# Reflective and Adaptive Middleware for Software Evolution of Information Systems

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur

(Dr.-Ing.)

angenommen durch die Fakultät für Informatik

der Otto-von-Guericke-Universität Magdeburg

von

M.Sc. Ahmed Mohamed Ali Ghoneim

geboren am 2. März 1972

in El-Menoufiya, Egypt

Gutachter:

Prof.Dr. Gunter Saake

Prof.Dr. Claus Rautenstrauch

Prof.Dr. Walter Cazzola

Promotionskolloquium am 16.02.2007

Abstract of the Dissertation
by
Ahmed Ghoneim

# Abstract

The high volatility and competitively of organizational ('socio-techno-economical') environment are putting tremendous pressure on software-intensive developers to come up with adaptive and evolving approaches. To contribute to such topical efforts towards adaptive information system at early phases, the present thesis addresses the rigorous development of self-adapting information systems. The approach we are putting forwards is referred to as RAMSES (Reflective and Adaptive Middleware for Software Evolution of non-stopping information Systems), and provides a reflective architecture for adapting the software applications, in response the requirements and environmental changes.

The reflective middleware is based on a UML-compliant base- and meta-level. RAMSES provides objects with the ability of dynamically changing their behavior by exploiting their design information. The meta-level of the proposed architecture reifies UML diagrams including structural and behavioral information of the system to adapt; then it uses such data for dynamically adapting the software system against environmental changes. The evolution takes place in two steps: a meta-object, called evolutionary meta-object, plans a possible evolution against the detected external events then another meta-object, called consistency checker meta-object validates the feasibility of the proposed plan before really evolving the system. The meta-objects use the system design information to lead its evolution.

Our middleware uses *reification library* to explicit an abstract view of the concrete-level that can be manipulated at run-time. Both evolutionary and consistency checker meta-objects work directly on the reifications. The evolutionary meta-object steers the evolution of reifications through evolutionary rules that describe the changes in environment. Where as the consistency checker meta-object check the reifications are consistent with the changes. To evaluate RAMSES and validate our claims, a number of examples of urban traffic control system (UTCS) are used.

# Abstract der Dissertation - deutsche Fassung

Schnelle Veränderungen und harter Konkurrenzkampf sind heute bestimmende Faktoren in der Software-Entwicklung. Aus diesem Grund wird der Ruf nach anpassungsfähigen und erweiterbaren Systemen immer lauter. Gerade die Domäne der informationsverarbeitenden Systeme unterliegt ständigen Schwankungen. Ziel ist es daher, schon in frühen Entwicklungsphasen Techniken einfliessen zu lassen, die der Forderung nach (weitestgehend automatisierter) Adaptivität und Erweiterbarkeit gerecht werden. Der in dieser Dissertation beschriebene Ansatz trägt den Titel RAMSES (Reflective and Adaptive Middleware for Software Evolution of non-stopping information Systems). Die Basis bildet eine Architektur, die das Mittel der (Selbst-)Reflektion nutzt, um dem Gedanken der Anpassbarkeit und Erweiterbarkeit gerecht zu werden.

Die reflektive Middleware bedient sich verschiedener UML-Konstrukte, die in der Basis- und Meta-Ebene wirken. RAMSES etabliert Objekte, die die Fähigkeit zur dynamischen Änderung ihres Verhaltens besitzen. Die Änderung geschieht unter Ausnutzung ihrer Design-Informationen. Die Meta-Ebene der Basis-Applikation verarbeitet (auch bezeichnet als "Verdinglichung", engl.: Reification) geeignete UML-Diagramme (Repräsentanten für Struktur und Verhalten der Basis-Software). Die generierten Daten werden dann für den Prozess der dynamischen Adaption (als Reaktion auf Änderungen in der Laufzeit-Umgebung) des Programms genutzt. Die Anpassung (Weiterentwicklung) erfolgt in zwei Phasen. Ein Meta-Objekt ("Evolutionary Meta-Object") plant eine geeignete Anpassungvariante gegen die aufgetretene Veränderung in der Umwelt. Ein zweites Meta-Objekt ("Consistency Checker Meta-Object") validiert die Machbarkeit/Korrektheit der geplanten Änderungen, bevor diese konkret werden. Die beiden Ausführungsschritte erfolgen unter Verwendung der Entwurfsinformationen.

Unsere Middleware nutzt "Reification"-Bibliotheken um die Manipulation der Basis-Applikation auf einem abstrakten Level (losgelöst von der Quellcode-Ebene) durchführen zu können. Sowohl "Evolutionary Meta-Object", als auch "Consistency Checker Meta-Object" arbeiten beide auf den generierten ("verdinglichten") Informationen. Die "Evolutionary Meta-Objects" planen die nötigen Anpassungen unter Berücksichtigung von "Evolutionary Rules" (Lösungsstrategien). Die "Consistency Checker Meta-Objekte" prüfen die Konsistenz der Anpassungen (mittels Validierungsregeln). Auf der Basis verschiedener Beispiel-Implementierungen des "Urban Traffic Control System" (UTCS) wird RAMSES analysiert und hinsichtlich der vorher definierten Zielstellung bewertet.

# Acknowledgments

This dissertation, while an achievement that bears my name, would not have been possible without the help of others, who I would now like to thank.

I wish to acknowledge high indebtedness and my deep gratitude to my good natured and devoted supervisor Prof. Dr. Gunter Saake for introducing me to the interesting field of Software evolution. His generosity to share his insight and ideas with me, was the starting point for the work of my research project. Due to financial support out of his grants, I was able to participate in conferences. During my stay, I found Prof. Gunter Saake and his family very hospitable and helpful.

I am highly indebted to my affectionate co-supervisor Dr. Walter Cazzola for giving me an initiative to this research. His inspiring guidance, remarkable suggestions, constructive criticism and friendly discussions enabled me to complete the research work and this thesis efficiently. He spared a lot of his precious time in advising and helping me throughout the research work.

I am overwhelmed with gratitude for the unconditional support provided by Prof. Claus Rautenstrauch a long these five years.

I am grateful thanks go to all colleague of Database Group, who were always quite helpful during my defense and organizing the party.

To my family for their love and support. My most grateful thanks go to my parents and my brothers. My most grateful thanks go to my wife (Wesam) and my kids (Mahmoud and Rahma). They have been especially patient and understanding during what was anticipated to be a five year process, though has now ended at five years.

Magdeburg, Germany

March 6, 2007

*Ahmed Ghoneim*

# Contents

**List of Figures** ix

**List of Tables** xi

**1 Introduction** 1
- 1.1 Thesis Statement . . . 1
- 1.2 Approach and Contribution of the Dissertation . . . 3
  - 1.2.1 Approach . . . 3
  - 1.2.2 Contribution . . . 4
- 1.3 The RAMSES Middleware: General Insight . . . 5
- 1.4 Outline of the Dissertation . . . 6

**2 Preliminaries** 9
- 2.1 Software Engineering Models . . . 9
- 2.2 Software Maintenance and Evolution . . . 11
  - 2.2.1 Software maintenance . . . 11
  - 2.2.2 Software evolution . . . 12
- 2.3 Reflection Terminology in Software Systems . . . 14
  - 2.3.1 Computational reflection . . . 14
  - 2.3.2 Reflection in the object-oriented paradigm . . . 15
  - 2.3.3 Reflection models . . . 16
- 2.4 Current Techniques of Analysis and Evolution of the Software Systems . . . 17
  - 2.4.1 Re-engineering (renovation) . . . 17
  - 2.4.2 Impact analysis techniques . . . 17
  - 2.4.3 Refactoring . . . 19
  - 2.4.4 Slicing object-oriented approaches . . . 19
- 2.5 Object-Oriented Analysis and Design Techniques . . . 20
- 2.6 Summary . . . 23

**3 Design Information: RAMSES Base and Meta Building Stones** 25
- 3.1 Introduction and Motivation . . . 25
- 3.2 The urban traffic control system (UTCS): simplified view . . . 26
- 3.3 Explicit versus Implicit View . . . 28
  - 3.3.1 Implicit behavior . . . 28

3.3.2 Explicit behavior . . . 33
3.4 Lightweight Formalisation of Design Information . . . 36
3.4.1 Structural design information . . . 37
3.4.2 Behavioral design information . . . 39
3.5 Summary . . . 42

**4 Evolution Planning: RAMSES Strategic Processes 45**
4.1 Introduction and motivation . . . 45
4.2 UML Diagrams as Meta-Data . . . 46
4.3 Evolution and Validation Planning . . . 49
4.3.1 Evolutionary planning of the UTCS example (3.2) . . . 52
4.3.2 Consistency validation of the UTCS example (3.2) . . . 56
4.4 Operation-Based Adaptation of Design Information . . . 57
4.4.1 Operations taxonomy of structural design information . . . 58
4.4.2 Operations taxonomy of behavioral design information . . . 59
4.5 Interpreting the Evolution by Using Script Language . . . 60
4.6 Summary . . . 64

**5 The Reflective Middleware: RAMSES At Work 65**
5.1 Software Evolution through Reflection . . . 66
5.1.1 The reflective architecture . . . 67
5.1.2 Decisional engines and evolutionary rule sets . . . 69
5.2 Reification and Reflection by Using Design Information . . . 71
5.3 Describing the Meta-level Behavior of RAMSES . . . 73
5.3.1 Evolutionary meta-object . . . 75
5.3.2 Consistency checker meta-object . . . 80
5.4 Summary . . . 84

**6 The UTCS: a Case Study 87**
6.1 The Specification and Components of the UTCS . . . 87
6.1.1 UTCS components . . . 88
6.1.2 Specification of the UTCS . . . 89
6.2 UTCS Cases . . . 90
6.2.1 Case (A): closing a lane or part of lane . . . 90
6.2.2 Normal layout for case (B): an overview . . . 92
6.2.3 Normal layout for case (C): an overview . . . 95
6.3 Design Information Realization for Case (A) . . . 97
6.3.1 Evolutionary rules . . . 97
6.3.2 Validation rules . . . 103
6.3.3 Samples of script rules for case (A) . . . 104
6.4 Practical Results: A Dynamic Evolution and Validation Prototype . . . 106
6.4.1 The external libraries required . . . 106
6.4.2 The prototype: an overview . . . 107
6.5 Summary . . . 109

**7 Comparison of RAMSES to Related Works** **111**
7.1 Evaluation Criteria . . . 112
7.2 Reflective Approaches and RAMSES . . . 113
7.2.1 The K-Component architecture . . . 114
7.2.2 Architectural reflection . . . 114
7.2.3 Co-operative actions (CO Actions) . . . 115
7.2.4 DART . . . 116
7.2.5 Some comments on the comparison . . . 116
7.3 Analysis&Evolution Approaches and RAMSES . . . 118
7.3.1 Re-engineering tools . . . 118
7.3.2 Impact analysis tools . . . 119
7.3.3 Refactoring tools . . . 119
7.3.4 Assessment against the evolution criteria . . . 120
7.4 Summary . . . 122

