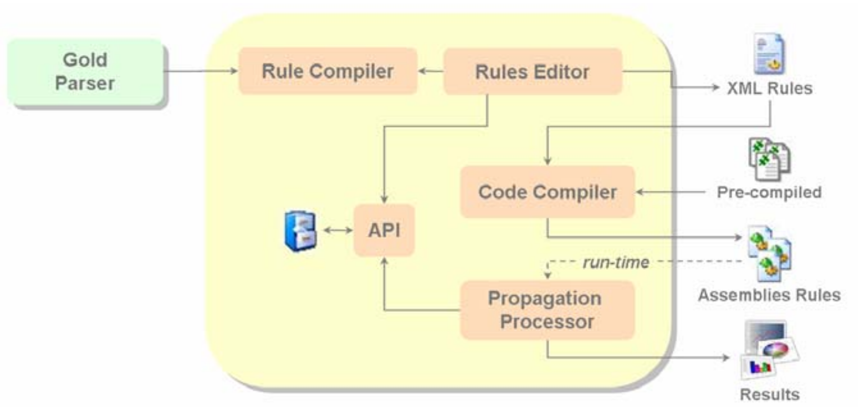


Figure 9-15 A sketched view of the Rules add-in

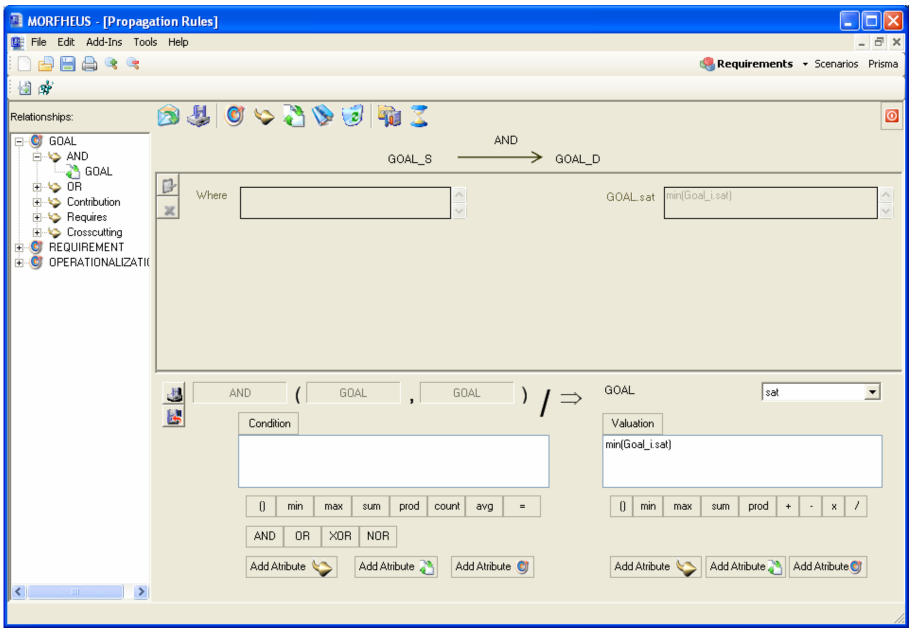


Figure 9-16 MORPHEUS while loading the Rule Editor

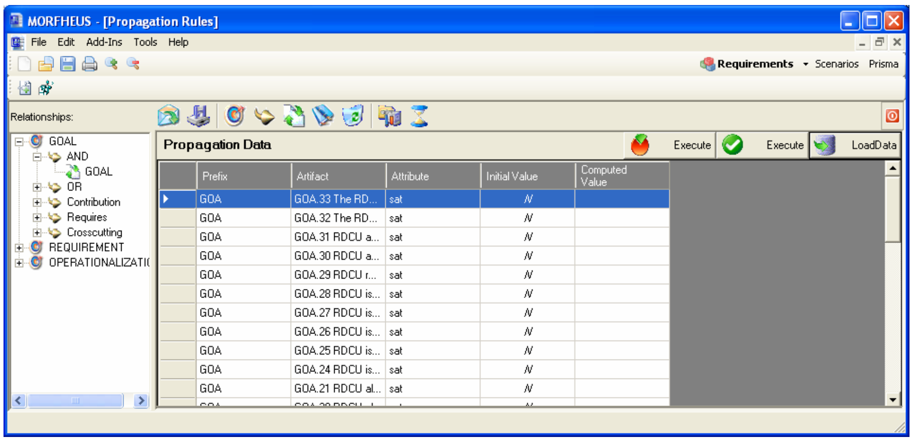
Figure 9-16 shows what MORPHEUS looks like whenever the Rule Editor is loaded. For its development, several alternatives were evaluated. However, the usability of the proposal was one of the main characteristics to be achieved. For this reason, a user interface (Rule Editor in Figure 9-15) that allows the analyst to introduce the rules in a simple and comprehensible manner was developed. The Rule Editor is split into three main parts: a *Browser*, a *Rules Descriptor* and a

Editor. The *Browser* allows one to navigate through the kinds of artifacts and kinds of relationships among them. The *Rules Descriptor*, on the right of the *Browser*, displays the applicable rules, for a selected kind of destination artifact, relationship and source artifact. Figure 9-15 shows that "GOAL", "AND", "GOAL" are the selected destination artifact, relationship and source artifact, respectively, that only has associated a rule that is described on the *Rules Descriptor*. It can be appreciated that the *when* text box describes the condition and next to it appears the *valuation* that must be performed whenever the rule is applicable. Close to the valuation appears the attribute of the kind of artifact that must be valuated. Beneath the rules descriptor, the *Editor* permits to edit the condition, the valuation, and the attribute to be valuated of a selected rule. This control provides the analyst with several buttons and capabilities that prevent him from knowing any detail about how his/her metamodel is described in the repository or how rules are internally implemented.

In addition, a syntactic checking and the generation of its C# code is performed when a rule is being defined by using the *Rule Compiler* generated using Golden Parser (GOLD, 2005). This is a free parsing system that can be used to develop one's own programming languages, scripting languages, and interpreters by previously writing your grammar using BNF. The BNF, which was described for the condition (Table 5-25), and the valuation (Table 5-26), was introduced in GOLD. Then, the GOLD Parser Builder was used to analyze this grammar and create the Compiled Grammar Table file (CGT) used by a compilation engine. It uses this CGT file to generate a C# skeleton program with a custom parser class that acts as a template for parsing any source satisfying the BNF grammar. By means of this template, the specific compilation to the objective code can be described. In this case, the *Rule Compiler* was developed to perform the translation to C# code making available artifacts and relationships from the repository, and that computation which was needed for both condition and valuation. Therefore, for each rule its valuation and condition description along with their respective compilation to C# is computed by means of the *Rule Compiler* and passed to the *Rule Editor*, which stores them in a XML rules file to be lately used for the *Code Compiler*.

The performance could have been seriously compromised due to the necessity of representing these rules as code for its use in run-time when they are used to perform the propagation throughout the model. For this reason, the approach of dynamic compiling, while it is more complex, provides us with a proper solution. Microsoft .NET Code Document Object Model technology, as described by (Harrison, 2003), has been used for the implementation of the *Code Compiler*. By using *Code Compiler* (Figure 9-15) a set of assemblies, containing both the code of the rules and other functionality, is generated at run-time. For each rule, which is saved in a XML rule file, a C# class is generated which inherits from IRule. It is an abstract class with two abstracts methods to override for each inherited class: applicable function, which checks if the rule can be applied; and valuate function, which performs the propagation computation. This class also has a set of functions to perform the minimum, maximum, etc. Therefore, while generating code, each rule C# class is going to override the abstract methods with that code stored in the XML rule file. Other classes are also used for the management of the generated classes, which are previously pre-compiled to speed up this process.

Afterwards, these assemblies are accessed by the *Propagation Processor* to perform the propagation on a specific Goal Model and generate the results. In terms of integration, the *Propagation Processor* makes use of the MORPHEUS API to access the model and pass through the relations and artifacts retrieved from the repository. Once the propagation is performed, the results are shown to the user using a specific interface that is shown in the Figure 9-17 where the initial and computed values of the attributes are displayed.



Prefix	Artifact	Attribute	Initial Value	Computed Value
GOA	GOA.33 The RD...	sat	N	
GOA	GOA.32 The RD...	sat	N	
GOA	GOA.31 RDCU a...	sat	N	
GOA	GOA.30 RDCU a...	sat	N	
GOA	GOA.29 RDCU r...	sat	N	
GOA	GOA.28 RDCU is...	sat	N	
GOA	GOA.27 RDCU is...	sat	N	
GOA	GOA.26 RDCU is...	sat	N	
GOA	GOA.25 RDCU is...	sat	N	
GOA	GOA.24 RDCU is...	sat	N	
GOA	GOA.21 RDCU al...	sat	N	

Figure 9-17 MORPHEUS while loading propagation data

One of the advantages of this add-in is its ability to be customized according to any kind of *artifact*, *dependency* and *refinement*. It means that its application is not constrained to Goal Models, but any kind of Metamodel can be exploited by describing properly its rules.

9.4 Scenarios Environment

This environment has been developed to facilitate the specification and exploitation of the ATRIUM Scenario Model described in the chapter 7. With this aim, it has been split into two different contexts as Figure 9-18 depicts.

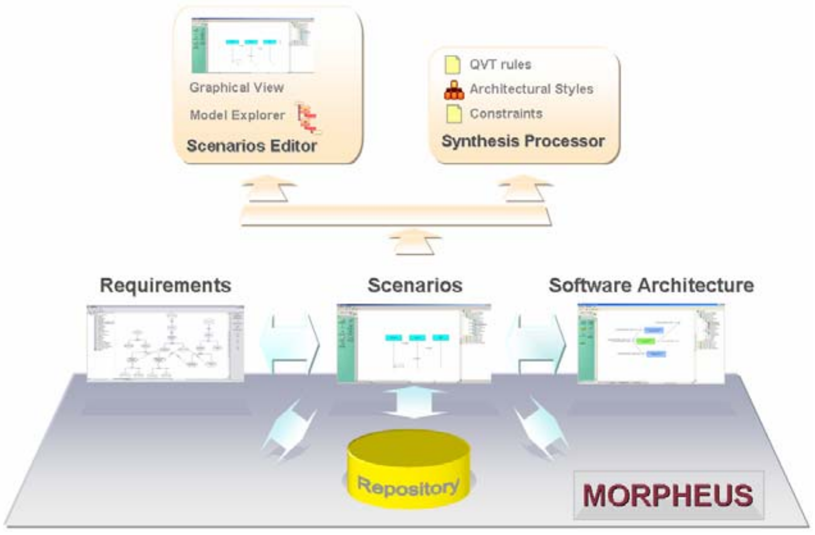


Figure 9-18 MORPHEUS: Capabilities of the Scenarios Environment

The first context is called the *Scenarios Editor*. It provides the analyst with facilities to describe the scenarios according to the notation described in chapter 7. As can be observed in the Figure 9-19, and similarly to the Requirements Model Management environment, it includes a *Model Explorer* to navigate through the Scenario model being defined in an easy an intuitive way. It is pre-loaded with part of the information of the Goal Model being defined. For this reason, the selected *operationalizations*, catalogued by their dimensions, are displayed. It facilitates to maintain the traceability between the Goal Model and the Scenarios Model. Associated to each operationalizations one or several scenarios can be specified to describe how the shallow architectural elements collaborate to realize that operationalization. The *Model Explorer* provides facilities for the manipulation of these scenarios. In the middle of the environment is situated the *Graphical View* where the elements of the scenarios can be graphically specified. On the right of the environment is situated the stencil developed for the Scenarios Model. It can be observed that each concept presented in the chapter 7 has a graphical shape to describe it.

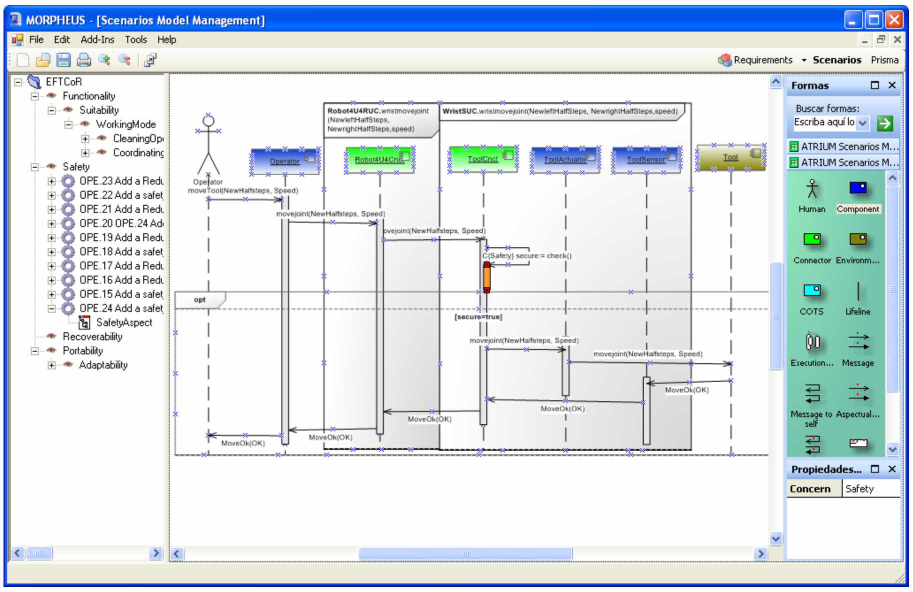


Figure 9-19 MORPHEUS while loading the Scenarios Management environment

The second context is the *Synthesis processor*. It is in charged of the generation of proto-architecture. For its development, the alternative selected was the integration of one of the existing model transformations engines considering that it has to provide support to the transformations described in chapter 8. Specifically, (ModelMorf, 2007) was selected because it supports the QVT-Relations language. It should be mention that this engine supports all the features described in chapter 8.2. Among them, it must be highlighted its support for multi-directional transformation specification and incremental transformations. This engine to perform the transformation accepts as inputs the metamodels and their corresponding models in XMI format. For this reason, the *Synthesis processor* first stores the Scenario Model in this format and, second, performs the transformation by invoking ModelMorf. The result is an XMI file describing the proto-architecture that is used by the Software Architecture environment, for its refinement. An example of a scenario in XMI format and the generated proto-architecture performed by using ModelMorf is presented in the Appendix D.

9.5 Architecture Environment

The third environment included in MORPHEUS is the Architecture Environment, as is described in Figure 9-20. It was developed to allow the analyst to refine the proto-architecture generated with the environment described above. Specifically, this environment provides support to the specification of Software Architecture using PRISMA as AO-ADL. Therefore, section 9.5.1 introduces the notation that is supported by the environment. Section 9.5.2 describes some of the capabilities implemented.

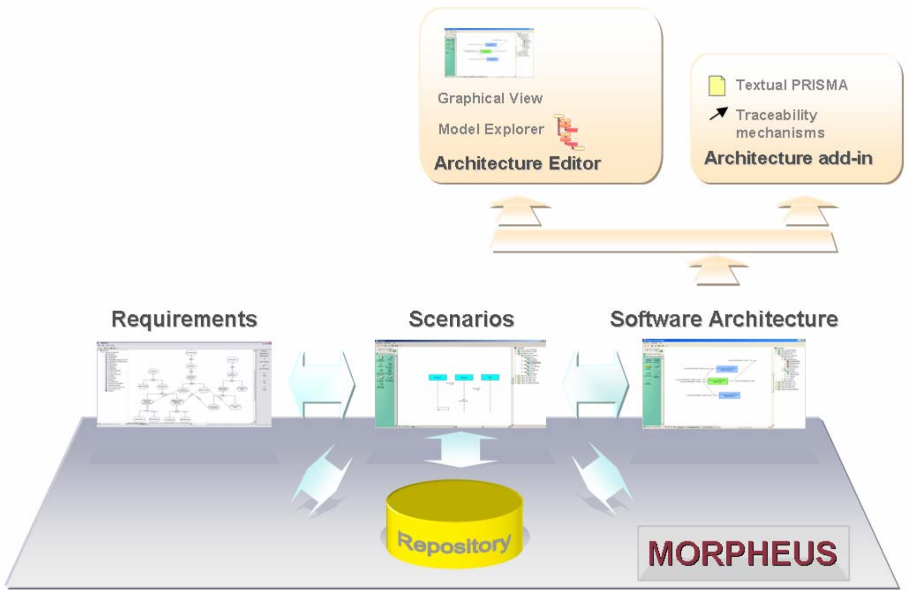


Figure 9-20. MORPHEUS: Capabilities of the Architectural Environment

9.5.1 Describing the notation

The PRISMA textual language could be used directly to model Software Architectures. However, for any modern modelling approach, it is important to have a visual representation of the specifications. This makes the application of PRISMA in the context of a modelling tool easier and more practical. Obviously, a textual representation is more suitable for some details of a PRISMA specification; for instance, to formulate the changes in the value of attributes by the execution of aspect services. Thus, only the main concepts and their relationships are visually specified; the rest of the concepts are represented in textual form and are included in the definition of the corresponding symbols.

We consider UML to be relevant as a graphical notation for PRISMA because it is a standard and popular notation. Although UML is intended for object-oriented modelling, thanks to its extension mechanisms that are associated to the definition of a UML profile, it can be customized to the particular needs of PRISMA specifications. These extension mechanisms are stereotypes, tagged values, and constraints, which are all used to define new derived concepts (metaclasses) from the standard UML metaclasses. For Software Architecture modelling, UML 2.0 (UML, 2005) includes the following concepts: component, connector, port and interface (required or provided). However, the provided expressivity is too basic in comparison with PRISMA. Furthermore, UML 2.0 does not include the aspect and weaving concepts.

(Perez et al., 2003) defined a profile that includes all the necessary extensions for using UML as a visual notation for PRISMA specifications. This profile has been implemented by extending an existing one as we describe bellow. Throughout the definition process of the PRISMA profile, especially for AOSD elements, it was taken into account the satisfaction of the requirements that (Aldawud et al., 2003) stated for defining a UML profile for AOSD:

I. *The Profile shall enable specifying, visualizing, and documenting the artifacts of software systems based on Aspect-Orientation.* This requirement has been satisfied by means of the stereotypes and their visual representation as it is described below.

II. *The Profile shall be supported by UML (avoid "Heavy-weight" extension mechanisms), this allows a smooth integrating of existing CASE tools that support UML.* This requirement is also satisfied due to the UML profile was defined according to the established construction rules in the UML specification (UML, 2005).

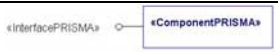
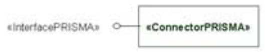
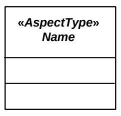
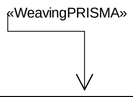
III. *The Profile shall support the modular representation of crosscutting concern.* The separation of concerns provided by aspects defining the architectural elements, along with the defined weaving relationship, allows us to identify and manage crosscutting in an early stage.

IV. *The Profile shall not impose any behavioural implementation for AOSD, however it shall provide a complete set of model elements (or Stereotypes) that enable representing the semantics of the system based on Aspect-Orientation.* No constraint has been defined about the implementation, only a proper semantic related to the way we use the ATRIUM elements at the requirements stage.

Following, we summarize the part of the PRISMA profile that is most related to aspect orientation. In this case, we are basically interested in representing aspects and weaving relationships. With this aim, we have derived these concepts extending the UML metaclasses. Table 9-1 shows the graphical notation used for representing aspect-oriented modelling in PRISMA. With respect to reuse, PRISMA allows us to define aspect types. There are some predefined stereotypes for modelling these aspect types, such as: «Quality», «Presentation», «Contex-awareness», «Coordination», «Navegation», «Functional», «Distribution», «Safety» and «Replication». As shown in Table 9-1, they are specializations of the stereotype «AspectPRISMA» and they have a common notation.

Similarly to other proposals that extend UML with aspect-oriented modelling, we allow the visualization of aspects and weaving relationships. Perez et al. have used the metaclass UML *Classifier* to derive the Aspect stereotype. Thus, the Aspect stereotype takes advantage of the structural and behaviour features that are available for UML metaclasses. *Dependency* has been chosen as the base class to represent weaving relationships. These selected metaclasses are the common choice in UML extensions for modelling aspects (Aldawud et al., 2003)(Suzuki & Yamamoto, 1999). In this sense, the main differences refer to the context in which aspect orientation is used. In our case, our concern is Software Architecture modelling according to PRISMA expressivity; however, in (Aldawud et al., 2003)(Suzuki & Yamamoto, 1999), for example, the extensions are made in the context of aspect design and programming.

Table 9-1 An extract of the PRISMA profile

Stereotype	**Base Class**	**Parent**	**Notation**
«ComponentPRISMA»	Component		
«ConnectorPRISMA»	Component		
«AspectPRISMA»	Classifier		Not represented explicitly
«AspectType»		AspectPRISMA	
«WeavingPRISMA»	Dependency		

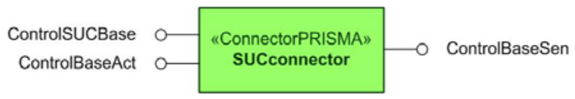


Figure 9-21 External view of a connector

Figure 9-21 and Figure 9-22 show diagrams that use the PRISMA profile. The external view shows components, connectors and their attachments. As an example, the Figure 9-21 shows the connector. For simplicity, we have decided to visualize in a separate view (internal view) the aspects of an architectural element. Thus, Figure 9-22 shows the internal view of the connector "SUCConnector" with its two aspects: "CProcessSUC" and "SMotion", whose stereotypes are «Coordination» and «Safety», respectively.

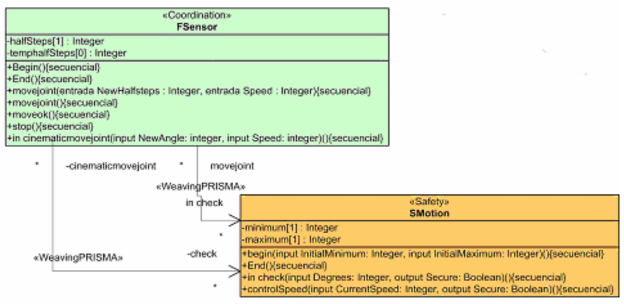

«Coordination»
FSensor
-halfSteps[1] : Integer
-temphalfSteps[0] : Integer
+Begin(){secuencial}
+End(){secuencial}
+movejoint(entrada NewHalfsteps : Integer, entrada Speed : Integer){secuencial}
+movejoint(){secuencial}
+moveok(){secuencial}
+stop(){secuencial}
+in cinematicmovejoint(input NewAngle: integer, input Speed: integer)(){secuencial}
-cinematicmovejoint
movejoint
«WeavingPRISMA»
in check
«Safety»
SMotion
-minimum[1] : Integer
-maximum[1] : Integer
+begin(input InitialMinimum: Integer, input InitialMaximum: Integer)(){secuencial}
+End(){secuencial}
+in check(input Degrees: Integer, output Secure: Boolean)(){secuencial}
+controlSpeed(input CurrentSpeed: Integer, output Secure: Boolean)(){secuencial}
«WeavingPRISMA»
-check

Figure 9-22 Internal view of the SUCconnector

9.5.2 Graphical Editor of PRISMA

Our graphical editor, which Figure 9-23 depicts, has not been developed from scratch. We have used the UML template of Microsoft Office Visio 2003 (Visio, 2003) to support the needs mentioned above and to take advantage of the functionality that is already provided by this element. Architecture environment allows us to specify, in a graphical way, PRISMA architectures and to generate automatically their PRISMA textual specification and/or their XML document.

It is worthy of note that an add-in is also available for Visio to translate a UML model to XMI. This means that other tools such as those listed in (Toval et al., 2003) can be used for semantic analysis. This allows for simple consistency checks and type checking in terms of defined OCL constraints of the PRISMA profile.