**8 Concluding Remarks** **123**
8.1 Recapitulation . . . 123
8.2 Further future work . . . 125
8.2.1 Reflecting the planned evolution by the AOP on the base-system code . . . 125
8.2.2 Formal underpinning . . . 126
8.2.3 Dynamic adaptation with reflective graph-transformations . . . 126

**Bibliography** **129**

**Curriculum Vitae** **139**

# List of Figures


1.1 The thesis outline. . . . . 6

2.1 Classic&Evolutionary software engineering life cycles [32]. . . . . 10
2.2 Reflective tower. . . . . 15
2.3 UML 2.0 diagrams classifications. . . . . 21
2.4 Engineering UML specification. . . . . 22

3.1 City layout: a) the layout during normal activities b) the layout during road maintenance. . . . . 27
3.2 The representation of concrete-level of UTCS. . . . . 30
3.3 The refactor view of the concrete-level of UTCS. . . . . 31
3.4 The slicing and impact analysis view of the concrete-level of UTCS. . . . 32
3.5 Normal behavior for the particular routes of the UTCS case1. . . . . 34
3.6 Class diagram of the urban traffic control system. . . . . 35
3.7 Normal layout: a) the static view . . . . . 36
3.8 Normal layout: the behavior view. . . . . 37
3.9 The taxonomy of design information-abstract level. . . . . 37

4.1 Modified Layout: a) the modified static view b) the modified behavior view. 55

5.1 RAMSES designed for the evolution of software systems. . . . . 67
5.2 The RAMSES meta-behavior using UTCS motivation example 3.1. . . . . 75
5.3 Evolutionary meta-objects for describing the meta-level behavior of RAMSES middleware. . . . . 76
5.4 The application of the evolutionary meta-object in the UTCS . . . . . 80
5.5 The role of consistency checking against evolution of the system structure. 83

6.1 The original map for Berlin city . . . . . 91
6.2 Layout of Case A. . . . . 91
6.3 Proposed evolution of Area A: a) adapted layout according first plan b) adapted layout according second plan. . . . . 93
6.4 Layout of Case B. . . . . 93
6.5 Proposed evolution of area B: a) adapted layout according first plan b) adapted layout according second plan. . . . . 94
6.6 Case (C): (a) normal layout, (b) emergency plan. . . . . 96

6.7 Class diagram for the normal layout. . . . . 98
6.8 Deployment diagram: a) object instances connection at traffic node (tn1)
b) object instances connection at traffic node (tn2). . . . . 99
6.9 Adapted deployment diagram realized from first plan at figure 6.3.a . . . 100
6.10 TwoGroupsSync with 4tf . . . . . 101
6.11 TwoGroupsSync with 5tf . . . . . 102
6.12 The evolution prototype interface. . . . . 107

# List of Tables

4.1 Operation categories for changes in class diagrams . . . . . 58
4.2 Operation categories for changes in deployment diagram . . . . . 59
4.3 Operations categories for changes in statechart diagram . . . . . 59
4.4 Operations categories for changes in sequence diagram . . . . . 60
4.5 Operations categories for changes in activity diagram . . . . . 61

2.1 Comparison of RAMSES with reflective architectures . . . . . 117
3.1 Comparison of RAMSES with analysis and evolution tools . . . . . 121

# 1 Introduction

Software systems in general and information systems in particular need more than ever to dynamically adapt against unanticipated requirement changes. The main scope of software evolution consists of rendering a system adaptable to any environmental changes. In the traditional software life cycle, as we illustrate in the next chapter, software evolution belongs to the last phase of maintenance, where incremental changes in the software are made.

This chapter purposes to shed some lights on the problem of the dynamic software evolution. We then detail the objectives of the thesis. The RAMSES middleware is then motivated and highlighted as an approach to solve the problems of software evolution. Finally, a roadmap for the remaining chapters of this thesis is introduced.

## 1.1. Thesis Statement

Information systems can easily be approached as being composed of many interacting components, specially as they are more and more supporting collaborating inter-organization. Their planning usually involve designer, planner, manager and programmers, among other stakeholders. The livability of the software systems means they should be able to face the challenges in their environments and requirements. As requirements, technology and business rapidly and unanticipatedly change, information systems should be redesigned and extended by new functionalities to timely face competitively. Such adaptation and evolution addressed on the bases of well defined criteria (time-dependent).

Software systems are thus expecting for mechanisms to face changes in their environment and be able to self-adapt their code and design models when unanticipated events occur. The UML is de facto the standard (graphical) language used during the design process, therefore its diagrams are considered as a good representation for the system design [11]. Dynamic events are hard to be captured at design-time whereas their occurrence surely affects also the design information.

Runtime evolution considers the case where the changes are made dynamically. Here, systems evolve dynamically for instance by changing functionalities of some classes, subsystems or components, hot-swapping existing components or by integrating newly developed classes or components without stopping the whole system. Runtime evolution

has to be either planned ahead explicitly in the system or the underlying platform has to provide means to effectuate software changes dynamically. Recent trends in software engineering research put the evolution as one of the most vivid and topical direction. Approaches and techniques endowed with tools are promised to produce software systems able to adapt themselves to environmental changes by adding new and/or modifying existing functionalities. Among these topical evolution-centered mechanisms for getting software adaptability ready for working we cite mainly *computational reflection* [74, 17].

Computational reflection is a technique that allows a system to maintain information about itself (meta-information) and to use these data for changing (adapting) its behavior. This is done through the casual connection between the base- (i.e., the controlled system) and the meta-level (the evolutionary system). *Reflection* is the ability of a system to watch its own computation and possibly change the way it is performed. Observation and modification imply an *underlay* that will be observed and modified. Since the system reasons about itself, the "underlay" is itself, i.e. the system has a *self-representation* [74]. A *reflective architecture* logically models a system in two layers, called *base-level* and *meta-level*. In the sequel, for simplicity, we refer to the "part of the system working in the base-level or in the meta-level" respectively as base-level and meta-level. The base-level realizes the *functional* aspect of the system, whereas the meta-level realizes the *nonfunctional* aspect of the system. Functional and nonfunctional aspects discriminate among features, respectively, *essential or not* for committing with the given system requirements. Security, fault tolerance, and evolution are examples of nonfunctional requirements[1]. The meta-level is *causally connected* to the base-level, i.e., the meta-level has some data structures, generally called *reification*, representing every characteristic (structure, behavior, interaction, and so on) of the base-level. The base-level is continuously kept consistent with its reification, i.e., each action performed in the base-level is *reified* by the reification and vice versa each change performed by the meta-level on the base-level reification is *reflected* on the base-level.

The systems running in the base-level are the non-stopping systems prone to be adapted, whereas the nonfunctional feature realized by the meta-level is the software evolution. Evolution takes place exploiting design information concerning to these non-stopping systems. To correctly evolve the base-level system, the meta-level system must face many problems. The most important are:

- to determine which events cause the need for evolving the base-level system;
- how to react on events and the related evolutionary actions;
- how to validate the consistency and the stability of the evolved system and eventually how to undo the evolution;

1 The borderline between what is a functional feature and what is a nonfunctional feature is quite confused because it is tightly coupled to the problem requirements. For example, in a traffic control system the security aspect can be considered nonfunctional whereas security is a functional aspect of an auditing system.

- to determine which information allows system evolution and how such information is involved in the evolution.

In this dissertation, we will tackle the above problems, by providing the RAMSES middleware for dynamically evolving the software systems by using its design information.

Design information have all the necessary data to evolve the modeled system. Therefore we have to build an infrastructure that exploits design information. Such an infrastructure has:

- the ability to manipulate the base-level design information for the meta-level;
- the meta-level takes care of the evolution through:
  ① meta-components observe the runtime events;
  ② for each runtime event, special types of meta-entities and engines built the evolution and validation plan (a set of strategic processes);
  ③ apply the evolution strategic into the meta-level design information;
  ④ apply the validation strategic into the meta-level design information.
- to reflect the consistently evolved meta-level design information to its base level.

The cooperative meta-components at meta-level consults the evolutionary and validation engines, adapting the meta-level design information for new requirements or new runtime behavior. Changes to the meta-level design information can be made at run-time and are reflected to its base-components after the validation. The evolution and consistency are not hard-coded. Instead, we build a reflective framework of the base-systems that can automatically self-adapt for any changes to be active long-life span.

## 1.2. Approach and Contribution of the Dissertation

### 1.2.1. Approach

The approach we will take to prove our dissertation concerns the dynamic evolution of the software systems against runtime changes by using the system design information. For that purpose we will use an approach based on reflective techniques. As design conceptual model, we respect the object-oriented paradigm and its UML method, as one of the mostly accepted and widely adopted methodology in developing different software systems, and information systems in particular. We refer to the output of the design phase—that is several UML diagrams—as the *design information*. The thesis discusses in depth how the dynamic evolution can be driven by the design information, and describes how it facilitates planning and validating the evolution. The main result of these investigations is a reflective middleware, we refer to as RAMSES, whose aim consists of consistently evolving software systems against runtime changes. This middleware

provides the ability to change both structure and behavior for the base-level system at run-time by using its design information. The meta-level is composed of cooperating meta-objects. Whereas, the base objects are controlled by the meta-objects to drive their evolution.

### 1.2.2. Contribution

The main contributions of this dissertation can be divided into the following scientific/-fundamental contributions:

- ❶ we present a lightweight formalisation of the design information. This formalisation serves as a basis for the of evolution and validation plan strategies. In a more context, it provides the explicit definition of each design information model and the connection between different models.
- ❷ we provide the evolution and validation planning aspects, as a set of evolution strategy and validation strategy of the software design information.
- ❸ we define the structure of the intermediate layer (virtual layer) for manipulating the design information.
- ❹ we provide an explicit manipulation method that uses the design information to evolve the systems according the changes in its environment. The UML models represent the data of the systems at the development phase. When we are going to the running phase, we need a model that represent the data included in the UML, we are using XMI schemas that represent the UML models at running phase. Our evolutionary meta-objects reify these data to their meta-level.
- ❺ we introduce the idea of use the rule-base approach for evolving and validating the representative system. We show how our evolutionary and validation rules actions built and applied into the representative of the design information.
- ❻ the main contribution of the research presented in this thesis is the development of a reflective middleware which support the dynamic evolution of the object-oriented software systems. This middleware allows the developer to make changes to the software system at its representative in form of design information. The middleware provides the developer with a runtime view of the base level system in order to allow its evolution.
- ❼ Develop consistent platform-based policies and rules for evolution. These concerns present conception and implementation of evolution and consistency checking rules.

Additionally, we also provide a number of practical contributions:

- ① while the previous contributions are scientific contribution, as a more practical contribution we developed a prototype tool capturing the planning phase of the RAMSES middleware.

② we show the applicability of our approach in the working case study.

③ we implement the evolutionary engines rules and the consistency engines rules by using scripts.