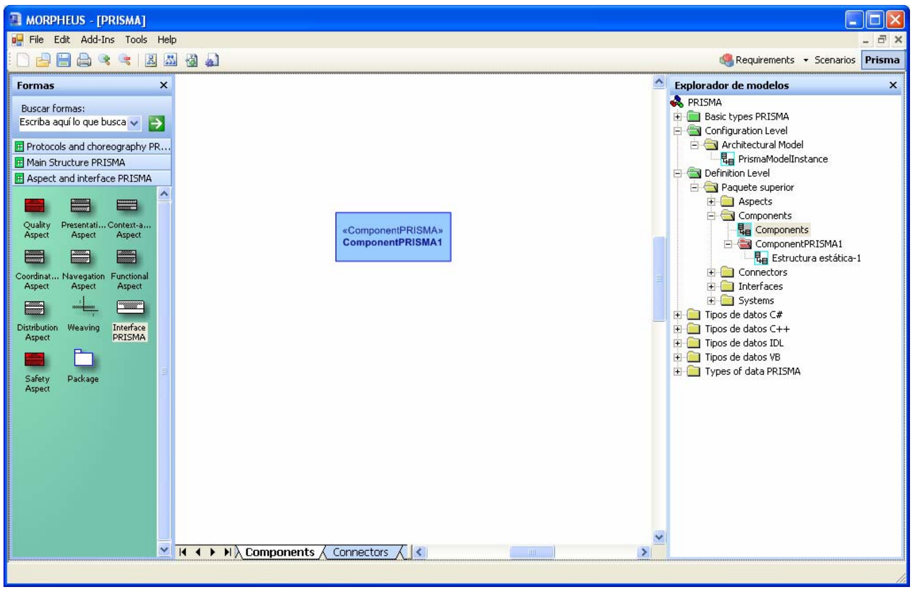


Figure 9-23. What MORPHEUS looks like whenever the Architecture Environment is active

Visual modelling in Visio is accomplished by means of *Stencils*. They are galleries of shapes that are organized according to their purposes, uses, etc. In the development of the Architecture environment, two stencils were specifically developed using the shapes that the Visio UML template provides. They are *Aspect and interface PRISMA* and *Main Structure PRISMA*. As can be observed in Figure 9-24, the former has been defined to allow the modelling of those aspects that are currently supported by PRISMA and the *weaving* relationships. It can also be observed that it is also possible the definition of *interfaces*. The latter shows the stencil for modelling *components*, *connectors* and *systems* along with the relationships made available to them, i.e., *binding* and *attachment*, to be properly glued together.

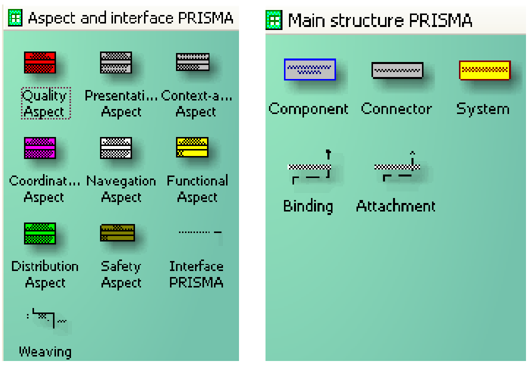
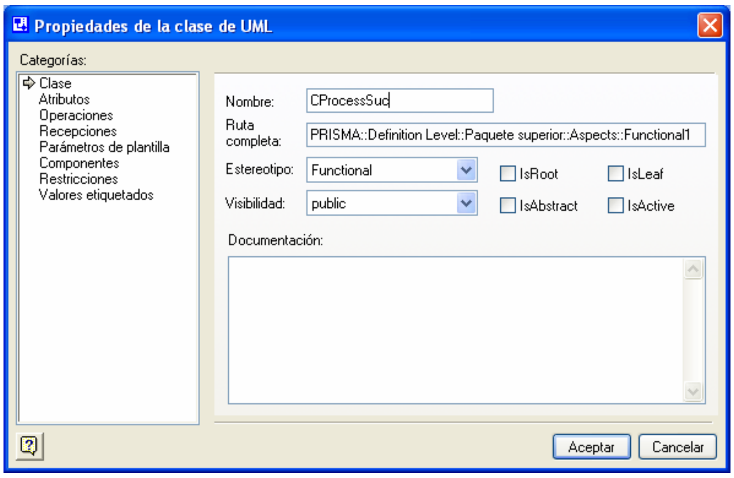


Figure 9-24. Developed Stencils for PRISMA

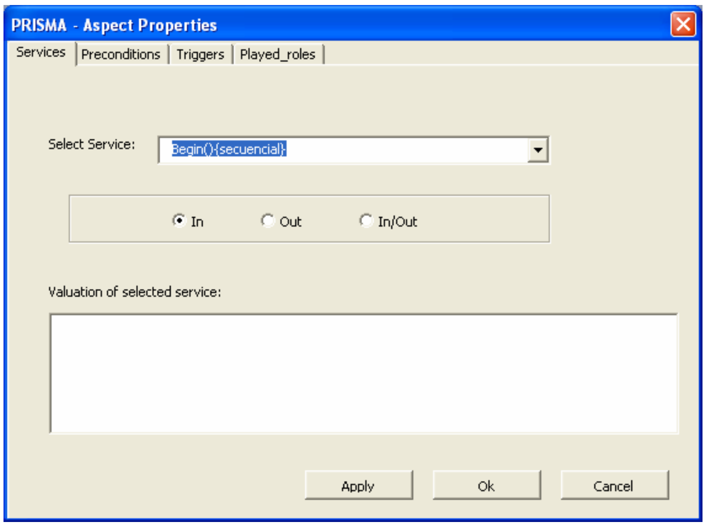
Each shape on these stencils is the visual representation of a stereotype that is defined in the PRISMA profile. This means that it inherits all the properties from the base class used for the stereotype in conjunction with others that are intrinsic. In this sense, most of the shapes in the *Aspect and Interface PRISMA* stencil allow us to model aspects: quality, presentation, context-awareness, coordination, navigation, functional, distribution and safety. As *classifier* is the selected base class, each aspect uses the utilities provided by the Visio UML template to specify the attributes and services of each aspect as Figure 9-25 shows.



Nombre: CProcessSuc
Ruta completa: PRISMA::Definition Level::Paquete superior::Aspects::Functional1
Estereotipo: Functional
Visibilidad: public
IsRoot ☐ IsLeaf ☐ IsAbstract ☐ IsActive ☐

Figure 9-25 Describing attributes (Atributos) and services (operaciones) of an Aspect

Additionally, a new utility has been developed in the Architecture environment to allow the architect to establish certain properties that are only made available to aspects, i.e., to type each service as *in*, *out* or *in/out*, to describe their preconditions, triggers and played_roles (subprocesses) (Figure 9-26).



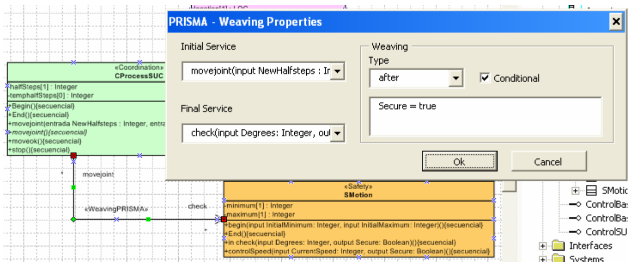
Select Service: Begin(){secuencial}

[✓ In] Out In/Out

Valuation of selected service:

Figure 9-26 Developed Form to specify those specific properties to aspects: kinds of service, preconditions, triggers and subprocesses.

In addition, a shape for representing the weaving relationship is shown in the same stencil that uses Dependency as the base class. This shape also has a specific form, shown in Figure 9-24, which not only allows the specification of both the initial and final services and the kind of weaving relationship (*after*, *before* and *instead*), but also facilitates the description of an obligation semantic if it is needed. In this way, *afterIf*, *beforeIf* or *insteadIf* can be easily specified according to a specific condition (Figure 9-27).



Initial Service: movejoint(input NewHalfsteps : Ir

Final Service: check(input Degrees: Integer, oul

Weaving Type: after

[✓ Conditional]

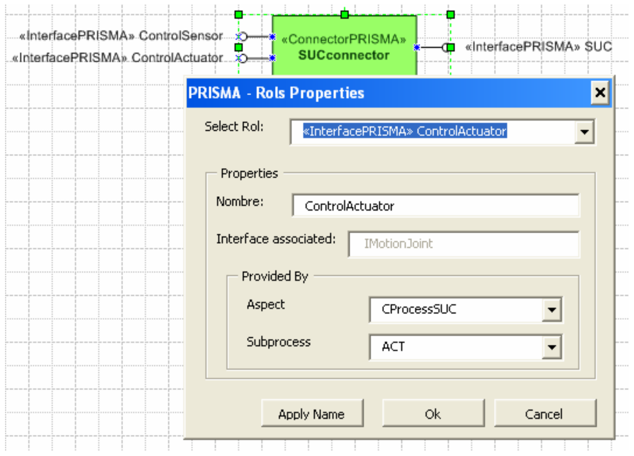
Secure = true

«Coordination»
CProcessSUC
halfSteps[1] : Integer
-temphalfSteps[0] : Integer
+Begin(){secuencial}
+End(){secuencial}
+movejoint(entrada NewHalfsteps : Integer, entra
+movejoint(){secuencial}
+moveok(){secuencial}
+stop(){secuencial}

«Safety»
SMotion
-minimum[1] : Integer
-maximum[1] : Integer
+begin(input InitialMinimum: Integer, input InitialMaximum: Integer)(){secuencial}
+End(){secuencial}
+in check(input Degrees: Integer, output Secure: Boolean)(){secuencial}
+controlSpeed(input CurrentSpeed: Integer, output Secure: Boolean)(){secuencial}

Figure 9-27. MORPHEUS provides support to model weaving relationships.

The *Main Structure PRISMA* stencil (Figure 9-24) works in a similar manner to the one previously described. The main difference is its purpose, because it is intended to model the main elements in PRISMA. Therefore, *component*, *connector* and *system* are visually represented by means of shapes specified on this stencil that are the visual representation of the specified stereotypes having *Component* as the base class. Other modifiable properties have been added to those inherited from the Visio UML template. For instance, a connector needs the definition of its roles by specifying its name, the associated interface, and the pair aspect-subprocess (Figure 9-28) which provides the role with the semantic.


«InterfacePRISMA» ControlSensor
«InterfacePRISMA» ControlActuator
«ConnectorPRISMA» SUCconnector
«InterfacePRISMA» SUC

PRISMA - Rols Properties

Select Rol: «InterfacePRISMA» ControlActuator

Properties

Nombre: ControlActuator

Interface associated: IMotionJoint

Provided By

Aspect: CProcessSUC

Subprocess: ACT

Figure 9-28 Describing roles when a connector is defined

Another main area in the Architecture Environment is the *Model Explorer* (see Figure 9-23 where it is named *Explorador de modelos*) that is reused from the Visio UML template. It provides a tree-view of the PRISMA model being defined to facilitate the navigation through the model, i.e., a hierarchy in which each PRISMA element or diagram is represented by an icon. The Model Explorer has been customized to provide the architect with guidance throughout the process in a similar way to how she/he would have fulfilled the PRISMA AO-ADL. In this sense, two levels can be distinguished:

– *Definition Level* that provides the view of the PRISMA repository where every PRISMA element is defined. For this reason, it is structured in several packages: Aspects, Components, Connectors and Systems. For instance, we can observe in Figure 9-22 the *SMotion aspect* that has been defined by dragging and dropping a Safety Aspect from the *Aspect and Interface* stencil onto the *Aspect* page. In a similar way, *Actuator* and *Sensor* components and the *SUCConnector* connector were defined. It also facilitates the reuse of every element defined on the repository by just dragging it to the locations where it is needed.

– *Configuration Level* which allows the user to access to the instances of the PRISMA model, i.e., the Architectural Model. For instance, in our case study we can see on the Model Explorer where the *Base* system, an instance of the SUC system, appears. This instance has been defined by means of the *Instance* item available on the PRISMA menu.

We would like to point out another functionality that has been added to Visio by means of the Architecture environment. It is related to the translation from the graphical model to the textual PRISMA ADL. A new item, called *Generate code,* has been added to the *PRISMA toolbar*; in such a way that the currently visually defined model is translated into the PRISMA AO-ADL by just clicking on it. An example of the textual specification generated is shown in the Appendix D. This means that the developed compiler, introduced in (Pérez, 2006), will allow the generation of C# code from the textual notation generated.

9.6 CONCLUSIONS

MORPHEUS has been presented in this chapter. It is a tool which assists throughout the whole application of ATRIUM. It is described by means of three different environments in order to provide the user with a better comprehension of the tasks to be performed in each moment.

The Requirements Environment provides support for the description of the ATRIUM Goal Model. One of the main advantages of this environment is that it has been split into two different tools. One of them provides support for describing Requirements Metamodels and the other for its later exploitation. It means that any other approach of RE could be exploited in this environment. This facility has been also exploited to provide support for customizable analysis process. Any defined model can be analysed to determine its satisfiability, conflicts, etc, using those rules that the user describes for that aim. This functionality was provided by means of dynamic compilation techniques that speed up the process.

The Scenarios Environment provides a graphical environment for the description of the ATRIUM Scenario Model. This environment integrates ModelMorf, a tool that supports QVT. By means of its integration in MORPHEUS the ATRIUM Scenario Model can be automatically transformed into the proto-architecture.

The Architectural Environment has been defined to complete the architectural description obtained in the previous environment. Specifically, it provides support to the description of PRISMA, the selected AO-ADL.

These environments has been applied to the description of the case study presented along the different chapters.

The work related to the definition of MORPHEUS has been presented in the following publications:

– E. Navarro, P. Letelier, D. Reolid, I. Ramos, "Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques", Information Systems Development Advances in Theory, Practice, and Education (to be published).

– J. Pérez, E. Navarro, P. Letelier, I. Ramos, "A Modelling Proposal for Aspect-Oriented Software Architectures", Proceedings 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06), Postdam, Germany, March 27th-30th, 2006, pp. 32-41.

– J. Pérez, E. Navarro, P. Letelier, I. Ramos, "Graphical Modelling For Aspect Oriented SA", Proceedings 21st Annual ACM Symposium on Applied Computing (SAC'06) Track on Programming for Separation of Concerns (short paper), Dijon, France, April 23 -27, 2006.

"The difficulties which I meet with in order to realize my existence are precisely what awaken and mobilize my activities, my capacities." — *Jose Ortega y Gasset*

CHAPTER 10

Conclusions and further work

10.1 CONCLUSIONS

We have shown how to address the iterative development of Requirements and Software Architectures during the development of software systems. ATRIUM, a methodology that guides the analyst from an initial set of Requirements to an instantiated Software Architecture, has been presented. It uses the strength provided by the coupling of scenarios and goals systematically to guide through an iterative process. Moreover, it allows the traceability between both artifacts to avoid the lack of consistency.

The definition of the Requirements Metamodel to be used in ATRIUM was performed following an iterative process in cooperation with the UPCT. These iterations meant many changes on the Metamodel and the integration of several approaches. This has motivated that we have followed a metamodeling approach. With this aim, we have defined a metamodel that includes the core set of concepts that corresponds to the essential expressiveness of some of the most popular and/or advanced approaches in requirements engineering. It allows us to adapt and extend a core set of concepts keeping a suitable level of semantics consistence. In addition, we have established a set of guidelines for adapting the metamodel to specific needs, according to the required expressiveness. In this way and according to the project specific needs, it is provided a proper integration as well as scalability from simpler up to other more sophisticated RE techniques. The definition of this core Metamodel also has facilitated the definition of an analyse process that can be customized according to the kinds of artifacts and relationships.

We consider that our proposal constitutes a step forward in achieving a successful application of RE techniques in real-life projects. In addition, we believe that our proposal provides the analyst with an additional advantage:

traceability between different requirements specifications. Because any type of artifact and relationship can be described, it would be possible, for instance, to introduce specifications following a goal-oriented approach and its traceability to a viewpoint approach to analyze the specification from different perspectives and techniques.

The Goal Model is a key artifact used along the guided process of ATRIUM. It has been designed to allow the analyst to reason about design alternatives by using the defined relationships. Additionally, the refinement process compels him/her to focus on a specific view of the system-to-be definition and, therefore, on a partial view of the architectural model, owing to its ability to trace low-level details back to high-level concerns. Moreover, one of its main advantages is it deals jointly with functional and non-functional goals, which improves consistency and maintainability. The Aspect-Oriented approach has been integrated in the definition of the ATRIUM Goal Model. It facilitates that the model can deal with complex and/or large systems whose specification emerges tangled, i.e., same requirements appear over and over along the specification, affecting to other ones. The introduction of the expressiveness for variability management was also compulsory in the description of the proposal. This is because the EFTCoR project exhibits specific needs in terms of product lines that must be specified just from the very beginning of the specification. Another advantage that offers our proposal is the use of the ISO/IEC 9126 as a starting point to establish the possible concerns of the system-to-be.

One of the advantages of the Goal-Oriented approach is its facility for tracing the established requirements to artifacts defined in later stages of development. It has been exploited to establish the traceability to the defined Scenario Model. The main idea behind its exploitation is to provide the analyst with a mechanism to study and analyze the main behaviour of the system-to-be. For this reason, each ATRIUM scenario is going to depict a partial view of the system-to be coping with an operationalization decision. This means that each scenario is traced from an operationalization and, thus, from a set of specific requirements. This facilitates the maintenance of the traceability throughout the lifecycle. In order to facilitate their description, a graphical notation has been defined by extending the Interaction Diagrams of UML 2.0.

It is worth noting that the operationalization decisions consider both functional and non-functional requirement along with their identified crosscutting. For this reason, the notation has been adapted to manage properly this constraint. In this sense, the Aspect-Oriented approach has meant an improved advantage. A notation for the enrichment of scenarios, using this technique, has been introduced. This alternative allows the analyst to introduce lightweight solutions for specific problems. In addition, it provides a systematic way of dealing with early aspects and their traceability to Software Architecture.

Another advantage of the ATRIUM Scenarios is that they provide a transversal view of the system-to-be. This allows the analyst to obtain the overall idea of the behaviour of the system-to-be and its structure. Therefore, the introduction of Architectural Styles, especially DSSAs, is a meaningful advantages because it gives the analyst some templates of how to define the scenario because they describe elements to use, allowed interactions, etc.

The Scenario Model is the input to generate the proto-architecture. It is used as a first draft of the SA for the system-to-be to be refined in a later stage of the software development. It facilitates that the analysis performed by defining the ATRIUM Scenarios Model to obtain an improved comprehension of the system, can be traced to a later stage with an automatic process. This automatic process is applied by a technique of Model Transformation. Specifically, QVT has been chosen as the most proper solution because it satisfies most of the established goals.

By using QVT Relations, a set of transformations rules have been defined that are applicable to the whole set of scenarios. Other challenge faced with this alternative was the introduction of styles during the transformation. In addition, it was shown how constraints specifics for some style can be described introducing some specific relations in a different transformation. The main idea is that different transformations can be described that generate the proto-architecture applying different Architectural Styles. This means that different proto-architectures could be generated using the same set of scenarios but taking into account different Architectural Styles.

It must be also pointed out that this transformation has been defined to provide as flexibility as possible in terms of the Architectural Model used for the generation of the proto-architecture. The main goal was to provide the analyst with facilities to generate it using that Architectural Model he/she considers more appropriate for his/her aims. For this reason, the set of Relations has been catalogued as architectural generative patterns, Architectural Style-oriented transformations and idioms-oriented transformations. This facilitates that the same set of scenarios could be transformed to different proto-architectures by selecting that idioms and/or Architectural-Style-oriented transformations specifics to the desired Architectural Model and/or Architectural Style.

In addition, the use of QVT plays a significant role for traceability top-down and bottom-up. The former is provided because the proto-architecture is generated automatically be establishing the appropriate transformations. The latter can be achieved because QVT Relations derives a Trace Class from each Relation used in order to generate traceability maps. This ability is highly meaningful because a mapping is established between every element in the proto-architecture and its related element/s in the ATRIUM Scenarios Model. It must be taken into account that a proto-architecture is generated from the set of scenario, not a full specification. This means that it should be refined during the next stage of the software development. During this activity, if any change is detected as necessary it could be traced-back to detect which scenario/s should be modified, helping to maintain the models up-to-date.

Finally, it must be pointed out that this thesis is not only a theoretical approach but it has been exploited with the description of a case study. In its development, it was key the collaboration of the UPCT in the context of the DYNAMICA project to improve and refine the results obtained. This case study was developed thanks to the developed tool: MORPHEUS. This tool provides support throughout the whole description of ATRIUM.

10.2 RESULTS OF THE PHD

This thesis has had impact both international as national as can be observed in terms of both publications and conference activities.

10.2.1 Publications

International Journals

- E. Navarro, P. Letelier, J. A. Mocholí, I. Ramos, "A Metamodeling Approach for Requirements Specification", Journal of Computer Information Systems, 46(5): 67-77, Special Issue on Systems Analysis and Design, ed. Keng Siau.

Book Chapters

- E. Navarro, P. Letelier, D. Reolid, I. Ramos, "Configurable Satisfiability Propagation for Goal Models using Dynamic Compilation Techniques", Information Systems Development Advances in Theory, Practice, and Education (to be published).
- P. Letelier, E. Navarro, V. Anaya, "Customizing Traceability in a Software Development Process", Information Systems Development Advances in Theory, Practice, and Education, Vasilecas, O.; Caplinskas, A.; Wojtkowski,

G.; Wojtkowski, W.; Zupancic, J.; Wrycza, S. (Eds.), Springer Science+Business Media, Inc., USA, 2005, pp. 137-148.

International Conferences & Workshops

- E. Navarro, P. Letelier, J. Jaén, I. Ramos, "A generative proposal for proto-architectures exploiting Architectural Styles", First European Conference on Software Architecture (ECSA'07), September 24-26, 2007, Aranjuez (Madrid) – Spain (submitted).
- E. Navarro, P. Letelier, I. Ramos, "Requirements and Scenarios: playing Aspect Oriented Software Architectures", Proceedings Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), Mumbai, India, January 6 - 9 2007 (short paper).
- E. Navarro, P. Sánchez, P. Letelier, J.A. Pastor, I. Ramos, "A Goal-Oriented Approach for Safety Requirements Specification", Proceedings 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06), Postdam, Germany, March 27th-30th, 2006, pp. 319-326.
- J. Pérez, E. Navarro, P. Letelier, I. Ramos,"A Modelling Proposal for Aspect-Oriented Software Architectures", Proceedings 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'06), Postdam, Germany, March 27th-30th, 2006, pp. 32-41.
- J. Pérez, E. Navarro, P. Letelier, I. Ramos, "Graphical Modelling For Aspect Oriented SA", Proceedings 21st Annual ACM Symposium on Applied Computing (SAC'06) Track on Programming for Separation of Concerns (short paper), Dijon, France, April 23 -27, 2006.
- E. Navarro, P. Letelier, I. Ramos, "Integrating Expressiveness of Modern Requirements Modelling Approaches", Proceedings 3rd International Conference on Software Engineering Research, Management & Applications (SERA 2005), Mount Pleasant, Michigan, USA, August 11 - 13, 2005, IEEE Computer Society, ISBN 0-7695-2297-1.
- E. Navarro, P. Letelier, I. Ramos, "Goals and Quality Characteristics: Separating Concerns", Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, collocated to OOPSLA 2004, Monday, October 25, 2004, Vancouver, Canada.
- E. Navarro, P. Letelier, I. Ramos, "UML Visualization for an Aspect and Goal-Oriented Approach", The 5th Aspect-Oriented Modeling Workshop

(AOM'04), collocated to UML 2004 Conference, Monday, October 11, 2004, Lisbon, Portugal.

- J. Jaén, J. H. Canos, E. Navarro, "A Web-Based Coordination Infrastructure for Grid Collective Services", 5th International Conference on Web-Age Information Management (WAIM 2004), Dalian, China, July15 - 17, 2004, Proceedings in Lecture Notes in Computer Science 3129 Springer 2004, ISBN 3-540-21044-X, pp. 449-458.
- E. Navarro, I. Ramos, J. Pérez, "Goals Model Driving Software Architecture", Proceedings 2nd International Conference on Software Engineering Research, Management & Applications (SERA 2004), Los Angeles, California, USA, May 5-17, 2004, ISBN 0-97007769-6, pp. 205-212.
- J. Jaén, E. Navarro, "An Infrastructure to Build Secure Shared Grid Spaces", VI International Conference on Coordination Models and Languages (COORDINATION 2004), Pisa, Italy, February 24-27, 2004, Proceedings in Lecture Notes in Computer Science 2949 Springer 2004, ISBN 3-540-21044-X, pp. 170-182.
- E. Navarro, I. Ramos, J. Pérez Benedí, "Software Requirements for Architectured Systems", Proceedings of 11th IEEE International Requirements Engineering Conference (RE'03)(short paper), Monterey, California, USA, September 8-12, 2003, IEEE Computer Society 2003, ISBN 0-7695-1980-6, pp. 365-366.
- J. Pérez., I. Ramos, J. Jaén, P. Letelier, E. Navarro, "PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures", Proceedings 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003, IEEE Computer Society 2003, ISBN 0-7695-2015-4, pp. 59-66.

National Conferences & Workshops

- Ángel Roche, P. Letelier, E. Navarro, "Validación incremental de Modelos usando escenarios y prototipado automático", XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2006), Sitges, Spain, October 3rd - 6th, 2006.
- E. Navarro, P. Letelier, I. Ramos, "Un Marco de Trabajo para Integrar y Adaptar Múltiples Enfoques para Especificación de Requisitos", Jornadas de trabajo DYNAMICA, Archena, Spain, Novembre 17-18, 2005.

- E. Navarro, P. Sánchez, P. Letelier, J. A. Pastor, I. Ramos, "Sistematizando la Especificación de Requisitos Safety en Aplicaciones Teleoperadas", Proceedings of X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2005) Granada, September 14-16, 2005, Toval, A. Hernández, J. (eds). Thomson Paraninfo, Spain, pp. 35-42, ISBN:84-9732-434-X.
- E. Navarro, P. Letelier, I. Ramos, P. Sánchez, B. Alvarez, "Variabilidad en un marco de requisitos basado en orientación a objetivos", Jornadas de trabajo DYNAMICA, Almagro, Spain, April 21-22, 2005.
- E. Navarro, P. Letelier, I. Ramos, B. Alvarez, "Especificación de requisitos software basada en características de calidad, separación de concerns y orientación a objetivos", IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004), Málaga, Novembre 10-12, 2004.
- E. Navarro, P. Letelier, I. Ramos, B. Alvarez, "Orientación a aspectos y Orientación a objetivos: una propuesta para su integración", Desarrollo de Software Orientado a Aspectos, collocated to IX JISBD'2004, Málaga, Spain, Novembre 9, 2004.
- E. Navarro, P. Letelier and I. Ramos, ATRIUM, Arquitecturas Software a partir de Requisitos - El Modelo de Objetivos", Jornadas de trabajo DYNAMICA, Málaga, Spain, Novembre 11, 2004.
- E. Navarro and I. Ramos, "Requirements and Architecture: a marriage for Quality Assurance", VIII Jornadas de Ingeniería del Software y Bases de Datos, Alicante, Novembre 12-14, 2003, ISBN 84-688-3836-5, pp. 69-78.

10.2.2 Conference Activities

- Organizing Committee, Workshop de Desarrollo de Software Orientado a Aspectos (DSOA'06), collocated to XI JISBD'2006, Sitges, Spain, October, 2006.
- Program Committee, 5th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2006), July 12-14, 2006, Honolulu, Hawaii, Sponsored by the IEEE Computing Society and International Association for Computer & Information Science (ACIS).
- Program Committee, 4th ACIS International Conference on Software Engineering Research, Management & Applications (SERA2006), August 9-11, 2006, Seattle, Washington, USA, Sponsored by the International

Association for Computer & Information Science (ACIS) and Software Engineering and Information Technology Institute (SEITI)

- Additional Reviewer, 5th IEEE International Conference on Quality Software (QSIC 2006), Beijing, China, October 26-28, 2006.
- Additional Reviewer, Encuentro Mexicano de Computación 2006 (ENC2006), San Luis Potosí, September 20 - 22, 2006
- Program Committee, 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA2005), Central Michigan University, Mount. Pleasant, Michigan, USA, August 11 - 13, 2005, Sponsored by the International Association for Computer & Information Science (ACIS)
- Additional Reviewer, 5th IEEE International Conference on Quality Software (QSIC 2005), Melbourne, Australia, September 19–20, 2005.
- Organizing Committee, Workshop de Desarrollo de Software Orientado a Aspectos (DSOA'05), collocated to X JISBD'2005, Granada, Spain, September 14-16, 2005.
- Additional Reviewer, IX Jornadas de Ingeniería del Software y Bases de Datos (JISBD'2004), Málaga, Novembre 10-12, 2004.
- Additional Reviewer, 3rd IEEE International Conference on Quality Software (QSIC 2003), Dallas, Texas, USA, November 6 - 7, 2003.
- Additional Reviewer, VIII Jornadas de Ingeniería del Software y Bases de Datos, (JISBD'2003), Alicante, Novembre 12-14, 2003.

10.3 FURTHER WORK

Considering that ATRIUM has a wide impact along the life-cycle of development, several open issues remains open. We are introducing them in the following. To establish a proper reference for them, they are presented according to the chapter which constitutes their background.

Considering the metamodeling for RE introduced in chapter 5 two topics constitutes our main work. The first of them is related to studying other interesting approaches in Requirements Engineering area in the context of our proposal: *View Points* (Finkelstein et al., 1992), *Problem Frames* (Jackson, 2000) and *Cognitive Mappings* (Siau & Tan, 2005). Based on the results of our case study, we think it will be easy to include the additional required expressiveness.

Our proposal facilitates a deeper analysis of the specification because proper traceabilities could be established, for instance, between conceptual maps described by means of Cognitive Mappings and services of the system-to-be described using goals models. We consider that they are just a first step towards describing an analysis process which could be called concern-oriented, i.e., a process where the rules to be applied depends on the concern that is dealing with. The second work is focused on providing a formal framework for analyzing and checking models. It is necessary to specify artifacts and to provide a precise semantics for the expressiveness provided by the model. The works by Letier and Lamsweerde's (Letier & Lamsweerde, 2002) or Katz and Rashid (Katz & Rashid, 2004) can be useful as a reference to achieve this aim.

In addition, a deep inspection about the associated semantic of weaving relationships remains as a current challenge. Until now, the traditional ones have been defined but those introduced by (Rashid et al., 2003) can suggest new alternatives for our approach. In this sense, we think the equilibrium between the readability/simplicity of the specification and the versatility of the weaving relation should be achieved.

Another topic for further work is related to identification of interaction patterns for several crosscutting concerns in order to manage the possible interference among them. In the developed case studies, we observed that several concerns crosscut another one. The associated semantic of this composition and how the tradeoffs between them has to be faced should be solved in the next future.

In chapter 7, it was presented the use of design patterns to help in the process of describing ATRIUM Scenario Model. Despite the apparent abundance and extensive use of patterns in decision-making, analyst do not always has easy access to the proper ones. Although, they are organized in catalogues, they are usually indexed with just a few mnemonic features that do not always indicate a design's relevance. Therefore, another of our future concerns is how we can encompass with those so well known best practices in software engineering without memorizing all that knowledge. For this reason, we are looking for alternatives to realize the Hollywood Principle ("Don't call us, we'll call you") or the Greyhound Principle ("Leave the driving to us.").

The increased awareness of the importance of an explicit design of the architecture of a software system has not decreased the importance of the components that make up the architecture. It was introduced, in the description of the ATRIUM Scenario Model, the identification of COTS as one of the architectural elements used. For this reason, the incorporation of some technique, such as CARE (Chung et al., 2004) or GBTCM+ (Ayala & Franch, 2006), for selecting components constitute another of our future works.

In chapter 8, it was presented the introduction of Architectural Styles for the generation of the proto-architecture. Incompatibilities between Architectural Styles may appear during their selection. For this reason, the analyst must be provided with techniques that facilitate the necessary trade-offs.

Once the proto-architecture has been defined another main activity should be performed: its evaluation respect the established requirements. For this reason the analysis of the applicability of proposals in this fields, as that presented by (Babar & Gorton, 2004), are quite appropriate to round up this proposal.

KEYWORDS

ACRONYMS

ADL	Architectural Description Language
AO-ADL	Aspect-Oriented Architecture Description Language
AOP	Aspect-Oriented Programming
AORE	Aspect-Oriented Requirements Engineering
AOSA	Aspect-Oriented Software Architecture
AOSD	Aspect Oriented Software Development
APL	Architectural Prescription Language
CBR	Case Based Reasoning
CBSD	Component-Based Software Development
CBSP	Component - System – Bus - Property
CIM	Computation-Independent Model
COTS	Commercial off-the Shelf
DSSA	Domain-Specific Software Architecture
DSA	Dynamic Software Architecture
FIPA	Foundation for Intelligent Agents
FS	Feature Space
GM	Goal Model
KS	Knowledge Sources
IPS	Interaction Pattern Specification
MDA	Model Driven Architecture
MDSD	Model-Driven Software Development
MUC	Mechanism Unit Controller
PIM	Platform-Independent Model
PSM	Platform-Specific Model
RBML	Role-Based Metamodeling Language

RE	Requirements Engineering
RIM	Requirements Interaction Management
RDCU	Robotic Devices Control Unit
RRC	Risk Reduction Categories
RUC	Robotic Unit Controller
SA	Software Architecture
SM	Scenario Model
SPS	Structural Patterns Specification
SPL	Software Product Lines
SoC	Separation of Concerns
SPEM	Software Process Engineering Metamodel
SRD	Software Requirements Document
SRS	Software Requirements Specification
SS	Solution Space
SUC	Simple Unit Controller
UCM	Use Case Map
UPCT	University Polytechnic of Cartagena

REFERENCES

(Abowd et al, 1995) G. Abowd, R. Allen, and D. Garlan. "Using style to understand descriptions of Software Architecture". *ACM Transactions on Software Engineering and Methodology*, 4(4): 319 – 364, October 1995.

(Andrade & Fiadeiro, 2003) L. F. Andrade, J. L. Fiadeiro, "Architecture Based Evolution of Software Systems", *Formal Methods for Software Architectures*, Springer Verlang, M. Bernardo and P. Inverardi (eds), LNCS 2804, September 2003

(Alencar et al., 2006) F. Alencar, A. Moreira, J. Araujo, J. Castro, C. Silva, J. Mylopoulos, "Using Aspects to Simplify iModels", Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), Minneapolis, MN, USA, September 11-15, 2006, pp. 139-148.

(ANSI, 2000) ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems.

(ANSI/RIA, 1999) ANSI/RIA R15.06-1999. American National Standard for Industrial Robots and Robot Systems, Safety Requirements. Robotic Industries Association, 1999

(Aldawud et al., 2003) O. Aldawud, T. Elrad, and A. Bader, "UML profile for Aspect Oriented Software Development". Aspect-Oriented UML Workshop, Collocated to the Aspect Oriented Software Development Conference, (Boston, USA March 18, 2003).

(Allen & Garlan, 1994) R. Allen and D. Garlan, *Formalizing architectural connection*, 16th International Conference on Software Engineering, May 1994.

(Ambriola & Gervasi, 1997) V. Ambriola and V. Gervasi "Processing natural language requirements," Proceedings of 12th International Conference on Automated Software Engineering, pages 36-45, Los Alamitos, November 1997. IEEE Computer Society Press.

(Antón et al., 2001) A.I. Antón, R.A. Carter, A. Dagnino, J.H. Dempster and D.F. Siege, Deriving Goals from a Use Case Based Requirements Specification *Requirements Engineering Journal*, Springer-Verlag, 6:63-73, May 2001.

(Antón & Potts, 1998) A.I. Anton and C. Potts, "The Use of Goals to Surface Requirements for Evolving Systems", Proceedings of the 20th International Conference on Software Engineering (ICSE-98), Kyoto, April 1998.

(Antón, 1997) A. I. Antón, *Goal Identification and Refinement in the Specification of Information Systems*, Ph.D. Thesis, Georgia Institute of Technology, June 1997.

(Antón, 1996) A. I. Antón, "Goal-Based Requirements Analysis", Proceedings 2nd International Conference on Requirements Engineering, Colorado Springs, CO April 15 - 18, 1996.

(AOSD) AOSD, http://www.aosd.net

(Araujo et al., 2004) J. Araujo, J. Whittle, D. Kim, "Modeling and Composing Scenario-Based Requirements with Aspects" Proceedings 12th IEEE International Requirements Engineering Conference, 6-10 September 2004, Kyoto, Japan, pp. 58-67.

(Araujo & Moreira, 2003) J. Araujo and A. Moreira, "An Aspectual Use Case Driven Approach," Proceedings VIII Jornadas de Ingeniería de Software y Bases de Datos (JISBD), Alicante, Spain, 2003.

(Argo, 2005) ArgoUML, http://argouml.tigris.org/

(Atkinson & Kuhne, 2003) C. Atkinson, T. Kuhne, "Aspect-Oriented Development with Stratified Frameworks," *IEEE Software*, 20(1):81-89, Jan/Feb, 2003.

(Avgeriou & Zdun, 2005) P. Avgeriou, U. Zdun, "Architectural Patterns Revisited – A Pattern Language", Proceedings of 10th European Conference on Pattern Languages of Programs, (EuroPlop 2005), Irsee, Germany, July 2005.

(Ayala & Franch, 2006) C. P. Ayala, X. Franch, "A Goal-Oriented Strategy for Supporting Commercial Off-the-Shelf Components Selection," Proceedings 9th International Conference on Software Reuse, ICSR 2006, Turin, Italy, June 12-15, 2006, LNCS 4039 Springer 2006, pp. 1-15

(Babar & Gorton, 2004) M. A. Babar, I. Gorton, "Comparison of Scenario-Based Software Architecture Evaluation Methods," 11th Asia-Pacific Software Engineering Conference (APSEC'04), 2004, pp. 600-607.

(Balzer et al., 1982) R. M. Balzer, N. M. Goldman, D. S. Wile, "Operational specification as the basis for rapid prototyping," *ACM SIGSOFT Software Engineering Notes* 7(5):3-16, December 1982.

(Baniassad et al., 2006) E. Baniassad, P. C. Clements, J. Araújo, A. Moreira, A. Rashid, B. Tekinerdogan, "Discovering Early Aspects", *IEEE Software* 23(1): 61-70, January/February 2006.

(Baniassad & Clarke, 2004) E. Baniassad, S. Clarke, "Theme: An Approach for Aspect-Oriented Analysis and Design". 26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, UK. IEEE Computer Society 2004, pp. 158-167.

(Baresi, 2004) L. Baresi, R. Heckel, S. Thöne, and D. Varró, "Style-Based Refinement of Dynamic Software Architectures", 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04), June 12 - 15, Oslo, Norway, pp. 155.

(Bass et al., 2003) L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*, 2nd Edition, Addison Wesley Professional, 2003.

(Bass et al., 2001) L. Bass, M. Klein, F. Bachmann, "Quality Attribute Design Primitives and the Attribute Driven Design Method", Proceedings of 4th International

Workshop Software Product Family Engineering (PFE 2001), Bilbao, Spain, October 3-5, 2001. LNCS 2290, pp. 169-186.

(Baskerville & Wood-Harper, 1996) R.L. Baskerville, A.T. Wood-Harper, "A Critical Perspective on Action Research as a Method for Information Systems Research," *Journal of Information Technology*, 11: 235-246, 1996.

(Bergmans & Aksit, 2001) L. Bergmans, M. Aksit, "Composing Multiple Concerns Using Composition Filters," *Communications of the ACM*, 44(10):51-57, October 2001.

(Berry et al., 2003) D. Berry, R. Kazman, R. Wieringa (eds), Proceedings of Second International Workshop From Software Requirements to Architectures (STRAW'03), Portland, USA, 2003.

(Bosch, 2000) J. Bosch, Design and Use of Software Architectures Adopting and Evolving a Product-Line Approach, ISBN 0-201-67494-7, Addison-Wesley, May 2000.

(Bosch & Molin, 1999) J. Bosch, P. Molin, "Software Architecture design: evaluation and transformation", Proceedings IEEE Conference and Workshop on Engineering of Computer-Based Systems, 1999 (ECBS '99), Nashville, TN, USA, 7-12, March 1999.

(Brandozzi & Perry, 2001) M. Brandozzi, D.E. Perry, "Transforming Goal-Oriented Requirement Specifications into Architecture Prescriptions" Workshop "From Software Requirements to Architectures" (STRAW'01) collocated to ICSE 2001, May 14, 2001, Toronto, Canada.

(Brito & Moreira, 2003) I. Brito, A. Moreira, "Towards a composition process for aspect-oriented requirements", Early Aspects 2003: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, collocated to 2nd Aspect-Oriented Software Development Conference (AOSD), March 17, 2003 - Boston, USA.

(Brichau & Haupt, 2005) J. Brichau and M. Haupt (Eds), "Survey of Aspect-oriented Languages and Execution Models", Technical Report AOSD-Europe-VUB-01 deliverable D12, May 17, 2005.

(Bruin & Vliet, 2003) H. de Bruin and H. van Vliet, "Quality-Driven Software Architecture Composition", *Journal of Systems and Software*, 66(3): 269-284, 2003.

(Bühne et al., 2003) Bühne, S., Halmans, G. and Pohl, K. "Modeling Dependencies between Variation Points in Use Case Diagrams", 9th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'03). Collocated to CAiSE'03, 16 -17 June 2003, Klagenfurt/Velden, Austria, 59-70.

(Buhr & Casselman, 1996) R. Buhr, R. Casselman, *Use CASE Maps for Object-Oriented Systems*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

(Buschmann et al., 1996), F. Buschmann, R. Meunier, H.Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System Of Patterns*. Addison-Wiley, 1996.

(Castro et al., 2002) J. Castro, M. Kolp, J. Mylopoulos, "Towards Requirements-Driven Software Development Methodology: The Tropos Project," *Information Systems*, June 2002.

(Castro & Kramer, 2001) J. Castro J. Kramer (eds): Proceedings of First International Workshop From Software Requirements to Architectures (STRAW'01), 2001.

(Chitchyan et al., 2005) R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto, J. Bakker, B. Tekinerdogan, S. Clarke, A. Jackson, "Survey of Analysis and Design Approaches", AOSD-Europe-ULANC-9, AOSD-Europe Project, May 18, 2005.

(Chung et al., 2004) L. Chung, K. Cooper, S. Courtney, "COTSAware Requirements Engineering and Software Architecting". Proceedings of the International Conference on Software Engineering Research and Practice, SERP '04, June 21-24, 2004, Las Vegas, Nevada, USA, Volume 1. CSREA Press 2004, pp. 57-63.

(Chung et al., 2000) L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishing, 2000.

(Clements & Northrop, 2001) P. Clements, and L. Northrop, *Software Product Lines - Practices and Patterns*, Pearson Education, Addison-Wesley, 2001.

(Coad et al., 1999) P. Coad, J. D. , Luca, and E. Lefebvre, "Java Monitoring in Color with UML". Chapter 6: Feature Driven Development , Prentice Hall 1999

(Cockburn, 2000) A. Cockburn, *Writing Effective Use Cases*, Addison Wesley Professional, 2000, p. 304.

(Cooper et al., 2005) K. Cooper, L. Dai, Y. Deng: "Performance modeling and analysis of Software Architectures: An aspect-oriented UML based approach. Science of Computer Programming, 57(1): 89-108, 2005.

(Cuesta et al., 2005) C. Cuesta, M. P. Romay, P. de la Fuente, M. Barrio-Solórzano, "Architectural Aspects for Architectural Aspects", European Workshop on Software Architecture(EWSA), Pisa, June 2005, Springer LNCS vol n.3527 2005, pp. 247-262.

(Cuesta, 2002) C.E. Cuesta, *Dynamic Software Architecture based on Reflection*. Phd Dissertation, Department of Computer Science, University of Valladolid, 2002. (In Spanish)

(Czarnecki & Helsen, 2006) K. Czarnecki, S. Helsen: Classification of Model Transformation Approaches, *IBM Systems Journal*, 45(3) 2006, http://researchweb.watson.ibm.com/journal/sj/453/czarnecki.html. Updated Version from Proceedings of the 2nd Workshop on Generative Programming

Generative Techniques in the context of Model Driven Architecture, collocated to OOPSLA’03, Monday 27, October 2003.

(Czarnecki & Antkiewicz, 2005) K. Czarnecki and M. Antkiewicz, “Mapping Features to Models: A Template Approach Based on Superimposed Variants,” Proceedings of the 4th International Conference on Generative Programming and Component Engineering, Tallinn, Estonia (2005), pp. 422–437.

(Dardenne et al., 1993) A. Dardenne, A. van Lamsweerde, and S. Fickas: “Goal-directed Requirements Acquisition”. *Science of Computer Programming*, 20(1-2): 3-50, 1993.

(Douglass, 2003) B. P. Douglass, Real-Time Design Patterns. Robust Scalable Architecture for Real-Time Systems. Reading, Addison-Wesley. 2003.

(Dounce & Le Botlan, 2005) R. Douence, D. Le Botlan D, “Towards a Taxonomy of AOP Semantics”. Technical Report AOSDEurope, Milestone M8.1, July 7 2005, http://wwwdgeinew.insa-toulouse.fr/~lebotlan/papers/dl05.pdf.

(Duran, 2000) A. Durán, *Un Entorno Metodológico de Ingeniería de Requisitos para Sistemas de Información*, Tesis Doctoral del Dpto. de Lenguajes y Sistemas de Información de la Universidad de Sevilla, septiembre 2000.

(Eden & Kazman, 2003) A. H. Eden, R. Kazman, “Architecture, Design, Implementation”, Proceedings of 25th International Conference on Software Engineering, May 03 - 10, 2003, Portland, Oregon

(EFTCOR, 2003) EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal. European Project within the 5th Framework Program (GROWTH G3RD-CT-00794), 2003.

(Egyed et al., 2000) A. Egyed, N. Medvidovic, and C. Gacek, “A Component-Based Perspective of Software Mismatch Detection and Resolution,” *IEE Software Engineering*, 147(6), December 2000, pp. 225-236.

(Elrad et al., 2001a) T. Elrad, R. E. Filman, A. Bader, “Aspect-oriented programming: Introduction”, *Communications of the ACM*, 44(10), October 2001 pp. 29 - 32

(Elrad et al., 2001b) T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher, “Discussing aspects of AOP”, *Communications of the ACM*, 44(10): 29 – 32, October, 2001.

(Ferrari & Madhavji, 2007) R. Ferrari and N. H. Madhavji, “The Impact of Requirements Knowledge and Experience on Software Architecting: An Empirical Study”, Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'07), Mumbai, India, January 6 - 9 2007.

(Fickas & Nagarajan, 1988), S. Fickas, P. Nagarajan, “Critiquing Software Specifications: a knowledge based approach”, *IEEE Software*, 5(6), 1988.

(Fiege, 2005) L. Fiege, *Visibility in Event-Based Systems*, PhD Dissertation, Department of Computer Science, Darmstadt University of Technology, 2005.

(Finkelstein et al., 1992) A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein and M. Goedicke, "Viewpoints: a framework for integrating multiple perspectives in system development," *International Journal of Software Engineering and Knowledge Engineering*, 2: 31--57, 1992.

(Filho et al., 2004) I. M. Filho, T. C. de Oliveira, and C. J. P. de Lucena, "A framework instantiation approach based on the Features Model", *Journal of Systems and Software*, 73(2), October 2004, Pages 333-349.

(France et al., 2004) R. B. France, D.-K. Kim, S. Ghosh, E. Song, "A UML-Based Pattern Specification Technique," *IEEE Transactions on Software Engineering*, 30(3): 193-206, Mar., 2004.