## 1.3. The RAMSES Middleware: General Insight

To provide the reader with a first insight of the RAMSES middleware, we present the main components of RAMSES and their functionalities. The RAMSES middleware can be regarded as two-layers (*RAMSES base-level* and *RAMSES meta-level*) and additional virtual intermediate layer (RAMSES reifications). In the following, the key features of each layer are sketched.

**RAMSES Base-level**: The RAMSES base-level is composed of the base level system. In order to allow describing and reasoning about this level, we explicit structural and behavior view of the base level system in form of design information models. The design information provides different views of the running application. This views help the developer to evolve this systems for currently changes in their domain. The *design information* is the central concept for documenting a software system and it plays also a relevant role in the system maintenance. The UML is the considered formalism for representing the design information.

**RAMSES Meta-level**: This level is the heart of RAMSES middleware. Through this level the evolutionary and validation strategies are created for each runtime events by using special meta-components. There are two meta-objects (*evolutionary meta-object* and *consistency checker meta-object*) are driving the evolution. At the meta-level the running information system—in terms of its interactions, internal computations and external interfaces with current context—is reified to a meta-representation, where adequate rules for controlling its consistent run-time evolution are conceived and implemented. For this difficult task of run-time reification and reflection, we will identify special engines for that purpose.

**RAMSES Reifications**: The RAMSES reification is a virtual layer that manipulates the design information. The RAMSES reification is the description for realized by reification library. It provides the system with the ability of manipulating its design information according to its evolution. It directly performs the manipulation on the XMI representation of the UML diagrams providing an API based on the logic concepts (diagrams, classes, relationships, and so on) and independent of the XMI syntax and complexity. The reification library has two benefits:

- it provides an abstract view of the design information that can be manipulated at run-time,
- it interfaces the data (design information) with the evolutionary application (the evolutionary meta-objects) keeping the data updated.

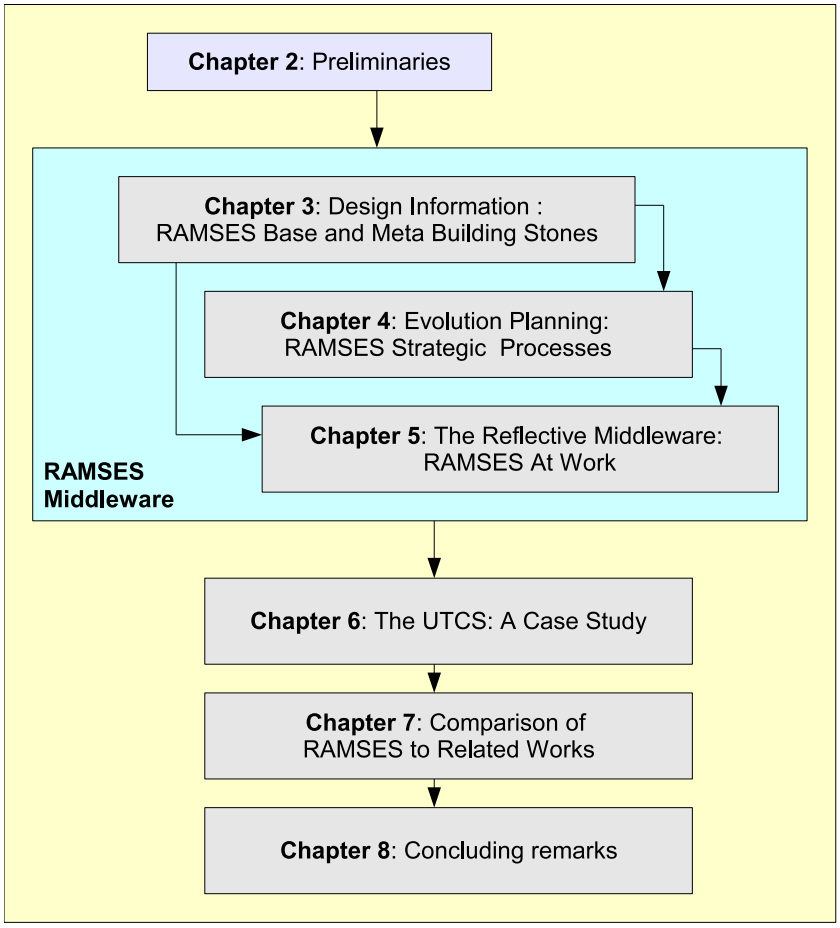



**Figure 1.1:** The thesis outline.

## 1.4. Outline of the Dissertation

The thesis is organized as follows:

Chapter 2, introduces the notions of software engineering methodologies for evolving the software systems. Then, we provide an overview of software maintenance&evolution, reflection, and the current techniques for software evolution. Finally, we introduce an overview of the object oriented analysis method, and the engineering UML 2.0 specification and it's tools.

The core of the dissertation is described by RAMSES middleware as shown in figure 1.1. The core is divided into three chapters: first, chapter 3 describes the UML-compliant base and meta stone building for RAMSES middleware; Second, chapter 4 describes the essence processes for describing the evolution and validation plan; finally, chapter 5 presents the reflective architecture (RAMSES) that provides an application with the ability to adapt according to evolution and validation strategies. The meta-level incorporates some knowledge about the application design to decide how to react to environmental events, and whether it is safe to perform changes into the running application.

Chapter 3, presents the role of design information as the essence for support software evolution. Then, the requirements of the motivation example introduced. This motivation example will be used thought the whole core of the dissertation. After that, we introduce our analysis of the explicit and implicit view for evolving software. Finally, the formalisation is introduced, that enables the precise definition of design information.

The detailed description of how to automate the evolution is introduced in Chapter 4. This chapter presents the way to plan the evolution of the design information, and interprets this kind of evolution by using scripts.

A reflective middleware called RAMSES is laid out in detail in Chapter 5, describing how the liaison between the base and meta level can be built and how it can be used to self adapting a running system to environment changes.

Chapter 6, devotes to assess and enhance the practicability of the RAMSES middleware. In this sense, we deal with a non trivial case study (urban traffic control systems). In order to adequately deal with the evolutionary and consistency checker engines, we present how to built the script rules for evolving and validating.

Chapter 7, compares the RAMSES middleware with respect to most of the existing similar approaches proposed in the literature.

Finally, Chapter 8, provides some conclusion, summaries the contribution drawn from this research, and provides a list of interesting open issues that require further research.

# 2 Preliminaries

This chapter provides some necessary background for the main topic of this dissertation, that deals with (information systems) software evolution using reflection techniques. In this sense, software evolution throughout the software development process, reflection, and conceptual problems related to evolution belong to the landscape of this thesis topic. The main part of this chapter is therefore devoted to bring more light on these fundamental ingredients to adapt software systems, which will be dealt in detail in subsequent chapters.

More precisely, firstly, we give an overview of the whole life cycle in developing information systems. Secondly, we recall the essential concepts of software maintenance and evolution. Thirdly, reflection concept is introduced and it's types are shown. Fourthly, the current techniques are presented. Finally, we close this chapter by introducing the object-oriented methodologies and giving a general overview of UML methodology.

## 2.1. Software Engineering Models

Software engineering models define how to build a correct software system in a stepwise way, and that are as much sensitive as possible for any change in their requirements. The main schools of thought in software engineering are:

- Linear thinking fostered by waterfall life cycle [97]. The waterfall life cycle is divided into sequential phases analysis, design, implementation and testing phase. In each phase we use special techniques and tools. When a phase ends, the next phase starts in a sequential flow, until code production. The problems of this model is that it works only for simple requirements and when the life span for the system software remains long without changes. When the requirement specifications are complex, to make them tractable we have to adopt instead the prototype life cycle [34].
- Iterative or evolutionary thinking, fostered by the spiral model [9]. This model integrates the waterfall with prototyping for producing a model dealing with the complex requirements. It divides the software engineering space into four quadrants: management planning, formal risk analysis, engineering, and customer assessment.

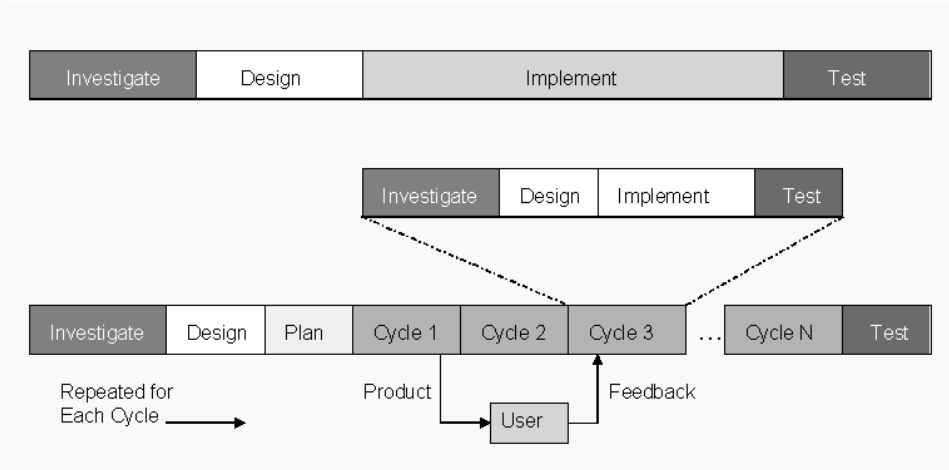



**Figure 2.1:** Classic&Evolutionary software engineering life cycles [32].

The spiral model for software engineering is currently the most realistic approach to the development for large-scale systems and software [32, 47]. There are two archetypes of the spiral model: the first one is, the *incremental life cycle*. This model separates the development cycle into n-cycles, each cycle consists of waterfall phases, with the end of each cycle we have a prototype until we reach the final version of the product. Second archetype is the *evolutionary development*. This model improves the incremental flow by adding the user feedback at the output of each cycle and document his view and new additional functions into the next development cycle as shown in figure 2.1.

The *Win-Win* spiral model [8] as a new version of the traditional spiral model, the Win-Win version strives to involve all stakeholders in the development process. It involves a collaborative engine that establishes "win" conditions set by users, customers, developers, and system engineers in order to evolve requirements throughout the process. The proposed model adds three activities to the front end of each spiral cycle:

- Identify the system or subsystem's key stakeholders.
- Identify the stakeholders win conditions for the system or subsystem.
- Negotiate win-win reconciliations of the stakeholders win conditions.

To be able to describe where evolution is addressed in the software development process, the software development life-cycle needs to be reviewed. Usually, this life-cycle is subdivided into different phases. During the *requirements* phase, the requirements for a software system are discovered, specified and analyzed. In this sense, this phase includes the analysis phase as a sub-phase. In the *design* phase, the software system is designed, but still independent of a specific programming language. Several sub-phases can be distinguished, such as architectural (or high-level) design, mechanistic design and detailed (or low-level) design. During the *implementation* phase, the actual code is written, based on the information given in the detailed design. Usually, template code can be generated directly from the design. The *testing* and validation phases check if

the software system fulfils the specified requirements, and see if the software behaves correctly in all situations. Finally, the *maintenance* phase deals with the software system after it has been delivered, by making bug fixes, implementing new requirements, etc. With respect to software evolution, this is a very important phase, since it is the stage where software evolution occurs continuously.

## 2.2. Software Maintenance and Evolution

Evolution is a critical issue in the life cycle of all software systems particularly those serving highly volatile business domains such as banking, e-commerce and telecommunications. The key difference between *development*, *evolution* and *maintenance* is that development is performed from scratch, whereas maintenance is performed by modifying existing software systems, finally, *evolution* is a subset of maintenance related to the activity and phenomenon of (*adaptive*, *perfective* and *preventive*) software systems [67]. In the next subsection, we describe in details the software maintenance and evolution.

### 2.2.1. Software maintenance

Software maintenance is the general process of changing a system after it has been delivered. The changes may be simple changes to correct coding errors, more extensive changes to correct design errors, significant enhancements to correct specification errors, or accommodate new requirements. Software maintenance does not normally involve major architectural changes to the system. Changes are implemented by modifying existing system components and, where necessary, by adding new functionalities to the system.

The maintenance phase consists of the process of modifying a software system or component after delivery, to correct faults, improve performance or other attributes, or adapt to a changed environment. Usually, four types of maintenance are distinguished in [70]:

**Corrective maintenance** : it concerns the repair of faults found.

**Adaptive maintenance** : deals with adapting the software to changes in the environment, such as new hardware or next release of an existing system. Adaptive maintenance does not lead to changes in the system's functionality.

**Perfective maintenance** : it allows accommodating to new or changed user requirements. It concerns functional enhancements to the system.

**Preventive maintenance** : concerns activities aimed at increasing the system's maintainability, such as updating documentation or adding comments.

All these types of maintenance are concerned with activities aimed at keeping the system usable and valuable for the target organization as long as possible. So, software

maintenance has more service-like aspects than software development, because the value of software maintenance is in the activities that result in benefits for the customers, such as correcting faults and adding new features.

The continuous process of maintenance or evolution characterizes the main features of software, namely its modifiability, and its capacity for change. Moreover, the maintenance phase can be considered as the most important phase of the software development process, since studies have shown that the costs of system maintenance (i.e., evolution) are as high as 60% of the overall development costs [45].

### 2.2.2. Software evolution

Software systems are continuously changing and adapting to meet the needs of their users and surrounding environment in terms of new functionalities, new architecture, new technological support, etc. Therefore, a good understanding of the evolution process is essential. This permits building better concepts, methods and tools to assist developers as they maintain and enhance these systems. Furthermore, it will pave the way for the investigation of techniques and approaches to monitor, plan and predict a successful evolutionary paths for long lived software projects. The dynamic behavior of the software systems as they are maintained and enhanced over their life times and preceded states [4]. That is, software evolution is defined as examining the dynamic behavior of the developed system, how it changes over time.

In research literature, an important distinction is made between two kinds of software evolution: Run-time evolution (autonomous or programmed evolution) and design-time evolution (heteronomous evolution) as follows:

**Design-time** : in this type, changes are made manually by a software engineer during the software development process. The changes to a software artifact can thus be totally unpredictable. As a result, it is very hard if not impossible to create fully automatic process with inherent tools that perform these changes, and ensure that the resulting software artifact is consistent and conflict-free.

**Run-time** : in this evolution type, the implemented software is dynamically modified while it is running. With run-time evolution, software artifacts can change themselves automatically when receiving triggers which activate evolution. An intrinsic aspect of autonomous evolution is that it can only be used to deal with anticipated changes.

With respect to the work in [67], the term software evolution relates to the activity and phenomenon of software change. It includes two aspects that reflect, respectively, the complementary concerns of the how and the what/why [69, 68] of software evolution. Interest in the former is concerned with methods, tools and techniques to change functional, performance and other characteristics of the software in a controlled, reliable, fast and cost effective manner. This is the more widespread view and is exemplified

by the contributions to a series of meetings on *principles of software evolution* [53, 52]. Interest in the what/why, on the other hand, focuses on understanding the software evolution phenomenon, its underlying causes and drivers, common patterns of evolutionary behaviour, and the characteristics of that behaviour. This line of investigation, the focus of the FEAST (Feedback, Evolution And Software Technology) studies in the Department of Computing at Imperial College. FEAST and their antecedents, has also been pursued by a small number of other groups world-wide (e.g. [58, 72, 87]).

Both views, the how and the what/why, must be pursued if mastery of the software evolution phenomenon is to be achieved in a world increasing dependent on computers and software. The following are examples of the type of questions whose answer is pursued under the latter view:

- why does software evolution occur?
- why is it inevitable?
- what are key attributes of the evolution process?
- what is their impact on the software process and its products?
- what are the practical implications of the above on the planning control and management of software system evolution?

With respect this approach, the authors proposed a set of methodological guidelines also named *laws* (Lehman's Laws) concerning system change. They claim these *laws* are invariant and widely applicable. Lehman and Belady examined the growth and evolution of a number of large software systems. The proposed laws were derived from these measurements. The Lehman's Laws (hypotheses, really) are illustrated in the following list [67].

**Continuing change**: a program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.

**Increasing complexity**: as an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.

**Large program evolution**: program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors are approximately invariant for each system release.

**Organisational stability**: over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.

**Conservation of familiarity**: over the lifetime of a system, the incremental change in each release is approximately constant.

**Continuing growth**: the functionality offered by systems has to continually increase to maintain user satisfaction.

**Declining quality:** the quality of systems will appear to be declining unless they are adapted to changes in their operational environment.

**Feedback system:** evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

This approach is rather code-driven evolution one and does not tackle design phases as we are aiming to.

## 2.3. Reflection Terminology in Software Systems

### 2.3.1. Computational reflection

Computational reflection or simply *reflection* is a solution to the problem of creating applications able to maintain, use, and change representations of their own design. Reflective systems are able to use self-representation to extend, modify, and analyze their own computation.

Reflection is born in the field of *Artificial Intelligence* before propagating to various fields in computer science such as logic programming, functional programming and object-oriented programming [36]. It was introduced in object-oriented programming thanks to the famous works of Pattie Maes [73, 74]. Reflection is the ability of a system to observe and manipulate its computation and possibly change the way it is performed. Observation and modification imply an *underlay* that will be observed and modified. Since the system reasons about itself, the *underlay* is itself, i.e. the system has a self-representation [74]. There are two aspects of such manipulation: *introspection* and *intercession*:

- Introspection is the ability for a program to observe and therefore reason about its own state.
- Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning.

Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called *reification* [7].

Computational reflection has been used in several fields, for example for developing operating systems [111, 48], fault tolerant systems [59], compilers [63], and also for building distributed frameworks [60, 66, 31, 71, 1].

### 2.3.2. Reflection in the object-oriented paradigm

An object-oriented reflective system is logically structured in two or more levels, constituting a reflective tower as shown in figure 2.2. The first level is the base-level and describes the computations that the system is supposed to do. The second one is the meta-level and describes how to perform the previous computations. The entities (objects) working in the base level are called base-entities, while the entities working in the other levels (meta-levels) are called meta-entities. Each level is causally connected to adjacent levels, i.e. entities working into a level have data structures reifying the activities and the structures of the entities working into the underlying level and their actions are reflected into such data structures. Any change to such data structures modifies entity behavior. Each level, except the first and the last one, is a base-level for the above level and is a meta-level for the underlying level.

Meta-entities supervise the base-entities activity. The concept of trap could be used to explain how supervision takes place. Each base-entity action is trapped by a meta-entity, which performs a meta-computation, then it allows such base-entity to perform the action. The infinite regression of the reflective tower can be managed in different ways. Brian Smith suggested the use of lazy evaluation in 3-Lisp [99]: an interpreter is not created unless needed.

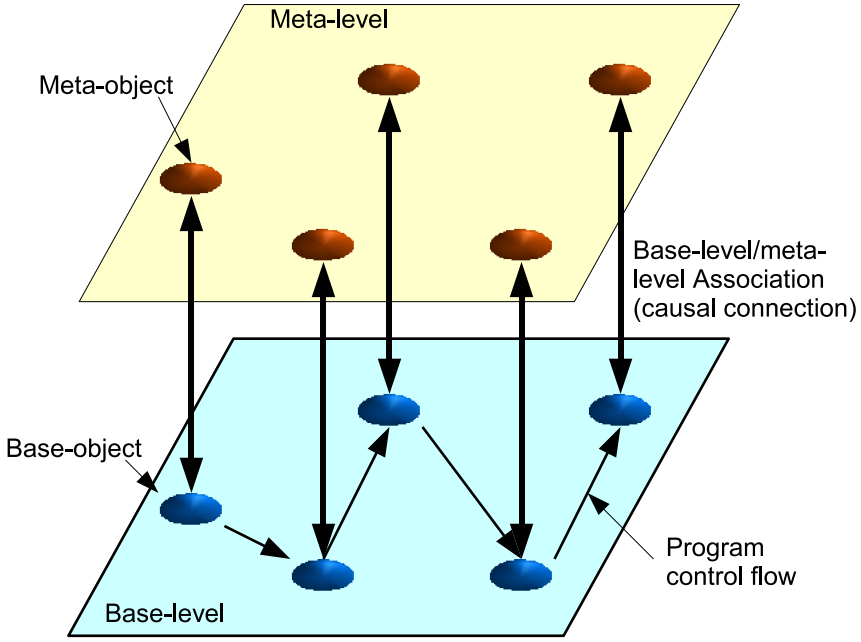



**Figure 2.2:** Reflective tower.

It is possible to observe, going beyond the reflective tower of compilers/interpreters, that each reflective computation can be separated into two logical aspects: computational flow context switching and meta-behavior. A computation starts with the computational flow in the base level; when the base-entity begins an action, such action is trapped by the

meta-entity and the computational flow raises at meta-level (shift-up action). Then the meta-entity completes its meta-computation, and when it allows the base-entity to perform the action, the computational flow goes back to the base level (shift-down action) [18].

### 2.3.3. Reflection models

Meta-levels can be used to explain and self-describe the structural and computational models of a language in term of its own data and control structures. Two reflection models in the object-oriented paradigm are identified in [39]: Structural [29, 15] and Behavioral Reflection [74].

#### Structural reflection

In the structural model, the meta-level is constituted by meta-classes. A meta-class is the class of a class considered as an object. Meta-classes have information on structural aspects of objects at the base level: if this information is modified, then the structure of these objects is modified accordingly. This model allows designers to extend the static part of an object-oriented language.

Structural reflection has been included in an extension of the Java programming language [76] known as metaXa. This is in addition to the reflective capabilities already present in standard Java, and allows more than one meta-object per object.

#### Behavioral reflection

In the behavioral model, objects at the meta-level are called meta-objects [73]. A meta-object is similar to a normal object, but it maintains all the reflective information. The class of a meta-object is named meta-object class. The activation of a meta-object depends on the access to the associated base-level object, named reflected object. Any invocation to a reflected object service produces the execution of a specific method of the associated meta-object. A meta-object has information about the behavioral aspects of base-level objects, for example, how a specific object treats a message. The meta-object model is different from the meta-class model, mainly, because of the association is established between objects and not between classes.