(Fuxman et al., 2001) A. Fuxman, P. Giorgini, M. Kolp, J. Mylopoulos, "Information systems as social structures", Proceedings 2nd Int. Conference on Formal Ontologies for Information Systems (FOIS'01), Ogunquit, USA, October 2001.

(Gamma et al., 1995) E. Gamma, R. Helm, R. Johnson, J.Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison Wesley, 1995.

(Garlan, 2001) D. Garlan, "Software Architecture", *Wiley Encyclopedia of Software Engineering*, J. Marciniak (Ed.), John Wiley & Sons, 2001.

(Garlan, 2000) D. Garlan, "Software Architecture: a Roadmap,", Proceeding of the International Conference on Software Engineering (ICSE), The Future of Software Engineering Track, Limerick, Ireland, June 4-11, 2000, ACM, pp. 91-101.

(Garlan & Perry, 1995) D. Garlan and D. Perry, "Introduction to the Special Issue on Software Architecture," *IEEE Transactions on Software Engineering*, 21(4): 269-274, April 1995.

(Garlan et al., 1994) D. Garlan, R. Allen and J. Ockerbloom. "Exploiting style in architectural design environments". Proceedings 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'94), December 1994, ACM Press, pp. 170-185.

(Garlan & Shaw, 1993) D. Garlan and M. Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola, G. Tortora (eds), 1993.

(Gerber et al., 2002) A. Gerber, M. Lawley, K. Raymond, J. Steel and A. Wood, "Transformation: The Missing Link of MDA". First International Conference Graph Transformation, (ICGT 2002), Barcelona, Spain, October 7-12, 2002, LNCS 2505 Springer 2002, ISBN 3-540-44310-X 90-105.

(Giorgini et al., 2003) P. Giorgini, E. Nicchiarelli, J. Mylopoulous, and R. Sebastiani. Formal reasoning techniques for goal models. Journal of Data Semantics, 1, 2003.

(GOLD, 2005) GOLD Parsing System, http://www.devincook.com/GOLParser/index.htm, 2005.

(GOYA, 1999) Robot escalador para la limpieza de cascos de buques, respetuoso con el medio ambiente (GOYA), FEDER TAP IFD97-0823 (1999-2001).

(Gonzales-Baixauli et al., 2004) B. Gonzales-Baixauli, J. C. Sampaio, J. Mylopoulos, "Visual Variability Analysis with Goal Models", Proceedings of the International Conference on Requirements Engineering, Los Alamitos, California: IEEE Computer Society Press, 2004, pp. 198-207

(Gotel, 1994) O. Gotel and A. W. Finkelstein, "An analysis of the requirements traceability problem," Proceedings of the International Conference on Requirements Engineering, pages 94--102, Colorado Springs, Colorado, 1994.

(Gregoriades, 2005) A. Gregoriades and A. Sutcliffe, "Scenario-Based Assessment of Nonfunctional Requirements," *IEEE Transactions on Software Engineering*, 31(5), (2005), 392-409.

(Gunter et al., 2000) C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A reference model for requirements and specifications," *IEEE Software*, 17(3), May/June 2000.

(Grünbacher et al., 2001) P. Grünbacher, A. Egyed and N. Medvidovic: "Reconciling Software Requirements and Architectures: The CBSP Approach". Proceedings 5th IEEE Int. Symp. RE, 27-31 August 2001, Toronto, Canada, pp. 202-211.

(Grundy, 1999) J. Grundy, "Aspect-Oriented Requirements Engineering for Component-Based Software Systems", Proceedings IEEE International Symposium on Requirements Engineering, June 07 - 11, 1999, Limerick, Ireland.

(Gurp et al., 2001) J. van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines", Proceedings of the Working IEEE/IFIP Conference on Software Architecture, IEEE Computer Society Press, 2001, pp 45—54.

(Hall et al., 2002) J. Hall, M. Jackson, R. Laney, B. Nuseibeh, and L. Rapanotti, "Relating Software Requirements and Architectures using Problem Frames", Proceedings of IEEE International Requirements Engineering Conference (RE'02) , Essen, Germany, 9-13 September 2002.

(Hallal et al., 2001) H. Hallal, A. Petrenko, A. Ulrich, and S. Boroday, "Using SDL Tools to Test Properties of Distributed Systems", Proceedings of Formal Approaches to Testing of Software (FATES'01), A Satellite Workshop of CONCUR'01, Aalborg, Denmark, August 25, 2001.

(Halmans & Pohl, 2003) G. Halmans, K. Pohl, "Communicating the Variability of a Software-Product Family to Customers," *Software and Systems Modeling*, 2(1): 15-36, March 2003.

(Hammond et al., 2001) J. Hammond, R. Rawlings, A. Hall: "Will It Work?," Proceedings of Fifth IEEE International Symposium on Requirements Engineering (RE '01), Toronto, Canada August, 2001, pp. 27 - 31.

(Hansen et al., 1998) K.M. Hansen, A.P. Ravn, V. Stavridou, "From Safety analysis to software requirements," *IEEE Transactions on Software Engineering*, 24 (7):573-584, 1998.

(Harrison, 2003) N. Harrison N, "Using the CodeDOM". *O'Reilly Network*, February 3rd 2003, http://www.ondotnet.com/pub/a/dotnet/2003/02/03/codedom.html

(Heitmeyer et al., 1996) C. L. Heitmeyer, R. D. Jeffords, B. G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Transactions on Software Engineering Methodology* 5(3): 231-261, July 1996

(Henninger, 2003) S. Henninger, "Tool Support for Experience-Based Software Development Methodologies," *Advances in Computers* 59: 29-82, 2003.

(HyperJ, 2000) HyperJ, http://www.alphaworks.ibm.com/tech/hyperj.

(Hui et al., 2003) B. Hui, S., Liaskos, J., Mylopoulos, "Requirements Analysis for Customizable Software Goals-Skills-Preferences Framework", Proceedings 11th IEEE International Requirements Engineering Conference, Monterey Bay, California, USA: September 08 – 12, 2003, pp. 117-126.

(Hursch & Lopes, 1995) W. Hürsch and C. Lopes, "Separation of Concerns", Technical report NU-CCS-95-03, the College of Computer Science, Northeastern University, 1995

(IEEE, 2000) IEEE Recommended Practices for Architectural Description of Software-Intensive Systems. IEEE Std 1471-2000, Software Engineering Standards Committee of the IEEE Computer Society, 21 September 2000.

(IEEE, 1998) IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications, In Volume 4: Resource and Technique Standards, The Institute of Electrical and Electronics Engineers, Inc. IEEE Software Engineering Standards Collection.

(ISO/IEC 9126) ISO/IEC Standard 9126-1 Software Engineering- Product Quality-Part1: Quality Model, ISO Copyright Office, Geneva, June 2001.

(JAMDA, 2006) JAMDA, Java Model Driven Architecture 0.2, http://sourceforge.net/projects/jamda.

(Jackson, 2000) M. Jackson, Problem Frames: *Analyzing and Structuring Software Development Problems*, Addison-Wesley Pub, 2000.

(Jacobson, 2003) I. Jacobson and P.-W. Ng, *Aspect-Oriented Software Development with Use Cases,* Addison Wesley Professional, 2005.

(Johnson & Russo, 1991) R. E. Johnson and V. F. Russo, "Reusing Object-Oriented Designs" University of Illinois technical report UIUCDCS 91-1696, 1991.

(Kaiya et al., 2002) H. Kaiya, H. Horai, M. Saeki, "AGORA: attributed goal-oriented requirements analysis method," Proceedings of the IEEE International Conference on Requirements Engineering, Essen, Germany, 2002, pp. 13- 22.

(Kande, 2003) M. M. Kande, *A concern-oriented approach to Software Architecture*. PhD. Thesis, Lausanne, Switzerland: Swiss Federal Institute of Technology (EPFL), 2003.

(Kang et al., 1990) K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. "Feature-Oriented Domain Analysis (FODA) Feasibility Study". Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

(Katara & Katz, 2003) M. Katara, S. Katz, "Architectural Views of Aspects". The 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), Boston, Massachusetts, USA, 2003.

(Katz & Rashid, 2004) S. Katz, and A. Rashid, "From Aspectual Requirements to Proof Obligations for Aspect-Oriented, Systems", Proceedings of 12th IEEE International Requirements Engineering Conference, Kyoto, Japan: September 06 - 10, 2004, pp. 48-57.

(Kazman et al., 2000) R. Kazman, M. Klein, and P. Clements. ATAM: Method for architecture evaluation. Technical report, CMU/SEI-2000.

(Kazman et al., 1994) R. Kazman, L. J. Bass, M. Webb, G. D. Abowd, "SAAM: A Method for Analyzing the Properties of Software Architectures", Proceedings of International Conference on Software Engineering, Sorrento, Italy, May 1994, 1994: 81-90.

(Kavakli & Loucopoulos, 2005) E. Kavakli, P. Loucopoulos, "Goal Modeling in Requirements Engineering: Analysis and Critique of Current Methods," *Information Modeling Methods and Methodologies*, John Krogstie, Terry Halpin and Keng Siau (eds), IDEA Group, pp 102 – 124, 2005.

(Kelly et al., 1996) S. Kelly, K. Lyytinen, M. Rossi: "METAEDIT+ A fully configurable Multi-User and Multi-tool CASE and CAME Environment". Proceedings of 8th International Conference on Advances Information System Engineering, LNCS1080, Springer-Verlag, 1996, pp. 1-21.

(Kiczales et al., 1997) G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming". Proceedings European Conference on Object-Oriented Programming, Finland. Springer-Verlag LNCS 1241. June 1997.

(Kickzales et al., 2001) G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, "An Overview of AspectJ", Proceedings 15th European Conference Object-Oriented Programming (ECOOP 2001), Budapest, Hungary, June 18-22, 2001, LNCS 2072, pp. 327-353.

(Kim et al., 2004) D.K. Kim, R. B. France, S. Ghosh. "A UML-based language for specifying domain-specific patterns," *Journal of Visual Language and Computing* 15(3-4): 265-289, 2004.

(Kock & Lau, 2001) N. Kock and F. Lau, "Information systems action research: serving two demanding masters," *Information Technology & People*, 14(1), Mar 2001.

(Kotonya & Sommerville, 1996) G. Kotonya, and I. Sommerville, "Requirements Engineering with Viewpoints," *Software Engineering Journal*, 11(1):5-18, 1996.

(Lamsweerde, 2004) A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice", Proceedings of 12th IEEE International Requirements Engineering Conference, Kyoto, Japan 2004, pp. 4-7.

(Lamsweerde, 2003)A. van Lamsweerde, "From System Goals to Software Architecture", *Formal Methods for Software Architecture*, M. Bernardo, P. Inverardi (eds), LNCS 2804, Springer-Verlag, 2003, pp. 25-43

(Lamsweerde, 2001a) A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour", Proceedings 5th IEEE International Symposium on RE, Toronto, August, 2001, pp. 249-263.

(Lamsweerde, 2001b) A. van Lamsweerde, "Building Formal Requirements Models for Reliable Software", Invited Paper for 6th International Conference on Reliable Software Technologies, Ada-Europe 2001, LNCS. 2043, Leuven, May 14-18, 2001.

(Lamsweerde, 2000) A. van Lamsweerde, "Formal Specification: a Roadmap", Proceedings of the International Conference on Software Engineering - The Future of Software Engineering Track, A. Finkelstein (ed.), ACM Press, (2000).

(Lamsweerde & Letier, 2000) A. van Lamsweerde and E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Transactions on Software Engineering* 26 (10): 978-1005, Special Issue on Exception Handling, October 2000.

(Lamsweerde et al., 1998a) A. van Lamsweerde, R. Darimont, E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Transactions on Software Engineering*, 24(1): 908-926, January 1998.

(Lamsweerde et al., 1998b) A. van Lamsweerde, L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Transactions on Software Engineering*, Special Issue on Scenario Management, 24(12): 1089-1114, December 1998.

(Lamsweerde et al., 1995) A. van Lamsweerde, R. Darimont and Ph. Massonet, "Goal-Directed Elaboration of Requirements for a Meting Scheduler: Problems and Lessons Learnt". Proceedings Second International Conference on Requirements Engineering (RE'95), York, UK, IEEE Computer Society Press, March 1995, pp. 194-203.

(LARLASC, 2002) Large Area Laser Surface Cleaning (LARLASC), EUREKA E! 2732 EULASNET (2002-2004).

(Lauder and Kent, 1998) A. Lauder and S. Kent, "Precise Visual Specification of Design Patterns", 12th European Conference Object-Oriented Programming (ECCOP'98), Brussels, Belgium, July 20-24, 1998, LNCS 1445, ISBN 3-540-64737-6.

(Lauesen, 2006) S. Lauesen "COTS tenders and integration requirements". *Requirements Engineering*, 11(2): 111-122, April 2006

(Lauesen, 2003) S. Lauesen "Task Descriptions as Functional Requirements". *IEEE Software 20*(2): 58-65, 2003.

(Leake, 1996) D. B. Leake, "CBR in Context: The Present and Future", *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, D. B. Leake (ed.), AAAI Press/MIT Press, 1996.

(Lehman, 1980) M. M. Lehman: On understanding Laws, Evolution and Conversation in the large-Program Life Cycle. *The Journal of Systems and Software* 1: 213-221(1980).

(Leite et al., 2000) J.C. S.P. Leite, G. Hadad, J. Doorn, G. Kaplan, "A Scenario Construction Process", *Requirements Engineering Journal*, 5(1): 38-61, (2000).

(Lemos & Saeed, 1995) R. Lemos, A. T. Saeed, "Analyzing Safety Requirements for Process-Control Systems", *IEEE Software*, 12(3), 42-53, May 1995.

(Letelier et al., 1998) P. Letelier, P. Sánchez, I. Ramos, O. Pastor. *OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objeto*, Servicio de Publicaciones Universidad Politécnica de Valencia, SPUPV -98.4011, ISBN 84-7721-663-0, 1998.

(Letier & Lamsweerde, 2004) E. Letier and A. van Lamsweerde, "Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering," Proceedings of FSE'04, 12th ACM International Symposium on the Foundations of Software Engineering, Newport Beach (CA), November 2004, pp. 53-62.

(Letier & Lamsweerde, 2002) E. Letier and A. van Lamsweerde, "Deriving Operational Software Specifications from System Goals", Proceedings 10th ACM S1GSOFT Symposium on the Foundations of Software Engineering (FSE'10), Charleston, November 2002, pp. 119-128.

(Letier & Lamsweerde, 2002) E. Letier and A. van Lamsweerde, "High Assurance Requires Goal Orientation," Proceedings International Workshop on Requirements for High Assurance Systems, Essen, Sept. 2002.

(Letier, 2001) E. Letier, *Reasoning about Agents in Goal-Oriented Requirements Engineering*, Ph.D. Thesis, University of Louvain, May 2001.

(Leue & Rezai, 1998) S. Leue, M. Rezai, "Synthesizing Software Architecture Descriptions from Message Sequence Chart Specifications", Proceedings of 13th

IEEE International Conference on Automated Software Engineering (ASE'98), Honolulu, Hawaii, USA, 13-16 October 1998, pp. 192-195.

(Leveson, 1995) N. Leveson. Safeware, *System Safety & Computers*, Addison-Wesley, 1995.

(Liang et al., 2006) H. Liang, J. Dingel, Z. Diskin, "A comparative survey of scenario-based to state-based model synthesis approaches", Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, collocated to ICSE'06, Shanghai, China, May 20-28, 2006, pp. 5 – 12.

(Lieberherr et al., 1999) K. Lieberherr, D. Lorenz D., M. Mezini, "Programming with Aspectual Components," Technical Report NU-CCS-99-01, Northeastern University, Boston, Massachusetts, March 1999, pp. 1-27.

(Liu & Yu, 2004) L. Liu, E. Yu, "Designing Information Systems in Social Context: A Goal and Scenario Modelling Approach", *Information Systems*, Special issue: The 14th international conference on advanced information systems engineering (CAiSE*02), 29(2): 187 – 203, April 2004.

(Louridas & Loucopoulos, 2000) P. Louridas, and P. Loucopoulos, "A generic model for reflective design," *ACM Transactions on Software Engineering*, 9(2, 2000), 199-237.

(Luckham & Vera, 1995) D. Luckham, J. Vera: An Event-Based Architecture Definition Language, *IEEE Transactions on Software Engineering*, 21(9): 717-734, Sep. 1995.

(Magee et al., 1995) J. Magee, N. Dulay, S. Eisenbach and J. Kramer, "Specifying distributed Software Architectures", Proceedings of the Fifth European Software Engineering Conference, ESEC'95, September 1995.

(Maiden, 1998) N.A.M. Maiden "CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements," *Automated Software Engineering*, 5:419-446, 1998.

(Maiden, & Sutcliffe, 1992) N. A. M. Maiden, & A. G. Sutcliffe, Exploiting Reusable Specifications Through Analogy. *Communications of the ACM*, 34(5): 55-64, 1992.

(Manna & Pnueli, 1992) Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

(Massonet & Lamsweerde 1997) P. Massonet, A. Van Lamsweerde, "Analogical Reuse of Requirements Frameworks", 3rd IEEE International Symposium on Requirements Engineering (RE'97), January 5-8, 1997, Annapolis, MD, USA. IEEE Computer Society, pp. 26-37.

(McDirmid et al., 2001) S. McDirmid, M. Flatt, and W. C. Hsieh. "Jiazzi: New-age components for old-fashioned Java," Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, October 14-18, 2001, Tampa, Florida, USA, pp. 211 – 222.

(Medvidovic, 1996) N. Medvidovic, P. Oreizy, J. E. Robbins and R. N. Taylor, "Using object-oriented typing to support architectural design in the C2 style," Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering (SIGSOFT'96), ACM Press. Oct 1996, pp. 24-32.

(Medvidovic & Taylor, 1997) N. Medvidovic and R. N. Taylor, "A framework for classifying and comparing architecture description languages," Proceedings of the 6th European Conference collocated to the 5th ACM SIGSOFT Symposium on Software engineering, , Zurich, Switzerland, Sept. 22-25 1997, pp. 60—76.

(Mehta et al., 2000) N. R. Mehta, N. Medvidovic, S. Phadke, "Towards a taxonomy of software connectors", Proceedings of the 22nd International Conference on on Software Engineering (ICSE 2000), June 4-11, 2000, Limerick Ireland, pp. 178-187.

(Mellor et al., 2004) S. J. Mellor, K. Scott, A. Uhl, and D. Weise. *MDA Distilled.* Addison-Wesley, 2004.

(Mens et al., 2005) T. Mens, P. Van Gorpa, D. Varróa, and G. Karsaia, "Applying a Model Transformation Taxonomy to Graph Transformation Technology," *Electronic Notes in Theoretical Computer Science*, 152(27) : 143-159, March 2006.

(Mettala & Graham, 1992) E. Mettala, and M. Graham, "The Domain-Specific Software Architecture Program". Special Report of the Carnegie Mellon University Software Engineering Institute, CMU/SEI-92-SR-9. 1992.

(Mezini & Ostermann, 2003) M. Mezini, K. Ostermann, "Conquering Aspects with Caesar". International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, Boston, Massachusetts, USA, March, 2003, pp. 90-100.

(Miller & Madhavji, 2001) J. A. Miller, N. H. Madhavji, "The Architecture-Requirements Interaction," Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA'07), January 6-9, 2007, Mumbai, India.

(ModelMorf, 2007) http://www.tcs-trddc.com/ModelMorf/index.htm, 2007.

(Monroe et al., 1997) R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, "Architecture Styles, Design Patterns, and Objects", *IEEE Software*, 14(1):43-52, January/February 1997.

(Moreira et al., 2005) A. Moreira, J. Araújo, A. Rashid, "A Concern-Oriented Requirements Engineering Model", Proceedings 17th International Conference Advanced Information Systems Engineering, (CAiSE 2005), Porto, Portugal, June 13-17, LNCS 3520, pp 293-308.

(Moreira et al., 2002) A. Moreira, J. Araújo, I. Brito, "Crosscutting quality attributes for requirements engineering", Proceedings 14th International Conference on Software Engineering and Knowledge Engineering (SEKE02), Ischia, Italy, 2002, pp. 167 – 174.

(Moody, 2000) D. L. Moody, "Building links between IS research and professional practice: improving the relevance and impact of IS research", Proceedings of the 21st International Conference on Information systems, Brisbane, Queensland, Australia, 2000, pp. 351-360.

(Munson, et al 2006) J. C. Munson, A. P. Nikora and J. S. Sherif, "Software faults: A quantifiable definition", *Advances in Engineering Software*, 37(5):327-333, May 2006.

(Navasa et al., 2005) A. Navasa, M. A. Pérez, J. M. Murillo, "Aspect Modelling at Architecture Design," 2nd European Workshop on Software Architecture (EWSA), Lecture Notes on Computer Science, Springer Verlang, LNCS 3527, Pisa, Italy, June, 2005, pp. 41-58.

(Newell & Simon, 1963) A. Newell and H. Simon, GPS, "A Program that Simulates Human Thought." *Computers and Thought*, E. A. Feigenbaum and J. Feldman, R. Oldenbourg(eds.), Mac Graw Hill., 1963.

(Niemela et al., 2005) E. Niemela, J. Kalaoja and P. Lago, "Toward an Architectural Knowledge Base for Wireless Service Engineering," *IEEE Transactions on Software Enginnering*, 31(5):361-379, May 2005.

(Nuseibeh, 2001) B. Nuseibeh, "Weaving the Software Development Process Between Requirements and Architecture", From Software Requirements to Architectures (STRAW '01), collocated to 23rd International Conference on Software Engineering, ICSE 2001.

(Nuseibeh & Easterbrook, 2000) B. Nuseibeh, S. Easterbrook. "Requirements Engineering: A Roadmap, The Future of Software Engineering," 22nd International Conference on Software Engineering, ACM-IEEE, pp. 37-46, 2000.

(Nuseibeh et al., 1994) B. Nuseibeh, J. Kramer, and A. Finkelstein, "A Framework for Expressing the Relationship Between Multiple Views in Requirements Specification," *IEEE Transactions on Software Engineering*, October 1994, pp. 760-773

(OCL Specification, 2005) OMG, OCL 2.0 Specification, Version 2.0, http://www.omg.org/docs/ptc/05-06-06.pdf, 2005.

(OCL tools, 2005) OCL tools & services http://www.klasse.nl/ocl/ocl-services.html, 2005.

(Odell et al., 2000) J. Odell, H. Van Dyke Parunak, B. and Bauer, "Extending UML for Agents", Proceedings of the Agent-Oriented Information System Workshop at the 17 National Conference on Artificial Intelligence, Austin, USA, July 2000, pp. 3-17.

(Oreizy et al., 1998) P. Oreizy, N. Medvidovic, R. N. Taylor, "Architecture-Based Runtime Software Evolution", Proceedings of the 20th International Conference on Software Engineering (ICSE-98), Kyoto, April 1998.

(OptimalJ, 2005) OptimalJ 4.0, User's Guide, Compuware, June 2005, http://www.compuware.com/products/optimalj.

(Ortiz et al., 2005) F. J. Ortiz D. Alonso, B. Álvarez, J. A. Pastor, "A Reference Control Architecture for Service Robots Implemented on a Climbing Vehicle", Proceedings 10th Ada-Europe International Conference on Reliable Software Technologies, York, UK, June 20-24, 2005, pp. 13-24

(Osborne & MacNish, 1996) M. Osborne, C. K. MacNish, "Processing Natural Language Software Requirement Specifications", 2nd International Conference on Requirements Engineering (ICRE '96), April 15 - 18, 1996, Colorado Springs, Colorado.

(Pang & Blair, 2004) J. Pang, L. Blair, "Refining Feature Driven Development - A Methodology for Early Aspects", Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop, March 22, 2004 - Lancaster, UK in conjunction with 3rd Aspect-Oriented Software Development Conference (AOSD).

(Padak & Padak, 1994) N. Padak and G. Padak, "Guidelines for Planning Action Research Projects", *Research To Practice*, October 1994.