Behavioral reflection can also be added to Java [109], allowing the programmer to alter the behavior of the virtual machine at run-time.

## 2.4. Current Techniques of Analysis and Evolution of the Software Systems

There are a number of high-level techniques for software analysis and evolution, however none have proved satisfactory as a general-purpose evolution strategy. We discuss some methodologies for software analysis and evolution in the following subsections.

### 2.4.1. Re-engineering (renovation)

Re-engineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Re-engineering is the most dominant maintenance approach, however the main goal is the recovery of design information, not its exploitation [28, 101]. Re-engineering aims to solve the growing problem of maintaining legacy systems. It consists of three parts: reverse engineering, restructuring and forward engineering. The reverse engineering stage aims to create representations of the system at higher levels of abstraction, to retrieve lost design information. The optional restructuring stage applies general-purpose transformations to some intermediate representation of the source code, and then the forward engineering maps the intermediate representation back to code, possibly of a different language. For example, during the re-engineering of information management systems, an organization generally reassesses how the system implements high-level business rules and makes modifications to changes in the business for the future.

Design recovery is a subset of re-engineering, and it is distinguished by the sources and span of information should be handle. According to Ted Biggerstaff: "Design recovery recreates design abstract from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains... Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software engineering representation or code" [6]. A key objective of design recovery is to develop structures that will help the software engineer understand a program or system.

### 2.4.2. Impact analysis techniques

Software change is a fundamental ingredient of software maintenance. Impact analysis is key in analyzing the changes or potential change and in identifying the software objects. Impact analysis is the activity of identifying what to modify to accomplish a change, or of identifying the potential consequences of a change [3]. Example of impact analysis are:

- using cross reference listings to see what other parts of a program contain references to a given variable or procedure,
- browsing a program by opening and closing related files,
- using traceability relationships to identify changing artifacts,
- using configuration management systems to track and find changes, and
- consulting designs and specifications to determine the scope of a change.

Research in change impact analysis has varied from approaches relying completely on static information, including the early analysis of [2, 62], to approaches that only utilize dynamic information, such as [65]. There also are some approaches that describe the dynamic traceability for unanticipated events [79] and different types of impact analysis of UML models [14, 37, 13].

An early form of change impact analysis used reachability on a call graph to measure impact. This technique was presented in [2] as intuitively appealing and a starting point for implementing change impact analysis tools. Kung et al. [62] described various sorts of relationships between classes in an object relation diagram, classified types of changes that can occur in an object-oriented program, and presented a technique for determining change impact using the transitive closure of these relationships.

Law and Rothermel [65] present a notation of dynamic impact analysis. In this approach, if a procedure p is changed, any procedure that is called after p, as well as any procedure that is on the call stack after p returns, is included in the set of potentially impacted procedures.

An approach is presented in [79], that fosters changes of software by managing runtime-traceable dependencies of requirement specifications and test cases to corresponding architectural elements and source code fragments. In case of (unexpected) change requests it is easy to find the affected system parts, thus facilitating timely change propagation and regression testing.

*Horizontal Impact Analysis* (HIA) and *Vertical Impact Analysis* (VIA) are presented in the context of UML-based iterative development. HIA focuses on changes and impacts at one level of abstraction, and corresponds to what people have generally been doing (e.g.[14, 2]). Whereas VIA focuses on changes at one level of abstraction and their impacts at another level of abstraction. Moreover, VIA is based on a careful classification of changes and refine in UML models and an automated identification of refinements based on detected changes. For each refinement, traceability links are then automatically established and can then be used to control the impact of changes in more abstract models on more refined models [37, 13].

### 2.4.3. Refactoring

Refactoring can be regarded as restructuring or behavior preserving transformations of the source-code of an object-oriented program without changing its external behavior [43, 84, 85]. The overall goal of refactoring is to improve the maintainability of software. The key idea is to redistribute attributes and methods across the class hierarchy to prepare the software for future extensions. If well applied refactoring improves the design of software, make software easier to understand, helps to find bugs, and helps to program faster. We argue that the concept of refactoring would be very worthwhile for runtime evolution. Whenever a change operation can be split into a refactoring and a functional change, it will be easier to handle, if only we have a way of validating that the semantics of the refactored system is kept.

The practical analysis for refactoring is presented in [94], this technique was to make refactoring more practical for object-oriented programming, hence the development of the *refactoring browser*. The main contributions this work brought were through outlining a specific definition of some of the more common refactorings by identifying pre and post conditions that are required to be met before/after applying any method. An empirical study was carry out on the usefulness of applying refactorings to a simple system. The study proved that using refactorings help to reduce the cost development and delivery time of a system [105, 104].

### 2.4.4. Slicing object-oriented approaches

Program slicing has a range of applications such as code understanding, debugging, program testing, reverse engineering, and metrics analysis [54]. Weiser [108] defines a slice with respect to a slicing criterion that consists of a program point (P) and a subset of program variables (V). Slices are executable programs that are constructed by removing zero or more statements from the original programs. Weiser's algorithm uses dataflow analysis on control flow graphs to compute inter-procedural slices. Ottenstein and Ottenstein [86] define a slicing criterion to consist of a program point (P) and a variable (V) that is defined or used at (P). They use a graph reachability algorithm on a program dependence graph to compute a slice that consists of the program that may affect the value of (V) at (P). The two-pass graph reachability algorithm described by Horwitz, Reps & Binkley [51] makes use of procedure dependence graphs to compute slices on procedures. A class dependence graph (CLDG) [96] is a graphical representation of a class. The construction of a class dependence graph makes use of procedure dependence graphs to represent the methods (class members) of the class.

Korel and Laski have introduced the notation of dynamic slicing [61]. A slice computed for a particular fixed input. The availability of runtime information makes dynamic slices smaller than static slices, but limits its applicability to that particular input. Larsen and Harrold [64] present how to compute slices for individual classes, groups of interacting classes and complete programs. the presented class dependence graphs are

efficiently constructed for derived classes and interacting classes by incorporating parts of previously constructed class dependence graphs.

Jackson and Rollins [54] have introduced chopping which reveals the statements involved in a transitive dependence from one specific statement (the source criterion) to another (the target criterion). A chop for a chopping criterion (s, t) is the set of nodes that are part of an influence of the (source) node s onto the (target) node (t). This is basically the set of nodes which are lying on a path from (s) to (t) in the procedure dependence graph (PDG).

## 2.5. Object-Oriented Analysis and Design Techniques

We give an overview for several semi-formal or informal object-oriented generations. There are three generations for object-oriented methodologies: The first generation methods are called object oriented analysis and design (OOA&D). These methods are *Booch* [10], *Objectory* [55] and *OMT* [98]. The second-generation fostered by *Fusion* [30] for providing the systematic object oriented software engineering methodology. The third-generation specifying method, visualizing, and documenting the application of an object-oriented system under development. The UML [11] is third-generation modeling methodology for analyzing and specifying object oriented systems. Moreover, UML provides models through all the life cycle. The UML combines and extends elements of previous object-oriented notations such as *OMT*, *Booch*, and *Objectory*. In contrast to these methodologies, its notations are precisely defined using the object constraint language (*OCL*) [82] and a meta-model to express the allowed forms of diagrams and their properties. The UML methodology provides systematic models through all software engineering phases.

Software systems have a complex structure, to capture the structure and behavior of these systems you need a set of models. Hence, UML is composed of structural and behavioral models that can be used to model a system at their life cycle [11, 83, 42]. These models are classified into three categories as shown in figure 2.3: Engineering the above classification of UML models by using waterfall life cycle is illustrated in figure 2.4. The proposed UML life cycle is composed of four phases:

**Requirement phase**: during this phase the functionality and non functionality of the software systems are captured in form of *use cases model*. The use cases model are a technique for capturing functional requirements of the system and such are used in the requirements phase of the development cycle. Use cases are also well known, however, nothing in UML describes how the content of a use case should be captured. The primary elements are termed as *actors* and the processes are called *Use Cases*. The Use case diagram shows which actors interact with each use case.

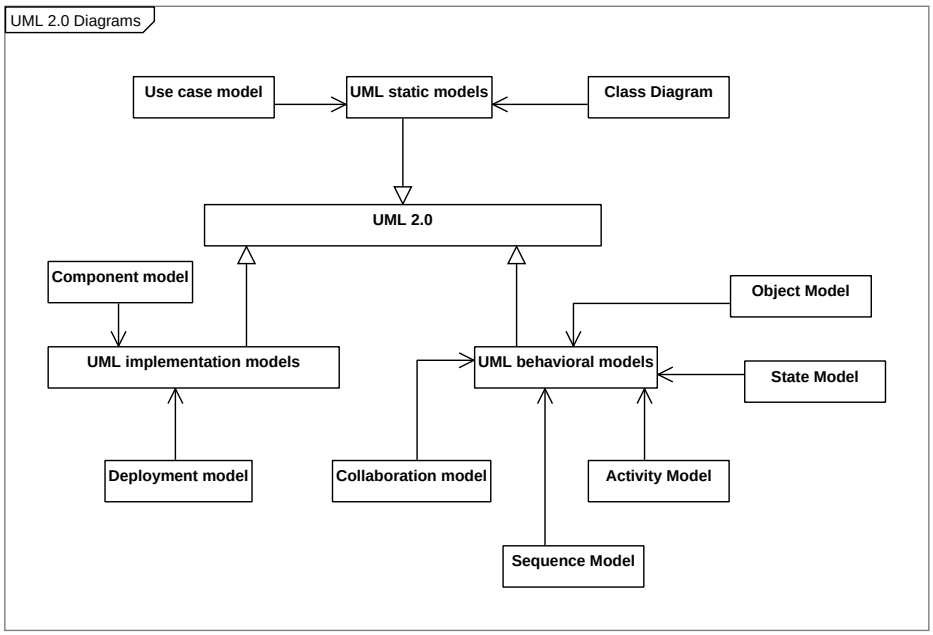



**Figure 2.3:** UML 2.0 diagrams classifications.

**Analysis phase:** based-on the output of the requirement phase. The goal of the analysis phase is to create the *sequence model* and the system structure description in form of *class diagram*.

- The class diagram classifies the actors defined in the use case diagram into a set of interrelated classes. The relationship or association between the classes can be either an *is-a* or *has-a* relationship. Each class in the class diagram may be capable of providing certain functionalities. These functionalities provided by the class are termed *methods* of the class. A part from this, each class may have certain *attributes* that uniquely identify the class.
- A sequence model represents the dynamic behavior of the system by depicting the sequence of actions that occur in a system. The important aspect of a sequence model is that it is time-ordered. This means that the exact sequence of the interactions between the objects is represented step by step. Different objects in the sequence model interact with each other by passing *messages* (method call).

**Design phase:** the design models are created based on the requirement and analysis models. The design models are:

- The object model is a special kind of class diagram. An object is an instance of a class. This essentially means that an object represents the state of a class at a given point of time while the system is running. The object diagram captures the state of different classes in the system and their relationships or associations at a given point of time.