(Parnas & Madey, 1995) D.L. Parnas, J. Madey, "Functional documents for computer systems", *Science of Computer Programming*, 25: 41-61, 1995.

(Parnas, 1972) D. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of ACM*, 15(12): 1053–1058, 1972.

(Perry & Wolf, 1992) D.E. Perry and A.L. Wolf, "Foundations for the study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, 17(4):40 52, October 1992.

(Pérez, 2006) J. Pérez, *PRISMA*. Phd of the Computer Department of Information Systems and Computation, Universidad Politécnica de Valencia, 2006.

(Perez et al., 2003) J. Pérez, "Oasis como Soporte Formal para la Definición de Modelos Hipermedia Dinámicos, Distribuidos y Evolutivos", Technical Report, DSIC-II/23/03, Polytechnic University of Valencia, October, 2003.

(Pinto et al., 2005) M. Pinto, L. Fuentes, J.M. Troya, "A Dynamic Component and Aspect-Oriented Platform," *The Computer Journal* 48(4):401–420, 2005.

(Pfleger and Hayes-Roth, 1997)K. Pfleger and B. Hayes-Roth. "An Introduction to Blackboard-style Systems Organization," Knowledge Systems Laboratory, Stanford University, Stanford, 1997.

(Ponsard et al., 2004) C. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, H. Tran Van, "Early Verification and Validation of Mission-Critical Systems," Proceedings of 9th International Workshop on Formal Methods for Industrial Critical Systems, (Linz, Austria, 2004) 218-228.

(QVT, 2005) MOF Query/Views/Transformations final adopted specification. OMG document ptc/05-11-01, 2005, November 5th 2005.

(Rapanotti et al., 2004) L. Rapanotti, J. G. Hall, M. Jackson, B. Nuseibeh, “Architecture-driven Problem Decomposition,” Proceedings of 12th IEEE International Requirements Engineering Conference (RE'04), September 6-40, 2004, Kyoto, Japan, pp. 80-89.

(Rashid et al., 2003) A. Rashid, A. Moreira, J. Araújo, “Modularisation and composition of aspectual requirements”, Proceedings 2nd international conference on Aspect-oriented software development, March 17 - 21, 2003, Boston, Massachusetts.

(Rashid et al., 2002) Rashid, A., Sawyer, P., Moreira, A. and Araújo, J. “Early Aspects: a Model for Aspect-Oriented Requirements Engineering”, Proceedings of the International Conference on Requirements Engineering, September 9-13, 2002, Essen, Germany, pp. 199-202.

(Rational, 2006) Rational Rose, http://www-306.ibm.com/software/rational/

(Robinson et al., 2003) W. N. Robinson, S. D. Pawlowski, V. Volkov: Requirements interaction management. *ACM Computing Survey* 35(2): 132-190, June 2003.

(Rolland et al., 1999) C. Rolland, G. Grosz and R. Kla: “Experience with Goal-Scenario Coupling in Requirements Engineering”, Proceedings IEEE International Symposium on Requirement Engineering, June 7-11, 1999, Limerick, Ireland.

(Rolland et al., 1998) C. Rolland, C. Souveyet, and C. Ben Achour, “Guiding Goal Modelling using Scenarios”, *IEEE Transactions on Software Engineering* 24(12): 1055-1071, Special Issue on Scenario Management, December 1998.

(Sánchez et al., 2006) P. Sánchez, J. Magno, L. Fuentes, A. Moreira and J. Araújo. "Towards MDD Transformations from Aspect-Oriented Requirements to Aspect-Oriented Architecture". 3rd European Workshop on Software Architecture (EWSA), September 2006, Nantes (France), LNCS 4344, Springer, pp. 159–174.

(Schippers et al., 2004) H. Schippers, P. Van Gorp and D. Janssens. “Leveraging UML Profiles to generate Plugins from Visual Model Transformations”. Software Evolution through Transformations (SETra). Satellite of the 2nd Intl. Conference on Graph Transformation (ICGT), Rome, Italy, October 2, 2004.

(Schmid et al., 2000) R. Schmid, J. Ryser, S. Berner, M.Glinz, R. Reutemann, E. Fahr, A Survey of Simulation Tools for Requirements Engineering, Technical Reports – 2000, Special Interest Group on Requirements Engineering, Germa Informatics Society (GI), August 2000.

(Schobbens et al., 2006) P. Y. Schobbens, P. Heymans, J. C. Trigaux, “Feature Diagrams: A Survey and a Formal Semantics,” Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), Minneapolis, MN, USA, September 11-15, 2006, pp. 139-148.

(Schopf & Berman, 1998) J. Schopf and F. Berman. "Performance prediction in production environments", Proceedings of the 12th International Parallel Processing Symposium, (Orlando, 1998) pp. 647-653.

(Schmid et al., 2006) K. Schmid, K. Krennrich, M. Eisenbarth, "Requirements Management for Product Lines: Extending Professional Tools," 10th International Software Product Line Conference (SPLC'06), August 21-24, 2006 Baltimore, Maryland, USA , pp. 113-122

(Schmidt et al., 2000) D. C. Schmidt M. Stal, H. Rohnert and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley & Sons, 2000.

(Selic, 2003) B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software*, 20(5): 19-25, Sept/Oct, 2003.

(Sendall & Kozaczynski, 2003) S. Sendall, W. Kozaczynski, "Model Transformation: The Heart and Soul of Model-Driven Software Development", *IEEE Software*, 20(5): 42-45 (2003).

(Shaw & Clements, 2006) M. Shaw and P. Clements, "The Golden Age of Software Architecture", *IEEE Software*, 23(2): 31-39, March/April 2006.

(Shaw & Clements, 1997) M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", Proceedings Annual International Computer Software and Applications Conference (COMPSAC'97), Washington, DC, Aug. 1997.

(Shaw, 1996) Mary Shaw, "Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status". Lecture Notes in Computer Science No 1078, Springer-Verlag, pp. 17-32, 1996.

(Shaw & Garlan, 1996) M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

(Shaw & Garlan, 1994) M. Shaw and D. Garlan. Characteristics of higher-level languages for Software Architecture. Technical Report CMU-CS-94-210, School of Computer Science, Carnegie Mellon University, December 1994.

(Shaw, 1994) M. Shaw, "Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status". Workshop on Studies of Software Design, January,1994.

(Siau & Tan, 2005) K. Siau, X. Tan, "Technical Communication in Information Systems Development: The Use of Cognitive Mapping" *IEEE Transactions on Professional Communications*: 48(3): 269-284, 2005.

(Shonle et al., 2003) M. Shonle, K. J. Lieberherr, A. Shah, "XAspects: an extensible system for domain-specific aspect languages", Proceedings Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, pp. 28-37.

(Souza & Wills, 1999) D. Souza, A. Wills, *Objects, Components and Frameworks with UML*. The Catalysis approach. Addison-Wesley, 1999.

(SPEM, 2005) OMG, Software Process Engineering Metamodel (SPEM), Version 1.1 formal/05-01-06, http://www.omg.org/cgi-bin/doc?formal/2005-01-06, January 2005.

(Sprinkle et al., 2003) J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, G. Karsai: "Domain Model Translation Using Graph Transformations". Proceedings 10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003), 7-10 April 2003, Huntsville, AL, USA. IEEE Computer Society 2003, pp. 159-167.

(Sutcliffe, 1998) A.G. Sutcliffe: Scenario-based requirement analysis", *Requirements Engineering Journal*, 3(1):48-65, 1998.

(Sutton & Rouvellou, 2004) S. M. Sutton and I. Rouvellou, "Concern Modeling for Aspect-Oriented Software Development," in *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, Eds.: Addison-Wesley, 2004, pp. 479-505.

(Sutton & Tarr, 2002) S. M. Sutton Jr., and P. Tarr. "Aspect-oriented design needs concern modeling," Position paper in the Aspect Oriented Design workshop in conjunction with AOSD 2002, Enschede, The Netherlands, Apr. 2002.

(Suvée et al., 2005) D. Suvée, W. Vanderperren, D. Wagelaar, V. Jonckers, "There Are No Aspects". *Electronical Notes in Theoretical Computer Sciences* (ENTCS), Special Issue on Software Composition, 114:153-174, January, 2005.

(Suzuki & Yamamoto, 1999) J. Suzuki and Y. Yamamoto, "Extending UML with Aspects: Aspect Support in the Design Phase", AOP Workshop at ECOOP'99, Lisbon, Portugal, 1999.

(Szyperski, 1998) C. Szyperski, *Component software: beyond object-oriented programming*. Addison Wesley, 1998.

(Tarr et al., 1999) P. Tarr, H. Ossher, W. Harrison, S. Sutton. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". Proceedings in of the International Conference on Software Engineering (ICSE'99), Los Angeles, CA, USA, 16-22 May 1999, pp 107-119.

(Tratt, 2006) L. Tratt, "The MT Model Transformation Language," Proceedings of ACM SIGAPP Symposium on Applied Computing (SAC'06), Dijon, France (2006), pp. 1296 – 1303.

(Trigaux & Heymans, 2003) J.C. Trigaux and P. Heymans, Modelling variability requirements in Software Product Lines: A comparative survey, Technical report PLENTY project, Institut d'Informatique FUNDP, Namur, Belgium, November 2003.

(Toval et al., 2003) J. A. Toval, V. Requena, J. L. Fernández, "Emerging OCL tools," *Software and System Modeling* 2(4): 248-261, 2003.

(RDCU, 2003) UPCT, Remote Control Unit Requirements, Administrative report of the EFTCoR Project, EFTCoR WP7 D1 a UPCT draft, March 20, 2003.

(UML, 2005) OMG, UML 2.0: Superstructure Specification, OMG Adopted Specification, September 4, http://www.omg.org/cgi-bin/doc?formal/05-07-04, July 5, 2005.

(Visio, 2003) Visio 2003, http://office.microsoft.com/es-es/FX010857983082.aspx

(Van et al., 2004) H. Tran Van, A. van Lamsweerde, P. Massonet, C. and Ponsard, "Goal-Oriented Requirements Animation", Proceedings International Conference on Requirements Engineering, September 06 - 10, 2004 Kyoto, Japan, IEEE Computer Society 2004.

(Wadsworth, 1998) Y. Wadsworth, What is Participatory Action Research? Action Research International, Paper 2. Available on-line: http://www.scu.edu.au/schools/gcm/ar/ari/p-ywadsworth98.html

(Walker et al., 2003) D. Walker, S. Zdancewic and J. Ligatti, "A theory of aspects", Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden, August 25-29, 2003, pp.127-139

(Westhuizen & Hoek , 2002) C. Van der Westhuizen and A. van der Hoek: "Understanding and Propagating Architectural Change", Proceedings of the Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA 3), Montreal, Canada, August 2002.

(Whittle & Schumann, 2000) J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios", Proceedings of the International Conference on Software Engineering (ICSE), Limerick, Ireland, 2000, pp. 314-323.

(Wile, 2001) D. Wile, "Residual Requirements and Architectural Residues", Proceedings of the 5th International Symposium on Requirements Engineering (RE'01), Toronto, Canada, pp.194-201, IEEE CS Press, 2001.

(Whittle & Araujo, 2004) J. Whittle and J. Araujo, "Scenario Modeling with Aspects," IEE Proceedings -Software, vol. 151, pp. 157-172, 2004.

(Yu et al., 2004) Y. Yu, J. C. Sampaio do Prado Leite, and J. Mylopoulos, "From Goals to Aspects: Discovering Aspects from Requirements Goal Models", Proceedings 12th IEEE International Requirements Engineering Conference (RE'04), September 06 - 10, 2004, Kyoto, Japan, IEEE Computer Society 2004.

(Yu & Mylopoulos, 1995) E. Yu and J. Mylopoulos: "From E-R to 'A-R': Modelling Strategic Actor Relationships for Business Process Reengineering," *International Journal of Intelligent and Cooperative Information Systems*, 4: 125-144, 2/3, 1995.

(XSLT, 1999) XSL Transformations (XSLT), Version 1.0, W3C Recommendation 16 November 1999, http://www.w3.org/TR/xslt

(Zave & Jackson, 1997) P. Zave and M. Jackson: Four dark corners of requirements engineering. *ACM Transaction Software Engineering and Methodology*, 6(1):1-30, Jan. 1997.

(Zave, 1997) P. Zave: Classification of research efforts in requirements engineering. *ACM Computing Surveys*, 29(4):315--321, 1997.

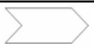
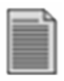
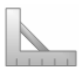
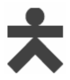
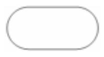
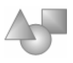
LIST OF FIGURES

LIST OF TABLES

Appendix A. Software Process Engineering Metamodel

As was described in the previous chapters, during the definition of ATRIUM several processes had to be described, considering the inputs for each process, activities and the steps which conform each activity. Several alternatives were possible, but, finally, Software Process Engineering Metamodel (SPEM) was selected.

SPEM is a metamodel for defining processes and their constituting components, oriented to the engineering process. It is based on UML, facilitating a profile which gives supports to any needed concept. Those elements of the profile that have been used in this work are presented in the following table, along with their base class, description and notation. The explanation has been directly extracted from (SPEM, 2005).

Stereotype	Base Class	Description	Notation
Activity	ActivityGraphs:: ActionState	It is the main subclass of WorkDefinition. It describes a piece of work performed by one ProcessRole: the tasks, operations, and actions that are performed by a role or with which the role may assist.	
Document	ActivityGraphs:: ObjectFlowState	Any document used or generated.	
Guidance	Core::Comment	Guidance elements may be associated with ModelElements, to provide more detailed information to practitioners about the associated ModelElement. Possible types of Guidance depend on the process family and can be for example: Guidelines, Techniques, Metrics, Examples, UML Profiles, Tool mentors, Checklist, Templates.	
ProcessRole	UseCases:: Actor	ProcessRole defines responsibilities over specific WorkProducts, and defines the roles that perform and assist in specific activities.	
Step	ActivityGraphs:: ObjectFlowState	It is an atomic element for the definition of an Activity.	
WorkProduct	ActivityGraphs:: ObjectFlowState	A work product or artifact is anything produced, consumed, or modified by a process. It may be a piece of information, a document, a model, source code, and so on. A WorkProduct describes one class of work product produced in a process.	

		A WorkProductKind describes a category of work product, such as Text Document, UML Model, Executable, Code Library, and so on. The range of work product kinds is dependent on the process being modeled.
WorkDefinition	ActivityGraphs:: ActionState	WorkDefinition is a kind of Operation that describes the work performed in the process. Its main subclass is Activity, but Phase, Iteration, and Lifecycle (in the Process Lifecycle package) are also subclasses of WorkDefinition. WorkDefinition is not an abstract class, and instances of WorkDefinition itself can be created to represent composite pieces of work that are further decomposed. It has explicit inputs and outputs referred to via ActivityParameter.

Appendix B. Transforming ATRIUM Scenarios – PRISMA

In the following sections, the Relations that have been defined are introduced according to its kind, that is, architectural generative patterns, architectural style-oriented transformation, and idioms-oriented transformation. It can be observed the latter transformation imports the others because they factorize the Relations that are used by it.

B.1 ARCHITECTURAL GENERATIVE PATTERN

```
transformation ATRIUM2ToArchModel(scenarios:ATRIUMScenarios;
archModel: ArchitecturalModel)
{
key archModel::System {name};
key archModel::Component {name};
key archModel::Connector {name};
key archModel::Port {name, ArchitecturalElement};
key archModel::Attachment {name, System};
key archModel::Aspect {name};

top relation FragmentToSystems
{
   cn: String;    c:archModel::Component;
   checkonly domain scenarios p:SystemFrame{
         fragment=sf:SystemFrame {systemName=cn}
   }{p.fragment->notEmpty()};
   enforce domain archModel s:System{
      name=cn
   };
   where{
      MessageToArchElements(sf, s);
      MessageBetweenComponentsToArchElements(sf, s);
      GeneralOrderingToWeaving(sf, s);
   }
}

relation SystemFrameToConnector
{
   cn: String;
   checkonly domain scenarios sf:SystemFrame{
      lifeline=l:Lifeline{name=cn}
   }{l->oclIsKindOf(Connector)};
   enforce domain archModel c:Connector{
      name=cn
   };
}
```

```
relation SystemFrameToComponent
{
   cn: String;
   checkonly domain scenarios sf:SystemFrame{
      lifeline=l:Lifeline{name=cn}
   }{l->oclIsKindOf(Component)};
   enforce domain archModel c:Component{
      name=cn
   };
}

query NameConnector(l1:ATRIUMScenarios::Lifeline,
l2:ATRIUMScenarios::Lifeline):ATRIUMScenarios::Lifeline
{
   if (l1->oclIsTypeOf(ATRIUMScenarios::Connector)) then     l1
   else  l2    endif
}

query NameComponent(l1:ATRIUMScenarios::Lifeline,
l2:ATRIUMScenarios::Lifeline):ATRIUMScenarios::Lifeline
{
   if (l1->oclIsTypeOf(ATRIUMScenarios::Component))  then    l1
   else  l2    endif
}

query KindService(s:String):String
{
   if (s='in')    then  'in'
   else  'out'    endif
}

relation MessageToArchElements
{
   cn: String;    cn1: String;   cn2: String;   cn3: String;
   cn4: String;   p:ARCHMODEL::Port;
   comp:ARCHMODEL::Component;    con:ARCHMODEL::Connector;
   lcon:ATRIUMScenarios::Lifeline;  lcom:ATRIUMScenarios::Lifeline;
   seqArch:Sequence(ARCHMODEL::ArchitecturalElement);
   checkonly domain scenarios sf:SystemFrame
   {
      message=m:Message{
         name=cn,
         sendEvent=m1:MessageOccurrenceSpecification{
            covered=l1:Lifeline{name=cn1}},
         receiveEvent=m2:MessageOccurrenceSpecification{
            covered=l2:Lifeline{name=cn2}}
      }
   }{(l1->oclIsKindOf(Connector) and l2->oclIsKindOf(Component)) or
      (l1->oclIsKindOf(Component) and l2->oclIsKindOf(Connector))};
   enforce domain archModel s:System{
      containsComps= comp ,
      containsCnct= con
   };
   where {
      lcom=NameComponent( l1, l2);
      lcon=NameConnector( l1, l2);
      LifelineComponentToComponent(lcom, comp);
      LifelineConnectorToConnector(lcon, con);
      MessageToAttachmentComponent(lcom, s, lcon.name, lcom.name);
      MessageToAttachmentConnector(lcon, s, lcon.name, lcom.name);
      LifelineToServiceAspectComponent(lcom, comp);
      LifelineToServiceAspectConnector(lcon, con);
   }
}

relation MessageToAttachmentComponent
{
   cn:String;  nComp:String;  attName:String;
   checkonly domain scenarios l:Lifeline{name=cn};
   enforce domain archModel s:System{
      connect=a:Attachment{
         name=attName,
         linkPort=p:Port{
            name=nComp,
            ArchitecturalElement=c:Component{name=cn}
         }
      }
   };
   primitive domain portComp:String;
   primitive domain portCon:String;
   where{
      attName=portComp+portCon;
      nComp=portComp;
   }
}

relation MessageToAttachmentConnector
{
   cn:String;     attName:String;   nCon:String;
   checkonly domain scenarios l:Lifeline{name=cn};
   enforce domain archModel s:System{
      connect=a:Attachment{
         name=attName,
         linkPort=p:Port{
            name=nCon,
            ArchitecturalElement=c:Connector{name=cn}
         }
      }
   };
   primitive domain portComp:String;
   primitive domain portCon:String;
   where{
      attName=portComp+portCon;
      nCon=portCon;
   }
}

relation LifelineComponentToComponent
{
   cn: String;
   checkonly domain scenarios l:Lifeline{
      name=cn
   }{l->oclIsKindOf(Component)};
   enforce domain archModel con :Component{
      name=cn
   };
}

relation LifelineConnectorToConnector
{
   cn: String;
   checkonly domain scenarios l:Lifeline{
      name=cn
   }{l->oclIsKindOf(Connector)};
   enforce domain archModel con :Connector{
      name=cn
   };
}

relation ArchitecturalElementsToAttachment
{
   cn:String;
   checkonly domain scenarios sf:SystemFrame{};
   enforce domain archModel a:Attachment{   name=cn  };
   primitive domain s:String;
   where{   cn=s; }
}
}
```

B.2 ARCHITECTURAL STYLE-ORIENTED TRANSFORMATION

```
Transformation
ATRIUM2ToArchModelACROSETStyle(scenarios:ATRIUMScenarios;
archModel: ArchitecturalModel)
{
key archModel::System {name};
key archModel::Component {name};
key archModel::Connector {name};
key archModel::Port {name, ArchitecturalElement};
key archModel::Attachment {name, System};
key archModel::Aspect {name};

top relation TransformationApplyingACROSET
{
   cn: String;    ln1: String;   sn1: String;   ln2: String;
   sn2: String;   attName:String;
   s1:archModel::System;   s2:archModel::System;   ps1:archModel::Port;
   ps2:archModel::Port;    pae1:archModel::Port;
   pae2:archModel::Port;
   checkonly domain scenarios p:Package
   {
      packagedElement= i:Interaction{
         message= m:Message{
            name=cn,
            sendEvent=m1:MessageOccurrenceSpecification{
               covered=l1:Lifeline{
                  name=ln1,
                  interaction=i1:SystemFrame{systemName=sn1}
               }
            },
            receiveEvent=m2:MessageOccurrenceSpecification{
               covered=l2:Lifeline{
                  name=ln2,
                  interaction=i2:SystemFrame{systemName=sn2}
               }
            }
         }
      }
   }{i.message->notEmpty() and
      ((i1.systemName.substring(i1.systemName.size()-2,
         i1.systemName.size())='MUC' and
      i2.systemName.substring(i2.systemName.size()-2,
         i2.systemName.size())='SUC') or
      (i1.systemName.substring(i1.systemName.size()-2,
         i1.systemName.size())='RUC' and
      i2.systemName.substring(i2.systemName.size()-2,
         i2.systemName.size())='MUC'))
      and (l1->oclIsTypeOf(Component) or l1->oclIsTypeOf(Connector))
      and (l2->oclIsTypeOf(Component) or l2->oclIsTypeOf(Connector))};
   enforce domain archModel s:System{
      name=ps1.ArchitecturalElement.name,
      connect=att:Attachment{
         name=attName,
         linkPort=ps1
      }
   };
   where{
      attName=sn1+ln1+sn2+ln2;
      LifelineToArchitecturalElementBinding(l1, ps1, sn2+ln2);
      MessageToBinding2(l2, s, sn1+ln1, sn2+ln2);
   }
}

top relation MessageToBindingComposed
{
   cn: String;    ln1: String;   sn1: String;   ln2: String;
   sn2: String;   attName:String;   s1:archModel::System;
   s2:archModel::System;   ps1:archModel::Port;    ps2:archModel::Port;
   pae1:archModel::Port;   pae2:archModel::Port;
   checkonly domain scenarios p:Package
   {
      packagedElement= i:Interaction{
         message= m:Message{
            name=cn,
            sendEvent=m1:MessageOccurrenceSpecification{
               covered=l1:Lifeline{
                  name=ln1,
                  interaction=i1:SystemFrame{systemName=sn1}
               }
            },
            receiveEvent=m2:MessageOccurrenceSpecification{
               covered=l2:Lifeline{
                  name=ln2,
                  interaction=i2:SystemFrame{systemName=sn2}
               }
            }
         }
      }
   }{i.message->notEmpty() and
      ((i1.systemName.substring(i1.systemName.size()-2,
         i1.systemName.size())='SUC' and
      i2.systemName.substring(i2.systemName.size()-2,
         i2.systemName.size())='MUC') or
      (i1.systemName.substring(i1.systemName.size()-2,
         i1.systemName.size())='MUC' and
      i2.systemName.substring(i2.systemName.size()-2,
         i2.systemName.size())='RUC'))
      and (l1->oclIsKindOf(Component) or l1->oclIsKindOf(Connector))
      and (l2->oclIsKindOf(Component) or l2->oclIsKindOf(Connector))};
   enforce domain archModel s:System{
      name=ps1.ArchitecturalElement.name,
      connect=att:Attachment{
         name=attName,
         linkPort=ps1
      }
   };
   where{
      attName=sn2+ln2+sn1+ln1;
      LifelineToArchitecturalElementBinding(l2, ps1, sn1+ln1);
      MessageToBinding2(l2, s, sn2+ln2, sn1+ln1);
   }
}

relation MessageToBinding2
{
   cn: String;    ln1: String;   sn1: String;   ln2: String;
   sn2: String;                  attName:String;
   s1:archModel::System;         s2:archModel::System;
   ps1:archModel::Port;          ps2:archModel::Port;
   pae1:archModel::Port;         pae2:archModel::Port;
   checkonly domain scenarios l2:Lifeline{
                 name=ln2,
                 interaction=i2:SystemFrame{systemName=sn2}
   };
   enforce domain archModel s:System{
      connect=att:Attachment{
         name=attName,
         linkPort=ps2
      }
   };
   primitive domain portName1:String;
   primitive domain portName2:String;
   where{
       attName= portName1 + portName2;
       LifelineToArchitecturalElementBinding(l2, ps2, portName1);
   }
}

relation LifelineToArchitecturalElementBinding
{
   cn: String;    ln: String;    pn:String;
   checkonly domain scenarios l:Lifeline{
      name=ln,
      interaction=i:SystemFrame{systemName=cn}
   };
   enforce domain archModel p:Port{
      name=pn,
      ArchitecturalElement=s:System{name=cn }
   };
   primitive domain portName:String;
   where{
      pn=portName;
      LifelineToComponentBinding(l, p, portName);
      LifelineToConnectorBinding(l, p, portName);
   }
}

relation LifelineToComponentBinding
{
   sn: String;    pn: String;    ln: String;    bn:String;
   a:archModel::ArchitecturalElement;    b:archModel::Binding;
   checkonly domain scenarios l:Lifeline{
      name=ln,
      interaction=sf:SystemFrame{systemName=sn}
   }{l->oclIsKindOf(Component)};
   enforce domain archModel p:Port{
      name=pn,
      ArchitecturalElement=s:System{
         name=sn,
         containsComps=c:Component{
            name=ln,
            has=pc1:Port{
               name=pn,
               isComponent=b:Binding{
                  name= ln
               }
            }
         },
         has=pc2:Port{
            name=pn,
            isComposed=b
         }
      }
   };
   primitive domain portName:String;
   where{
      pn=portName;
      nameBinding= ln;
   }


relation LifelineToConnectorBinding
{
   sn: String;    pn: String;    ln: String;
   a:archModel::ArchitecturalElement;    b:archModel::Binding;
   checkonly domain scenarios l:Lifeline{
      name=ln,
      interaction=sf:SystemFrame{systemName=sn}
   }{l->oclIsKindOf(Connector)};
   enforce domain archModel p:Port{
      name=pn,
      ArchitecturalElement=s:System{
         name=sn,
         containsCnct=c:Connector{
            name=ln,
            has=pc1:Port{
               name=pn,
               isComponent=b:Binding{
                  name= ln
               }
            }
         },
         has=pc2:Port{
            name=pn,
            isComposed=b
         }
      }
   };
   primitive domain portName:String;
   where
   {
      pn=portName;
      nameBinding= ln;
   }
}
}
```