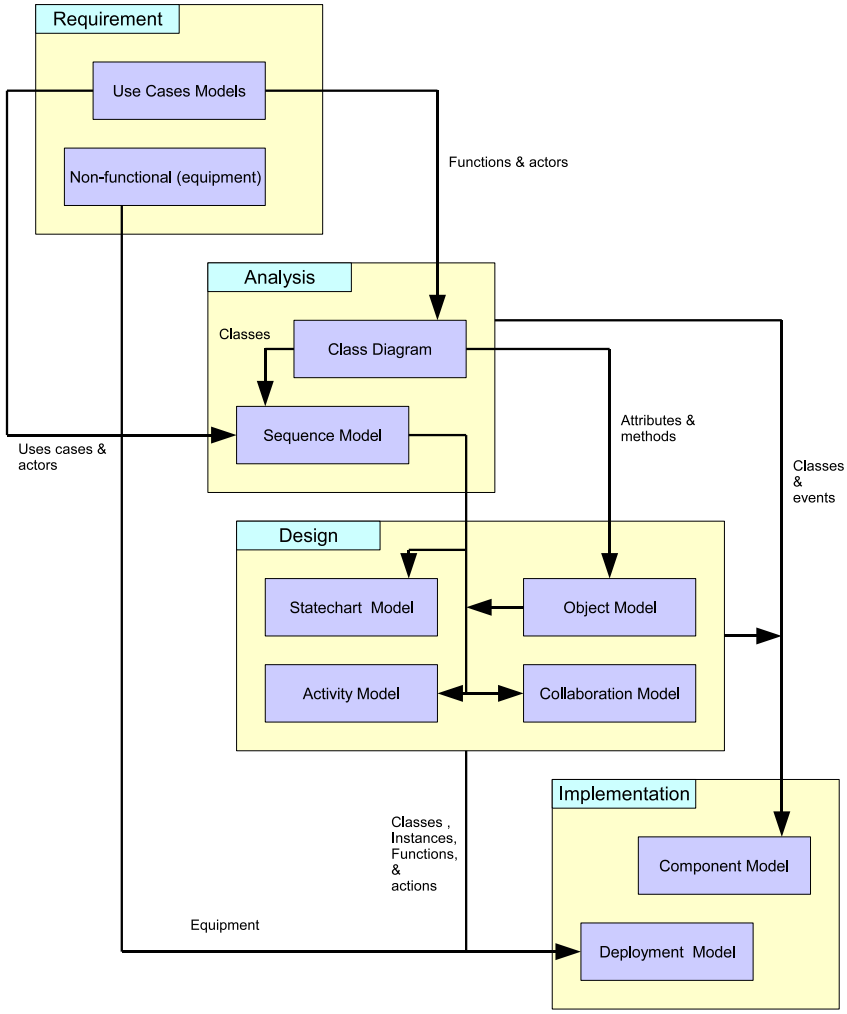



**Figure 2.4:** Engineering UML specification.

- A state model is a quite well-known technique to describe the behavior of the software system. In particular, a state model defines the possible states of a certain object can posses and the different state transitions existing between states. In addition to this, a state model also captures the transition of the object's state from an initial state to a final state in response to events affecting the system.
- An activity model is used to describe work flow or procedure logic. This model gives more lights on the object states transitions and the activities causing the changes in the object states. The process flows in the system are captured in the activity model. Similar to a state diagram, an activity model also consists of activities, actions, transitions, initial and final states, and guard conditions.
- A collaboration model represents the associations between different objects in the system. The associations are listed as numbered interactions that help

to trace the sequence of the interactions. This model helps to identify all the possible interactions that each object has with other objects in the system.

**Implementation phase:** The UML implementation models describe the way for adding new features and for deploying the system as follows:

- The component model represents the high-level parts that make up the system. This diagram depicts, at a high level, what components form part of the system and how they are interrelated.
- The deployment model captures the configuration of the runtime elements of the application. This diagram is by far most useful when a system is built and ready to be deployed.

## 2.6. Summary

This chapter has introduced relevant work from the literature, with particular emphasis on work related to software evolution, software architecture, software reflection, and modeling languages. This material forms the background, motivation, and basic material for the work presented in the rest of this thesis.

# 3 Design Information: RAMSES Base and Meta Building Stones

In this chapter, we describe the role of the design information, that means how the design information support software evolution. To evolve the software systems you need to know more about the classes, the collaboration between them and so on. For that we are going to the abstract level of design information, which gives us the ability to explicit a global view the concrete-level and all the software components. This chapter represents some aspects of design information in terms of presenting the structural and behavioral models of the concrete-level. This will be used in subsequent chapters as the cornerstone for software evolution. In chapter 5, our middleware uses the design information to drive the runtime evolution of the software system.

This chapter is organized as follows: Section 3.1, overviews the design information as the liaison between software design and concrete-level. Next, section 3.2, introduces the requirements of a case study, this case study models a UTCS application and is used as running example throughout this dissertation. Then, section 3.3, describes the implicit and explicit behavior. Moreover, it describes in more details the design information. Finally, section 3.4, presents the design information taxonomy and their formal realization.

## 3.1. Introduction and Motivation

UML [11, 80] is the main methodology for software development, which describes the system's behavior, architecture and components. The design phase provides all the models necessary to the system to plan its evolution and a good evolutionary plan can be directly designed on UML diagrams producing a new set of design models.

Software evolution is the most important process in software engineering, yet there is little consensus on how evolutionary changes should be made, thus there is a little coherency in maintenance practice.

The design information is the term that describes the software systems during the design phases and evolution phases. We consider, the design information is both the graphical representation in form of UML and their internal representation in form of XMI. Both forms have internal-connection between each others, this means any change in one form implies change to the other.

The design information is typically used at design time to describe and establish a common understanding about the abstract realization of the software system. Design information is usually not explicitly represented at runtime. Recently, the introduction of software platforms supporting component plug-in, dynamic binding, PIM (platform independent model) and PSM (platform specific models) have facilitated adaptation of the software systems at runtime. It is important that the models described by the design information are preserved during adaptation. In chapter 5, we propose a reflective middleware for self-adaptation that exploits design information to evolve the system against dynamic changes. In the following we describe the importance of using the design information:

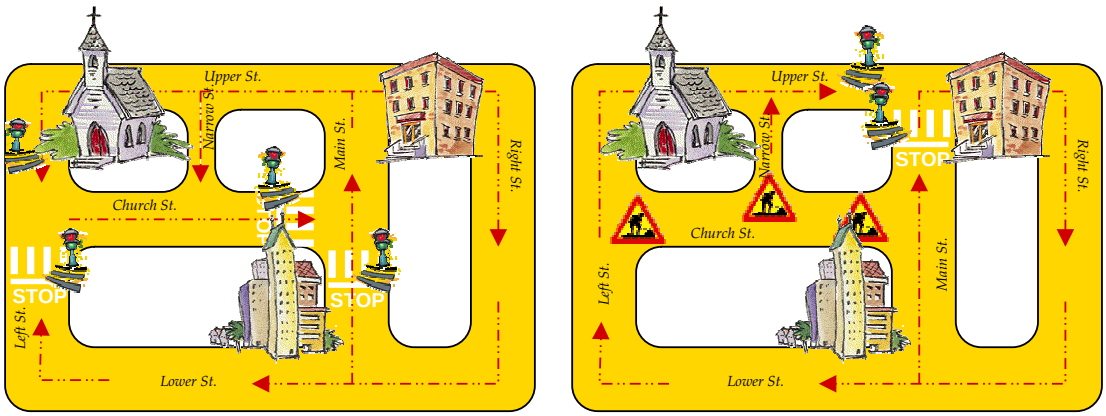
- Design information provides a global view impacting of the whole system.
- From the definition of the design information, it gives the opportunity, through their models to understand the concrete base-level system. From this point of view, we can say it is the representation that is able to support both the horizontal and vertical evolution that describe the new changes. Most of the other approaches drive the evolution to one model of the design information neglecting the strictly connecting models with it.
- The role of design information is realized from a version of a software system to aid in evolving on that particular version. In other words, the design information of the earlier versions can be used as a cornerstone to evolve the later versions.
- Describing the evolution of the code can be done by describing the changes that occur to it. The way to describe the source code changes is by describing the changes to their realized design information.

In the following sections, we describe the realized abstract representation of the software system through the running example (UTCS). Then, we present in details through example the explicit and implicit design information views.

## 3.2. The urban traffic control system (UTCS): simplified view

In this section, the case study is introduced as running example throughout the next chapters. Our case study is a design of a urban traffic control system (UTCS) simulation. We now specify the requirements for the UTCS example.

The case study concerns a simulation part of the urban traffic control system. Here we illustrate the reflective object model for the traffic control system by using UML [11, 100]. The evolution of complex urban agglomerates has posed significant challenges to the city planners in terms of optimizing traffic flows in a normally congested traffic network [95]. Simulation and analysis of such systems require modelling the behavioral, structural




(a) Standard situation (b) Temporary situation

**Figure 3.1:** City layout: a) the layout during normal activities b) the layout during road maintenance.

and physical characteristics of the road systems. This includes at least mobile agents themselves, the roads and intersections.

It is fairly evident that modeling and developing an urban traffic control system is an hard job for software engineers. The most important issues they have to deal with are: slowly evolving road situation, that the model must reflect accurately at all times, changes to the road situation that happens with no warning (accidents, broken traffic lights etc.) and that the system must take into account immediately, and of course the ever changing flow of people and vehicles and the dire consequences of restarting the system during rush hour or at all.

There are many other non-stoppable systems that have problems similar to the urban traffic control system. Air traffic control, assembly line and nuclear station power are some examples of this kind of systems. Their problems are related to the fact they are non-stoppable and need a higher reactivity to sudden environmental changes.

The map in Fig. 3.1.a could represent a simplification of a real city map. Notwithstanding that, it can help us in understanding the problems that a city planner has to face when plans the UTCS of its city. The city planner must plan traffic system taking in consideration several issues, two of them, that we consider in the case study, are:

- cars must be able to reach every road from everywhere; and
- opposite traffic lights at the same crossroad (e.g., traffic lights at the crossway between *Church St.* and *Main St.* in Fig. 3.1.a) must be synchronized or they are useless.

A city map can be easily represented by an oriented graph $G \equiv (\texttt{Crossroads}, \texttt{Roads})$ whose nodes are crossroads and whose edges are roads. Therefore, the first requirement

above can be formalized in:

***Proposition*** 3.2.1. $\forall c_1, c_2 \in$ Crossroads$\wedge c_1 \neq c_2\ \exists p \equiv \{r_1 \dots r_n\}, r_i \in$ Roads s.t.$\forall i, 1 \leq i < n$, $\exists c \in$ Crossroads s.t. $r_i \equiv (x, c) \wedge r_{i+1} \equiv (c, y) \wedge r_1 \equiv (c_1, v) \wedge r_n \equiv (w, c_2)$. That is, all crossroads in the map are connected by a path of roads. ■

Analogously the second requirement can be formalized in:

***Proposition*** 3.2.2. $\forall r, p \in$ Roads s.t. $r \perp p$ if $\exists t_r, t_p \in$ TrafficLights $\implies$

$$Sync(t_r, t_p) = \begin{cases} t_{r_{green}} \implies t_{p_{red}} \vee t_{p_{yellowGreen}} \\ t_{r_{red}} \vee t_{r_{yellowGreen}} \implies t_{p_{green}} \\ t_{p_{red}} \vee t_{p_{redYellow}} \implies t_{r_{green}} \end{cases}$$

***Where.*** `TrafficLights` is the set of all the traffic lights marked on the map, $t_r$ and $t_p$ are respectively the traffic lights in $r$ and in $p$. ■

An urban traffic control system that respects such criteria is consistent with the basic requirements that a livable city must have. Therefore, the software engineer that designs an UTCS should guarantee that such criteria will be respected. *These requirements define the* ***consistency*** *of the system design.*

## 3.3. Explicit versus Implicit View

Object-oriented software systems are composed of components (such as: *classes*, *packages*, *libraries*,..etc). These components have inter- or intra- connection between each other. The implementation code of these systems has implicit information, that describes the relations such as: *inheritance*, *delegation*, *shared attribute*, *method calls*, and *encapsulation*. To evolve these systems, we need to tangle the distributed components of the implementation code to constitute a global view of the whole system. It is a sort of snapshot that abstracts how these concrete components work and interact to each other. The global view should perfectly provide an explicit view for all the concrete and implicit information hidden in the code. One of the most critical problems in software evolution is to consistently propagate the evolution from a piece of code to all the *implicit* related pieces of code. The problem is just related to the implicit relation established among the code process.

### 3.3.1. Implicit behavior

Most of the techniques mentioned in section 2.4 are able to analysis and evolve some pieces of the applications. Usually, the applications are composed of many components. Evolving some parts of these application neglecting how these components are connected

to each other leads to inconsistency problems[1]. To correctly evolve these applications, we should have a complete overview of the concrete level of these applications. In the following items, we discuss the common failure of existing approaches:

**re-engineering**: is a general solution of design recovery and does not provides a specific solution how the recovered data should be structured in appropriate form, how they connect. The problem is part of the *forward engineering* phase, which is delegated to any sound development methodology. However for contemporary systems, especially those built with an incremental methodology, the problem of evolution is not how to migrate code to a new language, but rather how to quickly, reliably and easily implement desired changes to the running application. Re-engineering is not an appropriate solution for this problem.

**impact analysis techniques**: is promising techniques for dynamic and static traceability of the application and events. The technique is quite limited for evolution in the general sense. It requires a priori knowledge of the kinds of changes to expect for the impact class instances. Moreover, these techniques cannot support evolutionary change to program elements that do not participate in the defined impact class.

**slicing techniques**: slicing and chopping have many applications, but for software maintenance they are most useful for performing impact analysis. One of the most difficult tasks in software evolution is to identify potential consequences of a proposed change. This ripple effect analysis is strongly supported by these techniques. Unfortunately, slices only identify elements that may need to be changed - they do not compute what changes need to be made. Thus slicing and chopping should ideally be combined with some form of incremental computation.

**refactoring techniques**: refactoring techniques provide the best way to evolve each of the distributed pieces of the application code. This means, these approaches provide the way to vertically adaptation (we means with *vertical adaptation* is the ability to localize only one component, and adapt it for runtime changes). This lead us to say, these techniques lack of the ability of *horizontal adaptation*, that describes the ability to evolve the software components as well as the intra- and inter components connections.

To complete the description of the behavior for the above techniques, we show the role of these techniques through a simple case of the UTCS. In figure 3.2, we illustrate the snippet of the concrete representation. We have six classes: *road*, *trafficLight*, *syncManager*, *syncProtocol*, *Case1*, and *trafficLink*. The UTCS case1 is composed of ten crossroads, each crossroad connects two one-way road instances. There are two types of crossroads: first, complex crossroad that manages the traffic flow of two road instances by using two of synchronized traffic light instances. Second, simple crossroad connects

1 The inconsistency problems realize when the application does not accept the whole or part of changes.

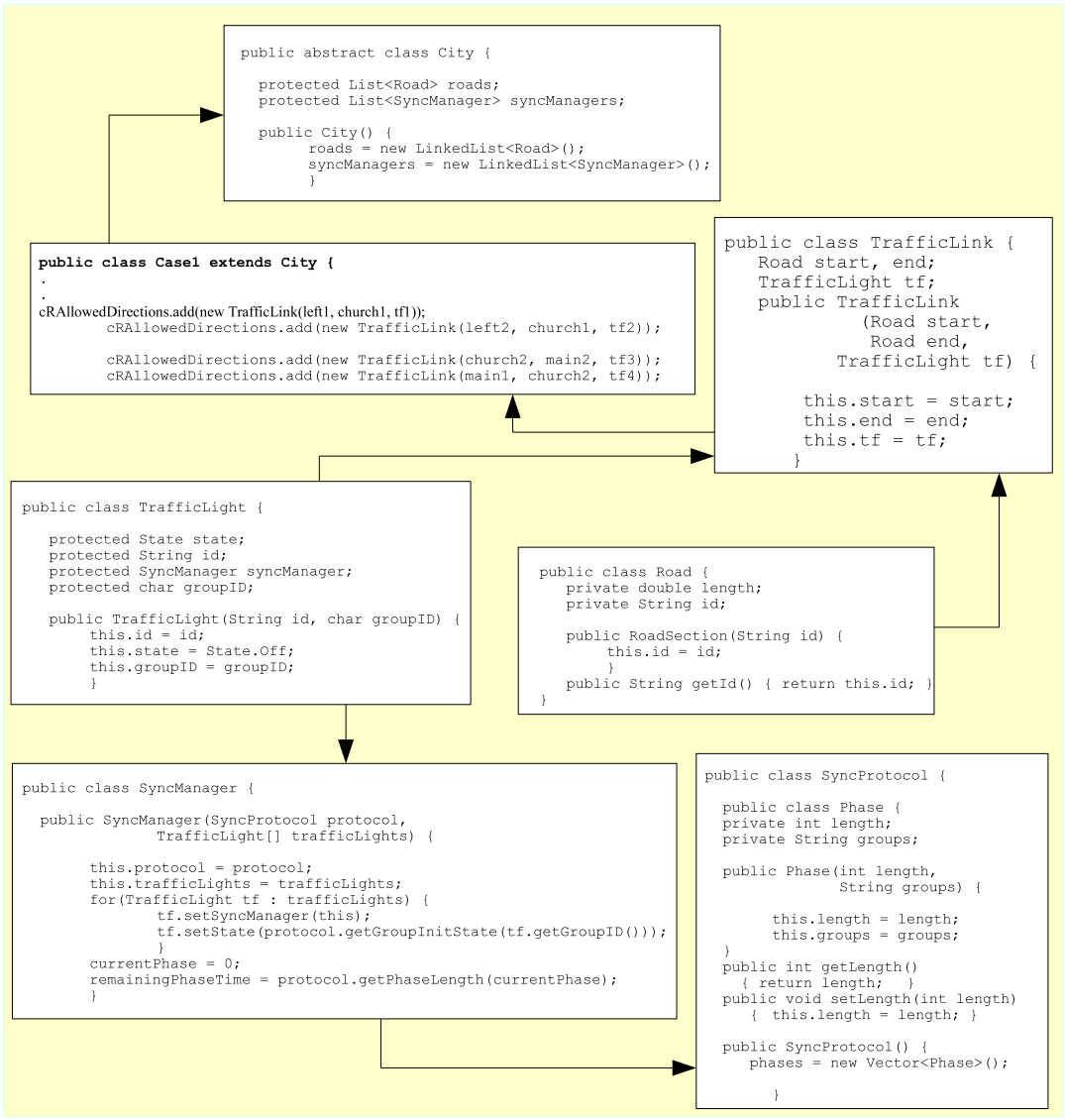



**Figure 3.2:** The representation of concrete-level of UTCS.

the two different road instances without traffic lights. In the following, we describe the complete structure of the UTCS case1 by listing all the simple and complex crossroads as follows:

- crossroad(left1, church1, tf1); crossroad(left2, church1, tf2);
- crossroad(church2, main2, tf3); crossroad(main1, church2, tf4);
- crossroad(main2, upper2); crossroad(main2, upper1);
- crossroad(upper2, right); crossroad(right, lower1);
- crossroad(lower1, main1); crossroad(lower1, lower2);
- crossroad(lower2, left1); crossroad(upper1, narrow);

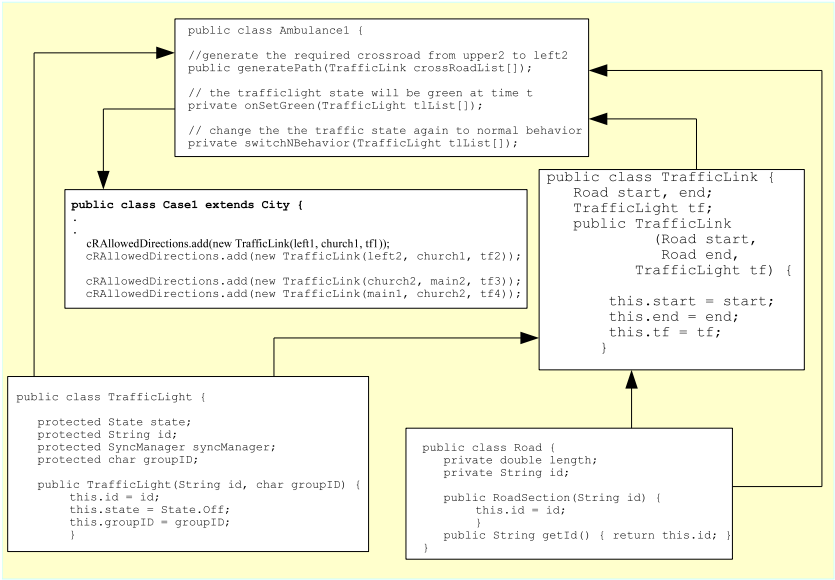
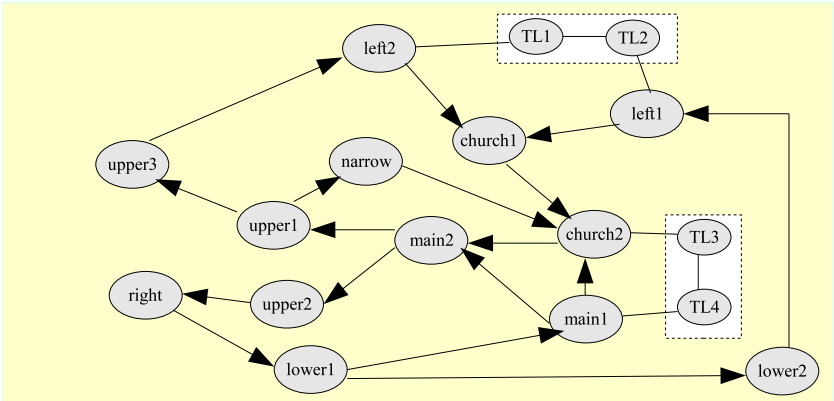



**Figure 3.3:** The refactor view of the concrete-level of UTCS.

- crossroad(narrow, church2); crossroad(upper1, upper3);
- crossroad(upper3, left2).