B.3 IDIOMS-ORIENTED TRANSFORMATION

```
import archpatt.qvt;
import archstyle.qvt;
transformation ATRIUM2ToArchModelPRISMA(scenarios:ATRIUMScenarios;
archModel: ArchitecturalModel)
{
key archModel::System {name};
key archModel::Component {name};
key archModel::Connector {name};
key archModel::Port {name, ArchitecturalElement};
key archModel::Attachment {name, System};
key archModel::Aspect {name};

relation LifelineToServiceAspectConnector
{
   cn: String;    cn1:String;    cn2:String;    as:ARCHMODEL::Aspect;
   checkonly domain scenarios l:Lifeline{
      coveredBy=mo:MessageOccurrenceSpecification {
         message=m:Message{}
      }
   }{m->oclIsTypeOf(Message)};
   enforce domain archModel c:Connector{
      imports=as
   };
   when{    LifelineToAspect(l,as, 'Coordination');  }
   where{
      cn1=mo.event.name.substring(1,2);
      cn2=KindService(cn1);
      LifelineToService(m, as, cn2);
   }
}

relation LifelineToServiceAspectComponent
{
   cn: String;   cn1:String;   cn2:String;   as:ARCHMODEL::Aspect;
   checkonly domain scenarios l:Lifeline{
      coveredBy=mo:MessageOccurrenceSpecification {
         message=m:Message{}
      }
   }{m->oclIsTypeOf(Message)};
   enforce domain archModel c:Component{
      imports=as
   };
   when{ LifelineToAspect(l,as, 'Functional'); }
   where{
      cn1=mo.event.name.substring(1,2);
      cn2=KindService(cn1);
      LifelineToService(m, as, cn2);
   }
}

relation LifelineToService
{
   cn: String; cn1: String;
   checkonly domain scenarios m:Message{ name=cn };
   enforce domain archModel as:Aspect{
      belongsTo=s:Service{
         type=cn1,
         name=cn
      }
   };
   primitive domain parType:String;
   where{  cn1=parType;  }
}

relation LifelineToAspect
{
   cn: String; conc: String;
   checkonly domain scenarios l:Lifeline{   name=cn  };
   enforce domain archModel a:Aspect{
      name=cn,
      concern=conc
   };
   primitive domain parConcern:String;
   where{
      conc=parConcern;
      LifelineToBegin(l,a);
      LifelineToEnd(l,a);
   }
}

relation LifelineToBegin
```

```
{
   cn: String;
   checkonly domain scenarios l:Lifeline{};
   enforce domain archModel a:Aspect   {
      belongsTo= b1:Service{name='begin()'}
   };
}
relation LifelineToEnd
{
   cn: String;
   checkonly domain scenarios l:Lifeline{};
   enforce domain archModel a:Aspect   {
      belongsTo= b1:Service{name='end()'}
   };
}

relation MessageBetweenComponentsToArchElements
{
   cn: String;    cn1: String;   cn2: String;
   comp:ARCHMODEL::Component;    con:ARCHMODEL::Connector;
   checkonly domain scenarios sf:SystemFrame
   {
      message=m:Message
      {
         name=cn,
         sendEvent=m1:MessageOccurrenceSpecification{
            covered=l1:Lifeline{name=cn1}},
         receiveEvent=m2:MessageOccurrenceSpecification{
            covered=l2:Lifeline{name=cn2}}
      }
   }{(l1->oclIsKindOf(Component) and l2->oclIsKindOf(Component))};
   enforce domain archModel s:System{
      containsComps= comp,
      containsCnct=con
   };
   where {
      LifelineComponentToComponent(l1, comp);
      LifelineConnectorToConnectorBetweenComponents(l1, con);
      MessageToAttachmentComponent(l1, s, con.name, comp.name);
      MessageToAttachmentConnectorBetweenComponents(l1, s,
         con.name, comp.name);
      MessageReceiveBetweenComponentsToArchElements(l2, s, cn1);
   }
}

relation MessageReceiveBetweenComponentsToArchElements
{
   cn: String;    cn1: String;   cn2: String;
   comp:ARCHMODEL::Component;    con:ARCHMODEL::Connector;
   att:ARCHMODEL::Attachment;
   checkonly domain scenarios l:Lifeline{   name=cn  };
   enforce domain archModel s:System{
      containsComps= comp,
      containsCnct=con
   };
   primitive domain compName:String;
   where{
      LifelineComponentToComponent(l, comp);
      LifelineConnectorToConnectorBetweenComponents(l, con);
      MessageToAttachmentComponent(l, s, con.name, comp.name);
      MessageToAttachmentConnectorBetweenComponents(l, s, con.name,
         comp.name);
   }
}

relation MessageToAttachmentConnectorBetweenComponents
{
   sn:String;  cn:String;  attName:String;   nCon:String;
   checkonly domain scenarios l:Lifeline{
      interaction=sf:SystemFrame{name=sn}
   };
   enforce domain archModel s:System{
      connect=a:Attachment{
         name=attName,
         linkPort=p:Port{
            name=nCon,
            ArchitecturalElement=c:Connector{name=cn}
         }
      }
   };
   primitive domain portComp:String;
   primitive domain portCon:String;
   where{
      attName=portComp+portCon;
      nCon=portCon;
      cn='Cnct'+sn;
   }
}

relation LifelineConnectorToConnectorBetweenComponents
{
   cn: String;    cn1: String;
   checkonly domain scenarios l:Lifeline{
      interaction=sf:SystemFrame{systemName=cn}
   };
   enforce domain archModel con:Connector{   name=cn1 };
   where{   cn1='Cnct'+cn;    }
}

relation MessageToAspectBetweenComponents
{
   cn: String;    cn1: String;
   checkonly domain scenarios sf:SystemFrame{   systemName=cn1 };
   enforce domain archModel s:System{
      containsCnct= con :Connector {name=cn}
   };
   where{   cn='Cnt'+cn1;  }
}

top relation MessageFromHumanToComponent
   {
      cn: String;  ln1: String; sn: String;  ln2: String; sn2: String;
      comp:archModel::Component;    lcom: ATRIUMScenarios::Lifeline;
      checkonly domain scenarios p:Package
      {
         packagedElement= i:Interaction{
            message= m:Message{
               name=cn,
               sendEvent=m1:MessageOccurrenceSpecification{
                  covered=l1:Lifeline{
                     name=ln1
                  }
               },
               receiveEvent=m2:MessageOccurrenceSpecification{
                  covered=l2:Lifeline{
                     name=ln2,
                     interaction=sf:SystemFrame{systemName=sn}
                  }
               }
            }
         }
      }{i.message->notEmpty() and l1->oclIsKindOf(Human)
         and l2->oclIsKindOf(Component) and m->oclIsKindOf(Message)};
      enforce domain archModel s:System{
         name=sn,
         containsComps=comp
      };
      where{
         lcom=NameComponent( l1, l2);    //this query determines which
                        //lifeline is the Component
         mcom=NameComponentMO(m1, m2); //this query determines which
               //MessageOccurrenSpecification is the Component
         LifelineComponentToComponent(lcom, comp);
         LifelineToServicePresentationAspectComponent(mcom, comp);
      }
   }
   }{i.message->notEmpty() and ((l1->oclIsKindOf(Human) and
      l2->oclIsKindOf(Component)) or (l1->oclIsKindOf(Human) and
      l2->oclIsKindOf(Component))) and m->oclIsKindOf(Message)};
   enforce domain archModel s:System{
      name=sn,
      containsComps=comp
   };
   where{
      LifelineComponentToComponent(l2, comp);
      LifelineToServicePresentationAspectComponent(m2, comp);
   }
}

relation LifelineToServicePresentationAspectComponent
{
   cn: String;    cn1:String;    cn2:String;    as:ARCHMODEL::Aspect;
   checkonly domain scenarios  mo:MessageOccurrenceSpecification{
      covered=l:Lifeline {},
      message=m:Message{}
   };
   enforce domain archModel c:Component{  imports=as  };
   when{    LifelineToAspect(l,as, 'Presentation');   }
   where{      LifelineToService(m, as, 'in');  }
}
}
```

Appendix C. ATRIUM Scenarios

C.1 SAFETY PATTERNS

Alternative Scenarios for the Safety Pattern: Redundant Safety Node

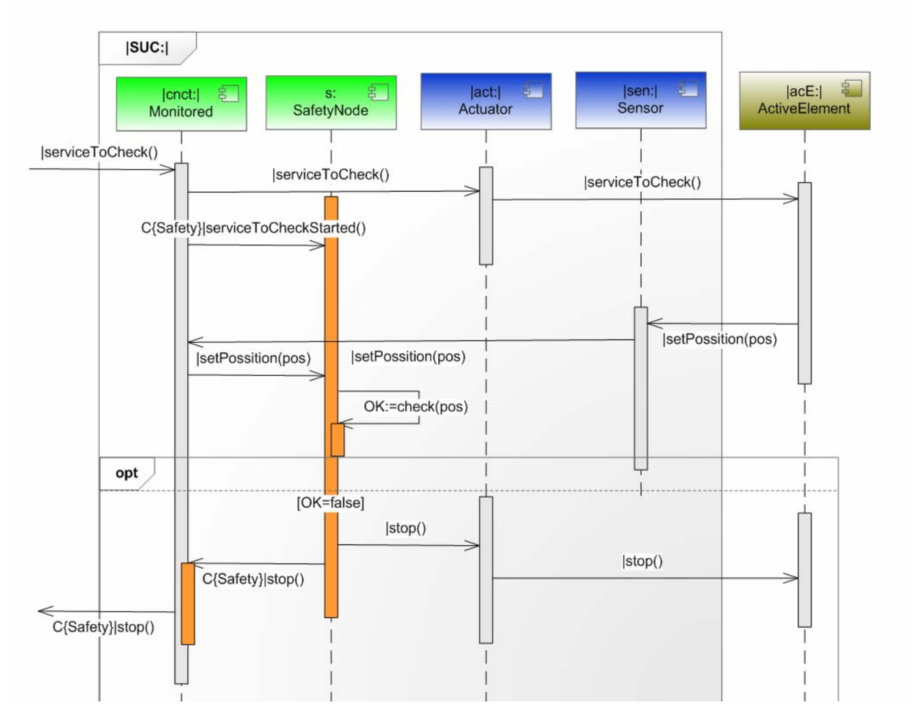


Figure C. 1 SafetyNode detects a fault during the movement

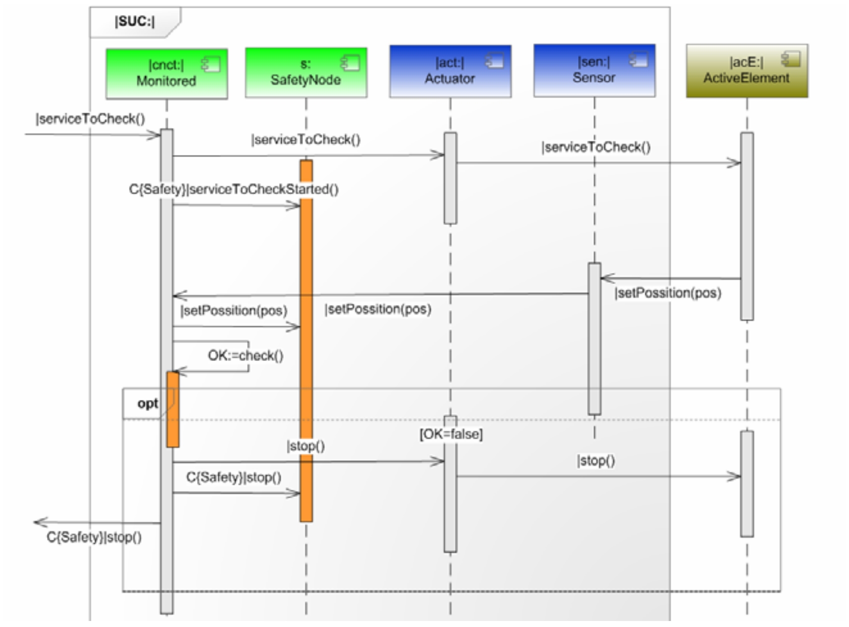


Figure C. 2 Monitored connector detects a fault during the movement

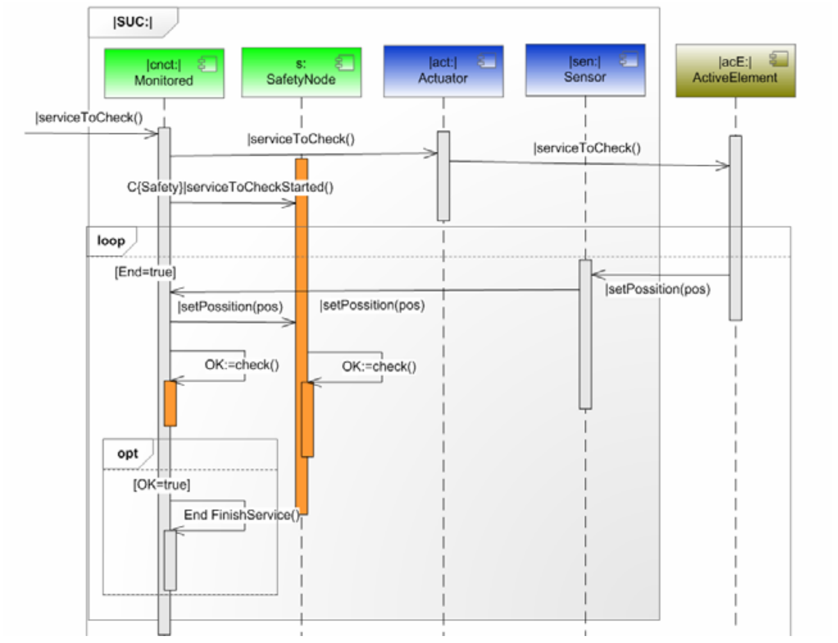
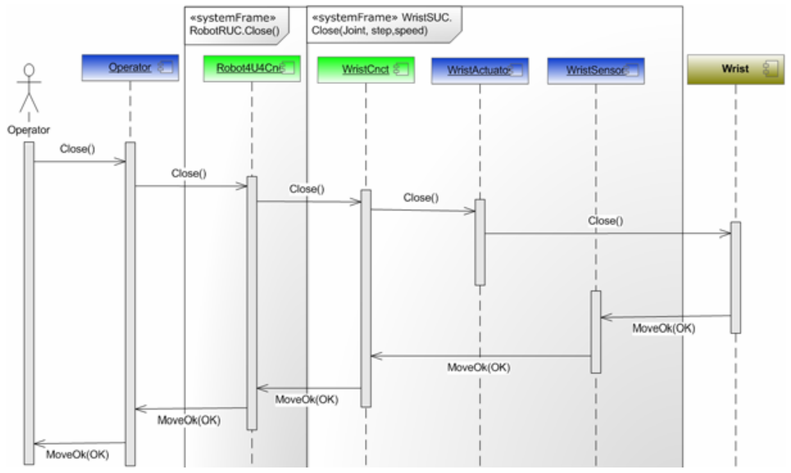


Figure C. 3 Any hazard arises during the movement

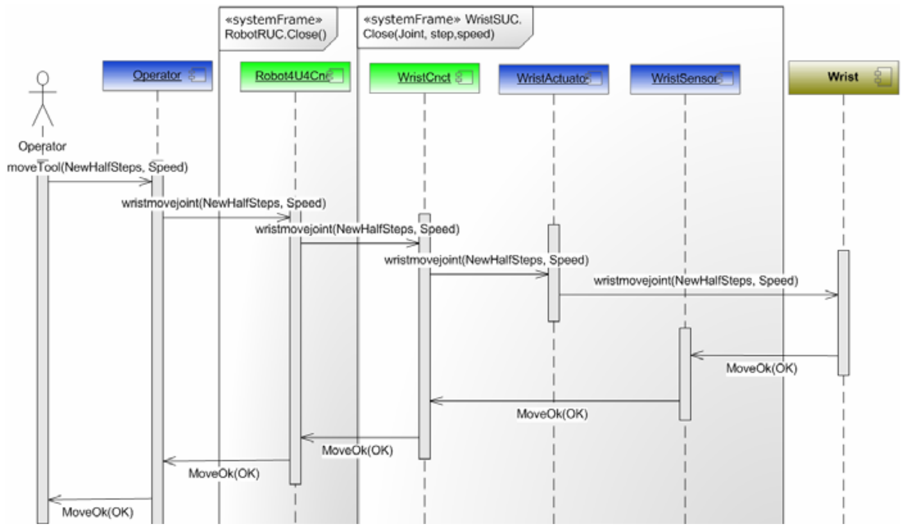
C.2 TEACHMOVER SCENARIOS

OPE. 1 Operational closing of the Wrist by TeachMover Control accessing RUC-SUC

The following scenario describes how the TeachMover controls the close of the tool. It can be observed that the system RUC directly accesses to the tool

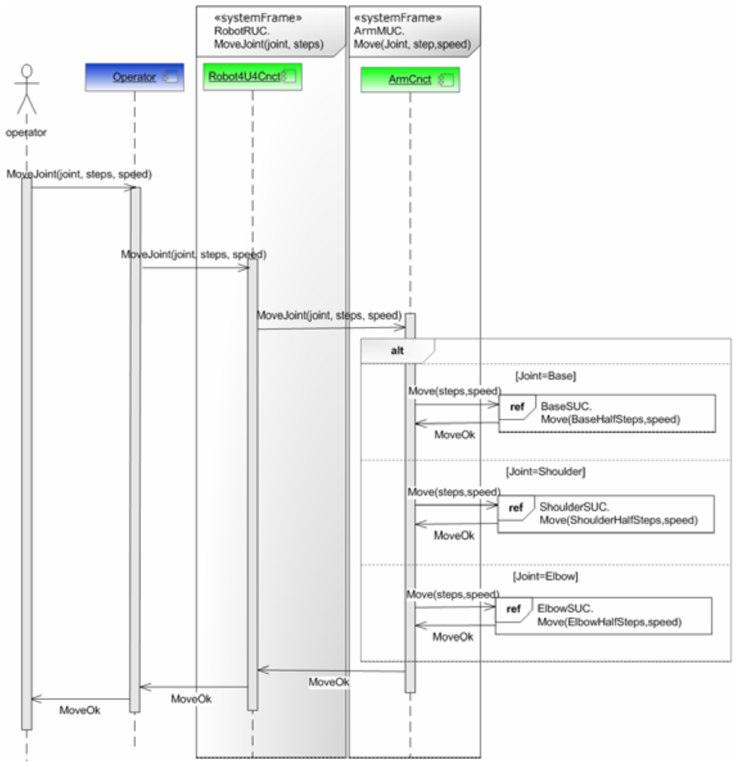


OPE. 2 Operational opening of the Wrist by TeachMover Control accessing RUC-SUC


«systemFrame» RobotRUC.Close()
«systemFrame» WristSUC. Close(Joint, step,speed)
Operator
Robot4U4Cnt
WristCnct
WristActuator
WristSensor
Wrist
Operator
moveTool(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)

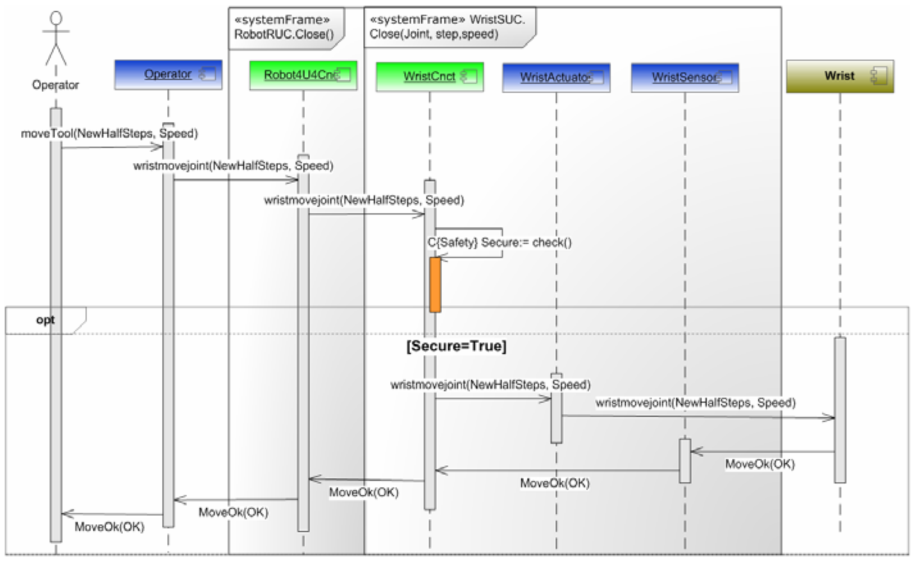
OPE. 8 Operational angular movement of the joint by TeachMover Control accessing RUC-MUC-SUC

In the following scenario it can be observed how the TeachMover can control each joint in a separated way. The scenarios that are referenced are described in the **¡Error! No se encuentra el origen de la referencia.**.

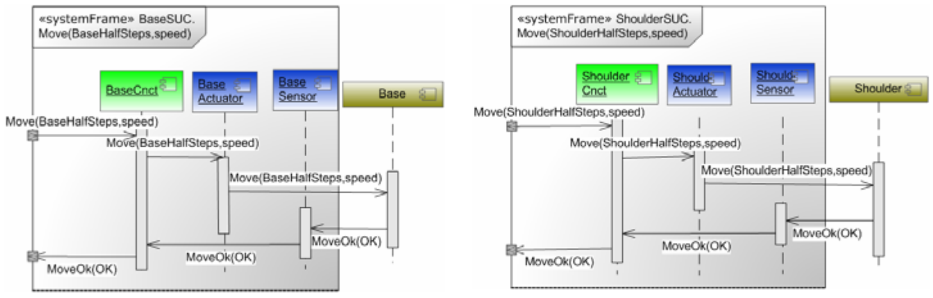

«systemFrame» RobotRUC. MoveJoint(joint, steps)
«systemFrame» ArmMUC. Move(Joint, step,speed)
operator
Operator
Robot4U4Cnct
ArmCnct
MoveJoint(joint, steps, speed)
MoveJoint(joint, steps, speed)
MoveJoint(joint, steps, speed)
alt
[Joint=Base]
Move(steps,speed)
ref BaseSUC. Move(BaseHalfSteps,speed)
MoveOk
[Joint=Shoulder]
Move(steps,speed)
ref ShoulderSUC. Move(ShoulderHalfSteps,speed)
MoveOk
[Joint=Elbow]
Move(steps,speed)
ref ElbowSUC. Move(ElbowHalfSteps,speed)
MoveOk
MoveOk
MoveOk
MoveOk

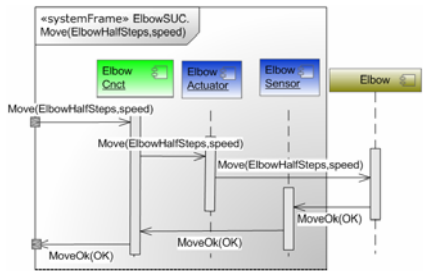
OPE. 15 Add a Safety Aspect to the control of the Wrist

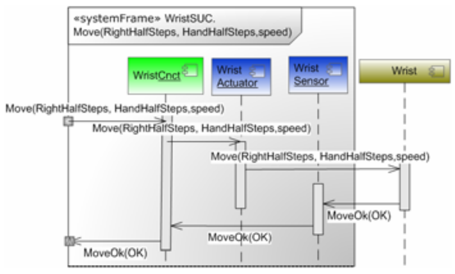
This operationalization describes how the scenario described in **¡Error! No se encuentra el origen de la referencia.** is modified to take into account Safety concerns.

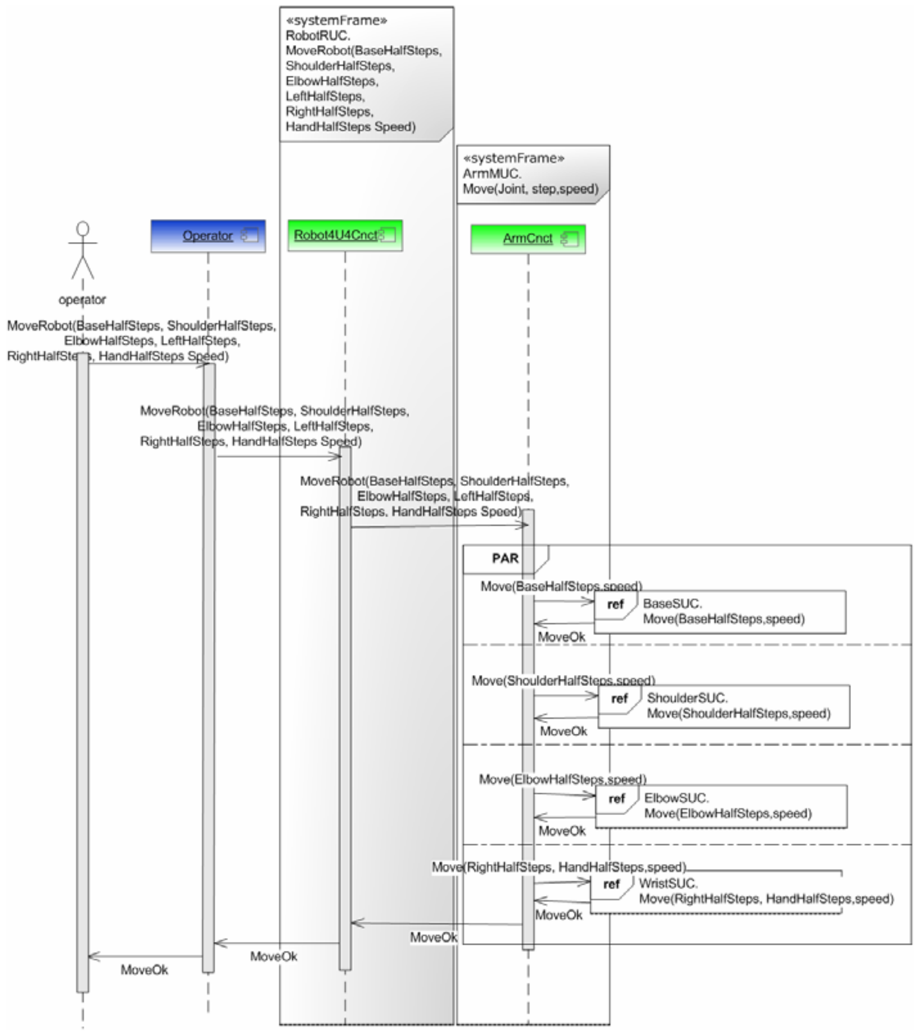

«systemFrame» RobotRUC.Close()
«systemFrame» WristSUC. Close(Joint, step,speed)
Operator
Operator
Robot4U4Cnt
WristCnct
WristActuator
WristSensor
Wrist
moveTool(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
C{Safety} Secure:= check()
opt
[Secure=True]
wristmovejoint(NewHalfSteps, Speed)
wristmovejoint(NewHalfSteps, Speed)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)

OPE. 9 Operational angular movement by TeachMover Control RUC- MUC – SUC


«systemFrame» BaseSUC. Move(BaseHalfSteps,speed)
BaseCnct
Base Actuator
Base Sensor
Base
Move(BaseHalfSteps,speed)
Move(BaseHalfSteps,speed)
Move(BaseHalfSteps,speed)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)
«systemFrame» ShoulderSUC. Move(ShoulderHalfSteps,speed)
Shoulder Cnct
Should. Actuator
Should. Sensor
Shoulder
Move(ShoulderHalfSteps,speed)
Move(ShoulderHalfSteps,speed)
Move(ShoulderHalfSteps,speed)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)


«systemFrame» ElbowSUC.
Move(ElbowHalfSteps,speed)
Elbow Cnct
Elbow Actuator
Elbow Sensor
Elbow
Move(ElbowHalfSteps,speed)
Move(ElbowHalfSteps,speed)
Move(ElbowHalfSteps,speed)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)


«systemFrame» WristSUC.
Move(RightHalfSteps, HandHalfSteps,speed)
WristCnct
Wrist Actuator
Wrist Sensor
Wrist
Move(RightHalfSteps, HandHalfSteps,speed)
Move(RightHalfSteps, HandHalfSteps,speed)
Move(RightHalfSteps, HandHalfSteps,speed)
MoveOk(OK)
MoveOk(OK)
MoveOk(OK)


«systemFrame»
RobotRUC.
MoveRobot(BaseHalfSteps,
ShoulderHalfSteps,
ElbowHalfSteps,
LeftHalfSteps,
RightHalfSteps,
HandHalfSteps Speed)
«systemFrame»
ArmMUC.
Move(Joint, step,speed)
operator
Operator
Robot4U4Cnct
ArmCnct
MoveRobot(BaseHalfSteps, ShoulderHalfSteps, ElbowHalfSteps, LeftHalfSteps, RightHalfSteps, HandHalfSteps Speed)
MoveRobot(BaseHalfSteps, ShoulderHalfSteps, ElbowHalfSteps, LeftHalfSteps, RightHalfSteps, HandHalfSteps Speed)
MoveRobot(BaseHalfSteps, ShoulderHalfSteps, ElbowHalfSteps, LeftHalfSteps, RightHalfSteps, HandHalfSteps Speed)
PAR
Move(BaseHalfSteps,speed)
ref
BaseSUC.
Move(BaseHalfSteps,speed)
MoveOk
Move(ShoulderHalfSteps,speed)
ref
ShoulderSUC.
Move(ShoulderHalfSteps,speed)
MoveOk
Move(ElbowHalfSteps,speed)
ref
ElbowSUC.
Move(ElbowHalfSteps,speed)
MoveOk
Move(RightHalfSteps, HandHalfSteps,speed)
ref
WristSUC.
Move(RightHalfSteps, HandHalfSteps,speed)
MoveOk
MoveOk
MoveOk
MoveOk

Appendix D. From an ATRIUM Scenario to a PRISMA description

D.1 ATRIUM SCENARIO USED FOR GENERATION

In the following is presented how the scenario depicted in **¡Error! No se encuentra el origen de la referencia.** is translated to xmi following the EMOF description of the ATRIUM Scenarios Model.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Package xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="ATRIUMScenarios" xmi:id="_jrkqYK-REdulGdzJuhFZlw"
name="EFTCoR">
-<nestedPackage xmi:id="_Rz8wkLCZEdu59b4jU24OaQ" name="Wrist">
- <packagedElement xsi:type="SystemFrame"
xmi:id="_tfjaMLI0EduhdIbZeY2ECA">
  <message xmi:id="_WSkjULdwEduSi-sHVP1Rgw"
name="wristmovejoint(newLeftHalfStep, newRightHalfStep, speed)"
receiveEvent="//@nestedPackage.0/@packagedElement.0/@fragment.3"
sendEvent="//@nestedPackage.0/@packagedElement.0/@fragment.2" />
  <message xmi:id="_gLvqILa7Edufg-VapKE-6g"
name="wristmovejoint(newLeftHalfStep, newRightHalfStep, speed)"
receiveEvent="_fZBZMLdqEduSi-sHVP1Rgw" sendEvent="_o6h_MLa7Edufg-
VapKE-6g" />
  <message name="wristmovejoint(newLeftHalfStep, newRightHalfStep,
speed)" receiveEvent="_p72gMK-SEdulGdzJuhFZlw"
sendEvent="_4PMkYLdxEduSi-sHVP1Rgw" />
  <message xmi:id="_b-3mgLdqEduSi-sHVP1Rgw"
name="wristmovejoint(newLeftHalfStep, newRightHalfStep, speed)"
receiveEvent="//@nestedPackage.0/@packagedElement.0/@fragment.5"
sendEvent="_3DOw4K-4EdubZJKMO78Tqg" />
  <message name="MoveOk(OK)" receiveEvent="_NsRvcK-7EduWeujrBWnIYg"
sendEvent="//@nestedPackage.0/@packagedElement.0/@fragment.6" />
  <message name="MoveOk(OK)"
receiveEvent="//@nestedPackage.0/@packagedElement.0/@fragment.0/@fragm
ent.2" sendEvent="_OqCGQK-9EduWeujrBWnIYg" />
  <message name="MoveOk(OK)" receiveEvent="_pbdiALa7Edufg-VapKE-6g"
sendEvent="//@nestedPackage.0/@packagedElement.0/@fragment.0/@fragment
.3" />
  <message name="MoveOk(OK)"
receiveEvent="//@nestedPackage.0/@packagedElement.0/@fragment.9"
sendEvent="_i9wEMLdqEduSi-sHVP1Rgw" />
  <lifeline xsi:type="Human" xmi:id="_WYfz4LdqEduSi-sHVP1Rgw"
name="HOperator" coveredBy="_i9wEMLdqEduSi-sHVP1Rgw
//@nestedPackage.0/@packagedElement.0/@fragment.2
//@nestedPackage.0/@packagedElement.0/@fragment.9
//@nestedPackage.0/@packagedElement.0/@fragment.11
//@nestedPackage.0/@packagedElement.0/@fragment.12" />
  <lifeline xsi:type="Component" name="Operator"
coveredBy="//@nestedPackage.0/@packagedElement.0/@fragment.13
```

```
//@nestedPackage.0/@packagedElement.0/@fragment.14 _o6h_MLa7Edufg-
VapKE-6g _pbdiALa7Edufg-VapKE-6g
//@nestedPackage.0/@packagedElement.0/@fragment.3" />
  <lifeline xsi:type="Environment" name="Wrist"
coveredBy="//@nestedPackage.0/@packagedElement.0/@fragment.16
//@nestedPackage.0/@packagedElement.0/@fragment.18
//@nestedPackage.0/@packagedElement.0/@fragment.17
//@nestedPackage.0/@packagedElement.0/@fragment.5
//@nestedPackage.0/@packagedElement.0/@fragment.6" />
- <fragment xsi:type="SystemFrame" xmi:id="_y_InsLdrEduSi-sHVP1Rgw"
name="RobotRUC.wristmovejoint(NewHalfSteps, Speed)"
systemName="RobotRUC" scenarioName="wristmovejoint(NewHalfSteps,
Speed)">
  <lifeline xsi:type="Component" xmi:id="_APdOQLdsEduSi-sHVP1Rgw"
name="Robot4U4Cnct" coveredBy="_fZBZMLdqEduSi-sHVP1Rgw _IzM0ILdwEduSi-
sHVP1Rgw _IXfIgLdwEduSi-sHVP1Rgw _4PMkYLdxEduSi-sHVP1Rgw
_FL_yALdyEduSi-sHVP1Rgw" />
  <fragment xsi:type="MessageOccurrenceSpecification"
xmi:id="_fZBZMLdqEduSi-sHVP1Rgw" name="MOS-Robot4U4Cnct-1-1-in"
covered="_APdOQLdsEduSi-sHVP1Rgw" event="_z4e4sLIUEduXkqb1bjAvKA"
message="_gLvqILa7Edufg-VapKE-6g" />
  <fragment xsi:type="MessageOccurrenceSpecification"
xmi:id="_4PMkYLdxEduSi-sHVP1Rgw" name="MOS-Robot4U4Cnct-1-1-out"
covered="_APdOQLdsEduSi-sHVP1Rgw" event="_A-rS4LIVEduXkqb1bjAvKA"
message="//@nestedPackage.0/@packagedElement.0/@message.2" />
  <fragment xsi:type="MessageOccurrenceSpecification" name="MOS-
Robot4U4Cnct-1-2-in" event="//@nestedPackage.0/@packagedElement.19"
message="//@nestedPackage.0/@packagedElement.0/@message.5" />
  <fragment xsi:type="MessageOccurrenceSpecification" name="MOS-
Robot4U4Cnct-1-2-out" event="//@nestedPackage.0/@packagedElement.9"
message="//@nestedPackage.0/@packagedElement.0/@message.6" />
  <fragment xsi:type="ExecutionOccurrenceSpecification"
xmi:id="_IXfIgLdwEduSi-sHVP1Rgw" name="EOS-Robot4U4Cnct-1-start"
covered="_APdOQLdsEduSi-sHVP1Rgw" execution="_FL_yALdyEduSi-sHVP1Rgw"
/>
  <fragment xsi:type="BehaviorExecutionSpecification"
xmi:id="_FL_yALdyEduSi-sHVP1Rgw" name="BES-Robot4U4Cnct-1"
covered="_APdOQLdsEduSi-sHVP1Rgw" start="_IXfIgLdwEduSi-sHVP1Rgw"
finish="_IzM0ILdwEduSi-sHVP1Rgw" />
  <fragment xsi:type="ExecutionOccurrenceSpecification"
xmi:id="_IzM0ILdwEduSi-sHVP1Rgw" name="EOS-Robot4U4Cnct-1-finish"
covered="_APdOQLdsEduSi-sHVP1Rgw" execution="_FL_yALdyEduSi-sHVP1Rgw"
/>
  </fragment>
- <fragment xsi:type="SystemFrame" xmi:id="_TeCPUK-SEdulGdzJuhFZlw"
name="WristSUC.wristmovejoint(newLeftHalfStep, newRightHalfStep,
speed)" systemName="WristSUC"
scenarioName="wristmovejoint(newLeftHalfStep, newRightHalfStep,
speed)">
  <message xmi:id="_mF_B4K-SEdulGdzJuhFZlw"
name="wristmovejoint(newLeftHalfStep, newRightHalfStep, speed)"
receiveEvent="_Wz-WwK-aEdulGdzJuhFZlw" sendEvent="_pVjVMK-
SEdulGdzJuhFZlw" />
  <message xmi:id="_6gMDIK-8EduWeujrBWnIYg" name="MoveOk(OK)"
receiveEvent="_FYXtoK-9EduWeujrBWnIYg" sendEvent="_N-T6oK-
7EduWeujrBWnIYg" />
```

```
  <lifeline xsi:type="Connector" xmi:id="_kfT_AK-SEdulGdzJuhFZlw"
name="WristCnct" coveredBy="_p72gMK-SEdulGdzJuhFZlw _sHU5QK-
SEdulGdzJuhFZlw _pVjVMK-SEdulGdzJuhFZlw _vUNsAK-SEdulGdzJuhFZlw
_FYXtoK-9EduWeujrBWnIYg _OqCGQK-9EduWeujrBWnIYg _v2yNkK-
SEdulGdzJuhFZlw" />
  <lifeline xsi:type="Component" xmi:id="_lZbd4K-SEdulGdzJuhFZlw"
name="WristActuator" coveredBy="_3DOw4K-4EdubZJKMO78Tqg _Wz-WwK-
aEdulGdzJuhFZlw _x0Wz4K-aEdulGdzJuhFZlw _tNKmUK-aEdulGdzJuhFZlw
_yisgkK-aEdulGdzJuhFZlw" />
  <lifeline xsi:type="Component" xmi:id="_k8N9cK-SEdulGdzJuhFZlw"
name="WristSensor" coveredBy="_N-T6oK-7EduWeujrBWnIYg _NsRvcK-
7EduWeujrBWnIYg _JUBgUK-8EduWeujrBWnIYg _nkfxAK_DEduWeujrBWnIYg
_ETWbwK-8EduWeujrBWnIYg" />
  <fragment xsi:type="MessageOccurrenceSpecification" xmi:id="_p72gMK-
SEdulGdzJuhFZlw" name="MOS-WristCnct-1-1-in" covered="_kfT_AK-
SEdulGdzJuhFZlw" event="_9w5IQLIUEduXkqb1bjAvKA"
message="//@nestedPackage.0/@packagedElement.0/@message.2" />
  <fragment xsi:type="MessageOccurrenceSpecification" xmi:id="_pVjVMK-
SEdulGdzJuhFZlw" name="MOS-WristCnct-1-1-out" covered="_kfT_AK-
SEdulGdzJuhFZlw" event="_BNubgLIVEduXkqb1bjAvKA" message="_mF_B4K-
SEdulGdzJuhFZlw" />
  <fragment xsi:type="MessageOccurrenceSpecification" xmi:id="_Wz-WwK-
aEdulGdzJuhFZlw" name="MOS-WristActuator-1-1-in" covered="_lZbd4K-
SEdulGdzJuhFZlw" event="_-hnakLIUEduXkqb1bjAvKA" message="_mF_B4K-
SEdulGdzJuhFZlw" />
  <fragment xsi:type="MessageOccurrenceSpecification" xmi:id="_3DOw4K-
4EdubZJKMO78Tqg" name="MOS-WristActuator-1-1-out" covered="_lZbd4K-
SEdulGdzJuhFZlw" event="_Beai4LIVEduXkqb1bjAvKA" message="_b-
3mgLdqEduSi-sHVP1Rgw" />
  <fragment xsi:type="MessageOccurrenceSpecification" xmi:id="_NsRvcK-
7EduWeujrBWnIYg" name="MOS-WristSensor-1-1-in" covered="_k8N9cK-
SEdulGdzJuhFZlw" event="__d0kILIUEduXkqb1bjAvKA"
message="//@nestedPackage.0/@packagedElement.0/@message.4" />
  <fragment xsi:type="MessageOccurrenceSpecification" xmi:id="_N-T6oK-
7EduWeujrBWnIYg" name="MOS-WristSensor-1-1-out" covered="_k8N9cK-
SEdulGdzJuhFZlw" event="//@nestedPackage.0/@packagedElement.7"
message="_6gMDIK-8EduWeujrBWnIYg" />
  <fragment xsi:type="MessageOccurrenceSpecification" xmi:id="_FYXtoK-
9EduWeujrBWnIYg" name="MOS-WristCnct-1-2-in" covered="_kfT_AK-
SEdulGdzJuhFZlw" event="//@nestedPackage.0/@packagedElement.18"
message="_6gMDIK-8EduWeujrBWnIYg" />
  <fragment xsi:type="MessageOccurrenceSpecification" xmi:id="_OqCGQK-
9EduWeujrBWnIYg" name="MOS-WristCnct-1-2-out" covered="_kfT_AK-
SEdulGdzJuhFZlw" event="//@nestedPackage.0/@packagedElement.8"
message="//@nestedPackage.0/@packagedElement.0/@message.5" />
  <fragment xsi:type="ExecutionOccurrenceSpecification"
xmi:id="_vUNsAK-SEdulGdzJuhFZlw" name="EOS-WristCnct-1-start"
covered="_kfT_AK-SEdulGdzJuhFZlw" execution="_sHU5QK-SEdulGdzJuhFZlw"
/>
  <fragment xsi:type="BehaviorExecutionSpecification" xmi:id="_sHU5QK-
SEdulGdzJuhFZlw" name="BESwristCnct" covered="_kfT_AK-SEdulGdzJuhFZlw"
start="_vUNsAK-SEdulGdzJuhFZlw" finish="_v2yNkK-SEdulGdzJuhFZlw" />
  <fragment xsi:type="ExecutionOccurrenceSpecification"
xmi:id="_v2yNkK-SEdulGdzJuhFZlw" name="EOS-WristCnct-1-finish"
covered="_kfT_AK-SEdulGdzJuhFZlw" />
  <fragment xsi:type="ExecutionOccurrenceSpecification"
xmi:id="_x0Wz4K-aEdulGdzJuhFZlw" name="EOS-WristActuator-1-start"
```

```
covered="_lZbd4K-SEdulGdzJuhFZlw" execution="_tNKmUK-aEdulGdzJuhFZlw"
/>
  <fragment xsi:type="BehaviorExecutionSpecification" xmi:id="_tNKmUK-
aEdulGdzJuhFZlw" name="BESwristActuator" covered="_lZbd4K-
SEdulGdzJuhFZlw" start="_x0Wz4K-aEdulGdzJuhFZlw" finish="_yisgkK-
aEdulGdzJuhFZlw" />
  <fragment xsi:type="ExecutionOccurrenceSpecification"
xmi:id="_yisgkK-aEdulGdzJuhFZlw" name="EOS-WristActuator-1-finish"
covered="_lZbd4K-SEdulGdzJuhFZlw" execution="_tNKmUK-aEdulGdzJuhFZlw"
/>
  <fragment xsi:type="ExecutionOccurrenceSpecification"
xmi:id="_JUBgUK-8EduWeujrBWnIYg" name="EOS-WristSensor-1-start"
covered="_k8N9cK-SEdulGdzJuhFZlw" execution="_ETWbwK-8EduWeujrBWnIYg"
/>
  <fragment xsi:type="BehaviorExecutionSpecification" xmi:id="_ETWbwK-
8EduWeujrBWnIYg" name="BESwristSensor" covered="_k8N9cK-
SEdulGdzJuhFZlw" finish="_JUBgUK-8EduWeujrBWnIYg" />
  <fragment xsi:type="ExecutionOccurrenceSpecification"
xmi:id="_nkfxAK_DEduWeujrBWnIYg" name="EOS-WristSensor-1-finish"
covered="_k8N9cK-SEdulGdzJuhFZlw" execution="_ETWbwK-8EduWeujrBWnIYg"
/>
  </fragment>
  <fragment xsi:type="MessageOccurrenceSpecification" name="MOS-
HOperator-1-1-out" covered="_WYfz4LdqEduSi-sHVP1Rgw"
event="_uEs8MLIUEduXkqb1bjAvKA" message="_WSkjULdwEduSi-sHVP1Rgw" />
  <fragment xsi:type="MessageOccurrenceSpecification" name="MOS-
Operator-1-1-in"
covered="//@nestedPackage.0/@packagedElement.0/@lifeline.1"
event="_x2P7MLIUEduXkqb1bjAvKA" message="_WSkjULdwEduSi-sHVP1Rgw" />
  <fragment xsi:type="MessageOccurrenceSpecification"
xmi:id="_o6h_MLa7Edufg-VapKE-6g" name="MOS-Operator-1-1-out"
covered="//@nestedPackage.0/@packagedElement.0/@lifeline.1"
event="_y_3OoLIUEduXkqb1bjAvKA" message="_gLvqILa7Edufg-VapKE-6g" />
  <fragment xsi:type="MessageOccurrenceSpecification" name="MOS-Wrist-
1-1-in" covered="//@nestedPackage.0/@packagedElement.0/@lifeline.2"
event="_-_3c0LIUEduXkqb1bjAvKA" message="_b-3mgLdqEduSi-sHVP1Rgw" />
  <fragment xsi:type="MessageOccurrenceSpecification" name="MOS-Wrist-
1-1-out" covered="//@nestedPackage.0/@packagedElement.0/@lifeline.2"
event="_HjCXwLIVEduXkqb1bjAvKA"
message="//@nestedPackage.0/@packagedElement.0/@message.4" />
  <fragment xsi:type="MessageOccurrenceSpecification"
xmi:id="_pbdiALa7Edufg-VapKE-6g" name="MOS-Operator-1-2-in"
covered="//@nestedPackage.0/@packagedElement.0/@lifeline.1"
event="//@nestedPackage.0/@packagedElement.20"
message="//@nestedPackage.0/@packagedElement.0/@message.6" />
  <fragment xsi:type="MessageOccurrenceSpecification"
xmi:id="_i9wEMLdqEduSi-sHVP1Rgw" name="MOS-Operator-1-2-out"
covered="_WYfz4LdqEduSi-sHVP1Rgw"
event="//@nestedPackage.0/@packagedElement.10"
message="//@nestedPackage.0/@packagedElement.0/@message.7" />
  <fragment xsi:type="MessageOccurrenceSpecification" name="MOS-
HOperator-1-2-in" covered="_WYfz4LdqEduSi-sHVP1Rgw"
event="//@nestedPackage.0/@packagedElement.21"
message="//@nestedPackage.0/@packagedElement.0/@message.7" />
  <fragment xsi:type="ExecutionOccurrenceSpecification" name="EOS-
HOperator-1-start"
execution="//@nestedPackage.0/@packagedElement.0/@fragment.11" />
```

```
  <fragment xsi:type="BehaviorExecutionSpecification" name="BES-
HOperator-1" covered="_WYfz4LdqEduSi-sHVP1Rgw"
finish="//@nestedPackage.0/@packagedElement.0/@fragment.12" />
  <fragment xsi:type="ExecutionOccurrenceSpecification" name="EOS-
HOperator-1-finish" covered="_WYfz4LdqEduSi-sHVP1Rgw"
execution="//@nestedPackage.0/@packagedElement.0/@fragment.11" />
  <fragment xsi:type="ExecutionOccurrenceSpecification" name="EOS-
Operator-1-start"
covered="//@nestedPackage.0/@packagedElement.0/@lifeline.1"
execution="//@nestedPackage.0/@packagedElement.0/@fragment.14" />
  <fragment xsi:type="BehaviorExecutionSpecification" name="BES-
Operator-1"
covered="//@nestedPackage.0/@packagedElement.0/@lifeline.1"
start="//@nestedPackage.0/@packagedElement.0/@fragment.13"
finish="//@nestedPackage.0/@packagedElement.0/@fragment.15" />
  <fragment xsi:type="ExecutionOccurrenceSpecification" name="EOS-
Operator-2-finish"
execution="//@nestedPackage.0/@packagedElement.0/@fragment.14" />
  <fragment xsi:type="ExecutionOccurrenceSpecification" name="EOS-
Wrist-1-start"
covered="//@nestedPackage.0/@packagedElement.0/@lifeline.2"
execution="//@nestedPackage.0/@packagedElement.0/@fragment.17" />
  <fragment xsi:type="BehaviorExecutionSpecification" name="BES-Wrist-
1" covered="//@nestedPackage.0/@packagedElement.0/@lifeline.2"
start="//@nestedPackage.0/@packagedElement.0/@fragment.16"
finish="//@nestedPackage.0/@packagedElement.0/@fragment.18" />
  <fragment xsi:type="ExecutionOccurrenceSpecification" name="EOS-
Wrist-1-finish"
covered="//@nestedPackage.0/@packagedElement.0/@lifeline.2"
execution="//@nestedPackage.0/@packagedElement.0/@fragment.17" />
  </packagedElement>
  <packagedElement xsi:type="SendOperationEvent" name="out1" />
  <packagedElement xsi:type="SendOperationEvent" name="out2" />
  <packagedElement xsi:type="SendOperationEvent" name="out3" />
  <packagedElement xsi:type="SendOperationEvent" name="out4" />
  <packagedElement xsi:type="SendOperationEvent" name="out5" />
  <packagedElement xsi:type="SendOperationEvent" name="out6" />
  <packagedElement xsi:type="SendOperationEvent" name="out7" />
  <packagedElement xsi:type="SendOperationEvent" name="out8" />
  <packagedElement xsi:type="SendOperationEvent" name="out9" />
  <packagedElement xsi:type="SendOperationEvent" name="out10" />
  <packagedElement xsi:type="SendOperationEvent" name="out11" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in1" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in2" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in3" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in4" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in5" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in6" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in7" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in8" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in9" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in10" />
  <packagedElement xsi:type="ReceiveOperationEvent" name="in11" />
  </nestedPackage>
  </Package>
```

D.2 Generated PRISMA Specification

In the following is described the PRISMA Specification automatically generated using ModelMorf, the set of Relations described in Appendix B and using as input the ATRIUM scenario described in the previous section.

```
  <?xml version="1.0" encoding="ASCII" ?>
- <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:PRISMA="PRISMA">
- <PRISMA:System
xmi:id="PRISMA.System.d05a152f445dfced706f0a043a85ff7ceb814d1decf3b4cd
b74e3bf75f37c1a2" name="RobotRUC"
containsComps="PRISMA.Component.01c0d52bfc7f41eafcf0f411954510fedefbb0
9c3b4811fa374dce933f6c5eec">
  <has
xmi:id="PRISMA.Port.367a2dfe895e6beac198ce14a994901aa0f0abf3b2278f3042
a5f389f12c9237" name="WristSUCWristCnct"
isComposed="PRISMA.Binding.b361f311a6d65d68-b9d3ab0d5c187760" />
  <has
xmi:id="PRISMA.Port.489b2dfa789e6beac167ce14a556901bb1f0abf3b2278f3042
a5f389a12b3423" name="Operator"
isComposed="PRISMA.Binding.f462f461b6f67d68-bad3bc0d5c187890" />
  <connect
xmi:id="PRISMA.Attachment.3b9190527eaa48f670fb713902e7411ca0cc5ac2d176
4d5bccc8fde200eedd0a" name="RobotRUCRobot4U4CnctWristSUCWristCnct"
linkPort="PRISMA.Port.367a2dfe895e6beac198ce14a994901aa0f0abf3b2278f30
42a5f389f12c9237
PRISMA.Port.de583f4a8341c3a165748bae1e9a4f9f547013d0f24d613f785f1d035b
214456" />
  </PRISMA:System>
  <PRISMA:Binding xmi:id="PRISMA.Binding.b361f311a6d65d68-
b9d3ab0d5c187760" name="RobotRUCRobot4U4Cnct"
ARPort="PRISMA.Port.5c9ab0d99591e47b5e7df0d3c2e2635581f1f3862a415a7cb1
3cbbc26e5c62e1"
SystemPort="PRISMA.Port.367a2dfe895e6beac198ce14a994901aa0f0abf3b2278f
3042a5f389f12c9237" />
  <PRISMA:Binding xmi:id="PRISMA.Binding.f462f461b6f67d68-
bad3bc0d5c187890" name="RobotRUCOperator"
ARPort="PRISMA.Port.5c9ab0d99591e47b5e7df0d3c2e2635581f1f3862a415a7cb1
3cbbc26e5c62e1"
SystemPort="PRISMA.Port.d05a152f445dfced706f0a043a85ff7ceb814d1decf3b4
cdb74e3bf75f37c1a2" />
- <PRISMA:Component
xmi:id="PRISMA.Component.01c0d52bfc7f41eafcf0f411954510fedefbb09c3b481
1fa374dce933f6c5eec" name="Robot4U4Cnct"
imports="PRISMA.Aspect.01c0d52bfc7f41eafcf0f411954510fedefbb09c3b4811f
a374dce933f6c5eec">
  <has
xmi:id="PRISMA.Port.5c9ab0d99591e47b5e7df0d3c2e2635581f1f3862a415a7cb1
3cbbc26e5c62e1" name="WristSUCWristCnct"
isComponent="PRISMA.Binding.b361f311a6d65d68-b9d3ab0d5c187760" />
  <has
xmi:id="PRISMA.Port.6cfaa0f9a5617435a7df1d3f2e5637584f1a3872a616a7cb13
```

```
cbbc56e5a34a7" name="Operator"
isComponent="PRISMA.Binding.f462f461b6f67d68-bad3bc0d5c187890" />
  </PRISMA:Component>
- <PRISMA:Aspect
xmi:id="PRISMA.Aspect.01c0d52bfc7f41eafcf0f411954510fedefbb09c3b4811fa
374dce933f6c5eec" name="CoorRobot4U4Cnct" concern="Coordination">
  <belongsTo xmi:id="PRISMA.Service.19579f27cc91cd66-ec7c5c354cc006f5"
name="begin()" />
  <belongsTo xmi:id="PRISMA.Service.86c2de34edc9f9dd-6ad4ed8a7ab2272b"
name="end()" />
  <belongsTo xmi:id="PRISMA.Service.c1d8cfea3f8840de-f0fe47983f14b21f"
name="MoveOk(OK)" type="inout" />
  <belongsTo xmi:id="PRISMA.Service.aebe2013a3eae286-d0574cfe9509d48c"
name="wristmovejoint(newLeftHalfStep, newRightHalfStep, speed"
type="inout" />
  </PRISMA:Aspect>
- <PRISMA:System
xmi:id="PRISMA.System.abd7d05b397de29648a3f73964410b3b55ae98553f23e624
46440892f54ab8c5" name="WristSUC"
containsComps="PRISMA.Component.505edf42ce0978ba691699aa5e020e4717d33f
dc007f20cf5588a4ab96b785f1
PRISMA.Component.bc6bc31975cb45e0aa92cb82072e914a5866883e5fde790044316
0d2fa93d136"
containsCnct="PRISMA.Connector.ac6f0c7950ebbf55df67f5f9fc45fdfe5764dd5
fce987c21bcff9348f4529e56">
  <has
xmi:id="PRISMA.Port.de583f4a8341c3a165748bae1e9a4f9f547013d0f24d613f78
5f1d035b214456" name="RobotRUCRobot4U4Cnct"
isComposed="PRISMA.Binding.6c92a297ee39d19d-524620d0b7880c66" />
  <connect
xmi:id="PRISMA.Attachment.4bf9d629df16ad3410f5d32e01c1f5647d6a2d7cb50e
b0ab3ee9bfccb440a03a" name="WristCnctWristActuator"
linkPort="PRISMA.Port.ab80097c7efdfc6a861460fb4e02df5582d59914ae38be43
8ff0fcdec55ad60b
PRISMA.Port.636238c2d2ed6b0978f2e6e946a07c2f400569287dd0cc02c019a51aaa
68dff9" />
  <connect
xmi:id="PRISMA.Attachment.2a46ac5b7063420c47a91c002387342526741c8cfe1f
b9208b695f0b5ac56e22" name="WristCnctWristSensor"
linkPort="PRISMA.Port.bd3953804692832e7662acc726a002700414ebeaecd36d7b
f24a1aa0f1f097d1
PRISMA.Port.ca52019b6b0062aa165c6f29fa69efffac5ee2a76998556cd03ecfe0bc
a7f4f6" />
  </PRISMA:System>
  <PRISMA:Binding xmi:id="PRISMA.Binding.6c92a297ee39d19d-
524620d0b7880c66" name="WristCnct"
ARPort="PRISMA.Port.2839f6179ed31c7d7a67ebb4b361b0cd03d405103c9ef2e134
86e15df53b7269"
SystemPort="PRISMA.Port.de583f4a8341c3a165748bae1e9a4f9f547013d0f24d61
3f785f1d035b214456" />
- <PRISMA:Connector
xmi:id="PRISMA.Connector.ac6f0c7950ebbf55df67f5f9fc45fdfe5764dd5fce987
c21bcff9348f4529e56" name="WristCnct"
imports="PRISMA.Aspect.ac6f0c7950ebbf55df67f5f9fc45fdfe5764dd5fce987c2
1bcff9348f4529e56">
  <has
xmi:id="PRISMA.Port.2839f6179ed31c7d7a67ebb4b361b0cd03d405103c9ef2e134
```

```
86e15df53b7269" name="RobotRUCRobot4U4Cnct"
isComponent="PRISMA.Binding.6c92a297ee39d19d-524620d0b7880c66" />
  <has
xmi:id="PRISMA.Port.636238c2d2ed6b0978f2e6e946a07c2f400569287dd0cc02c0
19a51aaa68dff9" name="WristActuator" />
  <has
xmi:id="PRISMA.Port.ca52019b6b0062aa165c6f29fa69efffac5ee2a76998556cd0
3ecfe0bca7f4f6" name="WristSensor" />
  </PRISMA:Connector>
- <PRISMA:Aspect
xmi:id="PRISMA.Aspect.ac6f0c7950ebbf55df67f5f9fc45fdfe5764dd5fce987c21
bcff9348f4529e56" name="CoorWristCnct" concern="Coordination">
  <belongsTo xmi:id="PRISMA.Service.8b8dae473afcf85c-247a0e994b9f5ff6"
name="begin()" />
  <belongsTo xmi:id="PRISMA.Service.2780fd6ce1fc6560-62f67ba515fd186f"
name="end()" />
  <belongsTo xmi:id="PRISMA.Service.91dd3432e7b602be-1c621974c1d84db1"
name="MoveOk(OK)" type="inout" />
  <belongsTo xmi:id="PRISMA.Service.6488fa61843ecae1-72f32c49d924ac17"
name="wristmovejoint(newLeftHalfStep, newRightHalfStep, speed"
type="inout" />
  </PRISMA:Aspect>
- <PRISMA:Component
xmi:id="PRISMA.Component.505edf42ce0978ba691699aa5e020e4717d33fdc007f2
0cf5588a4ab96b785f1" name="WristActuator"
imports="PRISMA.Aspect.505edf42ce0978ba691699aa5e020e4717d33fdc007f20c
f5588a4ab96b785f1">
  <has
xmi:id="PRISMA.Port.ab80097c7efdfc6a861460fb4e02df5582d59914ae38be438f
f0fcdec55ad60b" name="WristCnct" />
  </PRISMA:Component>
- <PRISMA:Aspect
xmi:id="PRISMA.Aspect.505edf42ce0978ba691699aa5e020e4717d33fdc007f20cf
5588a4ab96b785f1" name="WristActuator" concern="Functional">
  <belongsTo xmi:id="PRISMA.Service.7ef81bbeb15c0d9c-65802fd7c3a41635"
name="begin()" />
  <belongsTo xmi:id="PRISMA.Service.84f213be16689212-85155286b8879fdc"
name="end()" />
  <belongsTo xmi:id="PRISMA.Service.d352b0ec59ce9398-6ebe031c334720a6"
name="wristmovejoint(newLeftHalfStep, newRightHalfStep, speed)"
type="inout" />
  </PRISMA:Aspect>
- <PRISMA:Component
xmi:id="PRISMA.Component.bc6bc31975cb45e0aa92cb82072e914a5866883e5fde7
900443160d2fa93d136" name="WristSensor"
imports="PRISMA.Aspect.bc6bc31975cb45e0aa92cb82072e914a5866883e5fde790
0443160d2fa93d136">
  <has
xmi:id="PRISMA.Port.bd3953804692832e7662acc726a002700414ebeaecd36d7bf2
4a1aa0f1f097d1" name="WristCnct" />
  </PRISMA:Component>
- <PRISMA:Aspect
xmi:id="PRISMA.Aspect.bc6bc31975cb45e0aa92cb82072e914a5866883e5fde7900
443160d2fa93d136" name="WristSensor" concern="Functional">
  <belongsTo xmi:id="PRISMA.Service.294ef583f786645b-7723670b0cf529d7"
name="begin()" />
```

```
  <belongsTo xmi:id="PRISMA.Service.2d8f6e6d283c8705-bd04adb8728ff1e7"
name="end()" />
  <belongsTo xmi:id="PRISMA.Service.435105467f982732-4f601d3b30c19b3b"
name="MoveOk(OK)" type="inout" />
  </PRISMA:Aspect>
- <PRISMA:System
xmi:id="PRISMA.System.d05a152f445dfced706f0a043a85ff7ceb814d1decf3b4cd
b74e3bf75f37c1a2" name=""
containsComps="PRISMA.Component.bc6bc31975cb45e0aa92cb82072e914a586688
3e5fde7900443160d2fa93d136">
  <connect
xmi:id="PRISMA.Attachment.3b9190527eaa48f670fb713902e7411ca0cc5ac2d176
4d5bccc8fde200eedd0a" name="OperatorRobotRUCRobot4U4Cnct"
linkPort="PRISMA.Port.bd3953804692832e7662acc726a002700414ebeaecd36d7b
f24a1aa0f1f097d1
PRISMA.Port.489b2dfa789e6beac167ce14a556901bb1f0abf3b2278f3042a5f389a1
2b3423" />
  </PRISMA:System>
- <PRISMA:Component
xmi:id="PRISMA.Component.bc6bc31975cb45e0aa92cb82072e914a5866883e5fde7
900443160d2fa93d136" name="Operator"
imports="PRISMA.Aspect.bc6bc31975cb45e0aa92cb82072e914a5866883e5fde790
0443160d2fa93d136">
  <has
xmi:id="PRISMA.Port.bd3953804692832e7662acc726a002700414ebeaecd36d7bf2
4a1aa0f1f097d1" name="WristCnct" />
  </PRISMA:Component>
- <PRISMA:Aspect
xmi:id="PRISMA.Aspect.bc6bc31975cb45e0aa92cb82072e914a5866883e5fde7900
443160d2fa93d136" name="PresOperator" concern="Presentation">
  <belongsTo xmi:id="PRISMA.Service.d708ef8fbec9707b-26b1e0c2214fa032"
name="begin()" />
  <belongsTo xmi:id="PRISMA.Service.da9d0e1c09e798cd-34f6b1d147a089f1"
name="end()" />
  <belongsTo xmi:id="PRISMA.Service.dd0ad62fd06a6eca-168c3c1564fd1d5f"
name="wristmovejoint(newLeftHalfStep, newRightHalfStep, speed"
type="inout" />
  <belongsTo xmi:id="PRISMA.Service.7fb390edb7ff96fe-ba5f8384fb3451c"
name="MoveOk(OK)" type="inout" />
  </PRISMA:Aspect>
  </xmi:XMI>
```

D.3 PRISMA ADL GENERATED

In the following is described the result of the transformation to textual PRISMA ADL performed using MORPHEUS using as input the previous specification.

```
Interface InterfacePRISMA1
      wristmovejoint(newLeftHalfStep, newRightHalfStep, speed);
      MoveOk(OK);
End_Interface InterfacePRISMA1;
```

```
Interface InterfacePRISMA2
      wristmovejoint(newLeftHalfStep, newRightHalfStep, speed);
End_Interface InterfacePRISMA2;

Interface InterfacePRISMA3
      MoveOk(OK);
End_Interface InterfacePRISMA3;

Presentation Aspect PresOperator using
  Services
      Begin();
      End();
      in/out wristmovejoint(newLeftHalfStep, newRightHalfStep, speed);
      in/out MoveOk(OK);
End Presentation Aspect PresOperator;

Coordination Aspect CoorRobot4U4Cnct using
  Services
      Begin();
      End();
      in/out wristmovejoint(newLeftHalfStep, newRightHalfStep, speed);
      in/out MoveOk(OK);
End Coordination Aspect CoorRobot4U4Cnct;

Coordination Aspect CoorWristCnct using
  Services
      Begin();
      End();
      in/out wristmovejoint(newLeftHalfStep, newRightHalfStep, speed);
      in/out MoveOk(OK);
End Coordination Aspect CoorWristCnct;

Coordination Aspect FuncWristSensor using
  Services
      Begin();
      End();
      in/out MoveOk(OK);
End Coordination Aspect FuncWristSensor;

Coordination Aspect FuncWristActuator using
  Services
      Begin();
      End();
      in/out wristmovejoint(newLeftHalfStep, newRightHalfStep, speed);
End Coordination Aspect FuncWristActuator;


End_Component «Functional»FuncWristActuator;

Component Operator
  Ports
       PortPRISMA1: InterfacePRISMA1;
  End_Ports;
  Presentation Aspect Import PresOperator;
End_Component Operator;
```

```
Component WristActuator
  Ports
       PortPRISMA2: InterfacePRISMA2;
  End_Ports;
  Functional Aspect Import FuncWristActuator;
End_Component WristActuator;

Component WristSensor
  Ports
       PortPRISMA3: InterfacePRISMA3;
  End_Ports;
  Functional Aspect Import FuncWristSensor;
End_Component WristSensor;

End_Connector «ComponentPRISMA»WristSensor;

End_Connector «ComponentPRISMA»WristSensor;

Connector Robot4U4Cnct
  Roles
       RolPRISMA2: InterfacePRISMA1;
       RolPRISMA1: InterfacePRISMA1;
  End_Roles;
  Coordination Aspect Import CoorRobot4U4Cnct;
End_Connector Robot4U4Cnct;

Connector WristCnct
  Roles
       RolPRISMA4: InterfacePRISMA1;
       RolPRISMA5: InterfacePRISMA2;
       RolPRISMA3: InterfacePRISMA3;
  End_Roles;
  Coordination Aspect Import CoorWristCnct;
End_Connector WristCnct;

System RobotRUC
  Ports
      PortPRISMA5: InterfacePRISMA1;
      PortPRISMA4: InterfacePRISMA1;
  End_Ports;
  Import_Types
  End_Import_Types;
  Instances
  End_Instances;
  Attachments
  End_Attachments;
  Bindings
      RobotRUC.PortPRISMA4 <-->
ConnectorPRISMA»Robot4U4Cnct.RolPRISMA2;
      RobotRUC.PortPRISMA5 <-->
ConnectorPRISMA»Robot4U4Cnct.RolPRISMA1;
  End_Bindings;
End_System RobotRUC;

System WristSUC
  Ports
      PortPRISMA6: InterfacePRISMA1;
  End_Ports;
  Import_Types
  End_Import_Types;
  Instances
  End_Instances;
  Attachments
      ComponentPRISMA»WristSensor.PortPRISMA3 <-->
ConnectorPRISMA»WristCnct.RolPRISMA5;
      ComponentPRISMA»WristActuator.PortPRISMA2 <-->
ConnectorPRISMA»WristCnct.RolPRISMA4;
  End_Attachments;
  Bindings
      ConnectorPRISMA»WristCnct.RolPRISMA3 <--> WristSUC.PortPRISMA6;
  End_Bindings;
End_System WristSUC;
```