As in crossroad description, there are four traffic lights (tf1,tf2,tf3,tf4). These traffic lights are composed of two synchronization groups. Each group consists of two traffic lights as follows: *twoGroupSync(tf1,tf2)* and *twoGroupSync(tf3,tf4)*. These two groups of synchronization are managed by two classes at the concrete level: (*syncManager* and *syncProtocol*). Suppose we want to describe a particular behavior state such as (a priority for the ambulance route from road upper2 (accident) to left2 (hospital)). The main goal is to explicit this behavior from the implicit data at the concrete level components. This kind of behavior could be used to describe the shortest path between the hospital and the point where the accident occurred. It is difficult to realize from specific part of the code this kind of particular behavior. For that, we are looking for a method, that is able to detect such kind of explicit data. In figure 3.2, we illustrate the implicit view through the distributed classes of the UTCS case1. In the rest of this subsection, we describe the role of the refactoring and slicing&impact analysis techniques of the UTCS case1 related to the particular behavior *ambulance route*. Moreover, we illustrate how these techniques provide their solutions to describe this particular behavior.

An important kind of change to object-oriented software is a refactoring. Examples of refactoring such as: changing the names of classes and methods, moving methods and fields from one class to another, and splitting methods or classes. The common philosophy of refactoring is that change the structure of a concrete level (program) not its behavior. For that, it is difficult to explicit the required data that describes a specific behavior of the system and change it. The normal behavior of the refactoring techniques is shown in figure 3.3, that proposes a solution for solving the ambulance case




```
public class motivationCase extends City {
.
.
cRAllowedDirections.add(new TrafficLink(left1, church1, tf1));
        cRAllowedDirections.add(new TrafficLink(left2, church1, tf2));

        cRAllowedDirections.add(new TrafficLink(church2, main2, tf3));
        cRAllowedDirections.add(new TrafficLink(main1, church2, tf4));

        cRAllowedDirections.add(new TrafficLink(main2, upper2));
        cRAllowedDirections.add(new TrafficLink(main2, upper1));
        cRAllowedDirections.add(new TrafficLink(upper2, right));
        cRAllowedDirections.add(new TrafficLink(right, lower1));
        cRAllowedDirections.add(new TrafficLink(lower1, main1));
        cRAllowedDirections.add(new TrafficLink(lower1, lower2));
        cRAllowedDirections.add(new TrafficLink(lower2, left1));
        cRAllowedDirections.add(new TrafficLink(upper1, narrow));
        cRAllowedDirections.add(new TrafficLink(narrow, church2));
        cRAllowedDirections.add(new TrafficLink(upper1, upper3));
        cRAllowedDirections.add(new TrafficLink(upper3, left2));
```

**Figure 3.4:** The slicing and impact analysis view of the concrete-level of UTCS.

by introducing a new class named *Ambulance1*, that is used to apply the emergency plan for the crossroad instances of the class *case1*. With this solution, refactoring techniques increase the implicit information at the concrete level. Moreover, it does not exploit the required data that describe new behavior (*ambulance route*). The semantic information about a system is scattered in a large amount of implicit code, with this practical simple refactoring scenario is leading to the structural evolution not behavioral. The example in figure 3.3, shows how implicit information represents particular behavior at the code. In this simple scenario a developer starts a short refactoring session, in which he/she refactors the new class *ambulance1*. He/She (1) extracts a new class named *ambulance1* ; (2) creates new associations between the other classes; and (3) renames the synchronization method to ambulanceSyncCase, replacing all references to *ambulance1* in the base-code . This refactoring scenario requires additional implicit code to switch the system behavior to the normal state.

The main goal of the slicing and impact analysis techniques provides the traceability of the whole application as shown in figure 3.4. Object-oriented slices techniques are able to produce static of the software system. This kind of traceability graph is named *Class Dependency Graph (CDG)*. This graph lacks for understandability by the developer and its difficulty to support new adaptations. In contrast, the impact analysis techniques provide a good traceability for both static and dynamic view of the system, and drive pieces of evolution by providing impact class which contact to all the other components.

But the problem is the evolution proposed by impact analysis still static. Impact analysis techniques introduce a set of atomic changes into the implicit code such as: add new method named *ambulanceSyncCase*, to the class *syncProtocol* for describing the new particular behavior, and another method to the main class *case1* named *switchNBehavior* to switch the system again to the normal behavior.

The fundamental problem underlying the approaches sketched above is that, first, refactoring techniques do not provide an explicit view of the description of the whole behavior. Moreover, it increases the implicit data through the code. Second in case of slicing, these techniques are able to built an explicit view from the concrete level, but it lacks for a particular explicit view and evolution.

In the next subsection, we describe the design information to explicit design models from the concrete level.

### 3.3.2. Explicit behavior

In this subsection, we introduce how to explicit global view from the implicit concrete code of the UTCS case1 by using design information models. With logic thinking of evolution problem, we need to give an explicit data of the whole system, then propagate the evolution based on the consistency propagated evolution of the explicit data.

From the definition of the design information, it provides an explicit data for the implicit concrete-level. The explicit data can be structured in a set of models such as: (1) explicit model that describes the structure of the implicit components and their interaction; (2) explicit model that describes the behavior of these components; and (3) explicit model that describes the particular behavior of the whole system. For that, the design information provides a set of explicit models for the concrete level. These models are connected and the most properties of these explicit models is driving the evolution in both *vertical* and *horizontal*. We give a closer look for the both type of evolution for the design information: first, *vertical evolution* is the ability to evolve and analysis each model or component alone. Most of the other techniques apply this kind of analysis and evolution. Second, *horizontal evolution*, in sense that all the explicit models should be provide the whole structure and behavior of the system. For that reason, The evolution based for elements of one model of the design information should be reflected also to the connected elements at the other explicit models. For example, the changes in the implicit component *traffic light* implies changes to the *explicit structure model*, *explicit behavior model*, and *explicit system behavior model*.

Now, we illustrate the role of the design models to explicit specific models for the UTCS case1, that describe the normal behavior for the UTCS case and the ambulance behavior. The realized explicit models should describe the particular behavior of the ambulance behavior case.

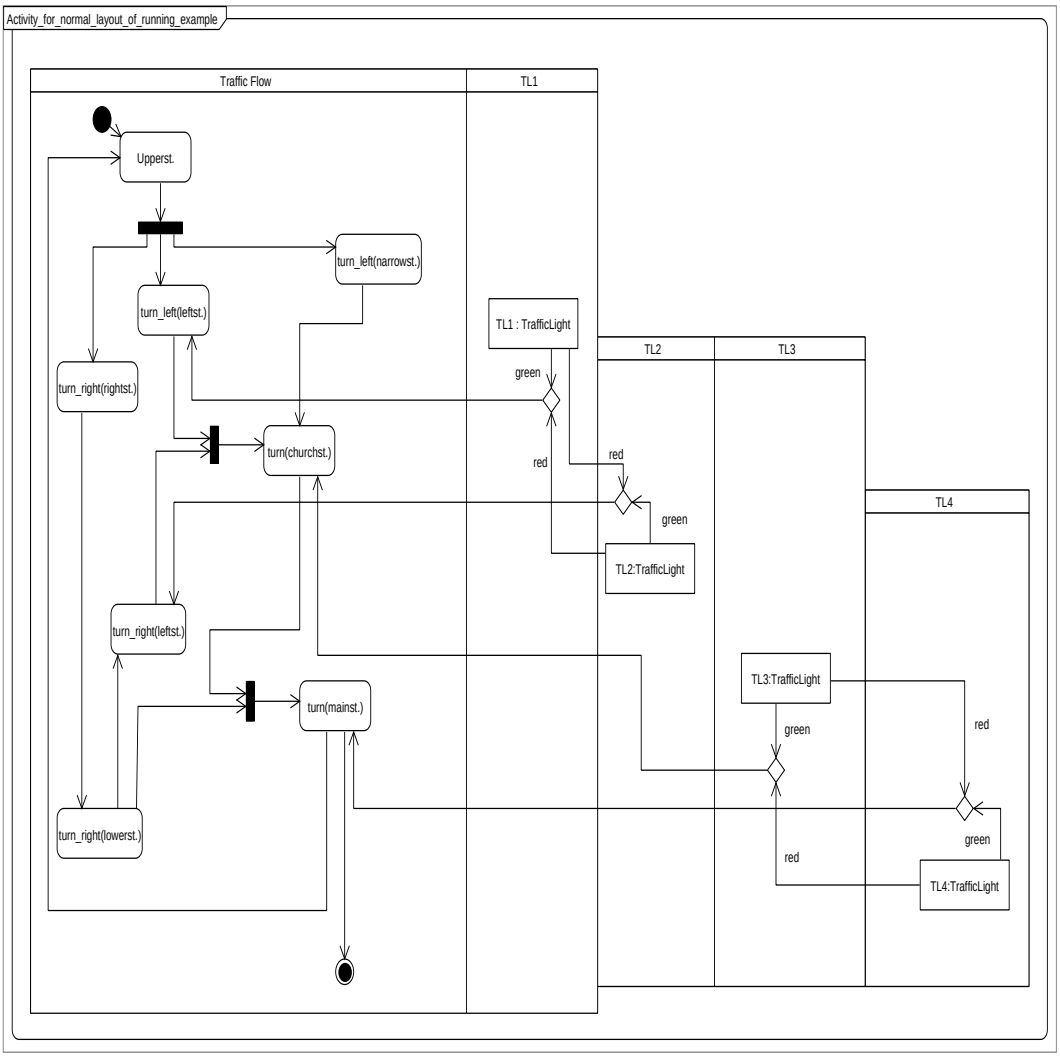



**Figure 3.5:** Normal behavior for the particular routes of the UTCS case1.

As stated in the UML specification [80], an activity allows a very readable modelling of concurrency and of all elementary programming concepts namely: sequence, branch, loop, swimlane, fork and join. Activity diagrams are usually associated to a class and, as such, they model the operations flow inside the class. This flow can depend on internal or external events. Nevertheless, the activity diagram also allows a hierarchical decomposition, through the use of sub-activity states, and it can model several classes related by class aggregation. Through the use of external events we can even synchronize several activity diagrams.

With the help of an example, we describe why the design information used for evolution. The UTCS for an ambulance case, can be described the flow of the part of it by an activity diagram shown in figure 3.5, that opens a window as dynamic view of the concrete code that describe the flow in normal case. This activity diagram defines the interconnections among roads and crossroads and swimlanes that express the dependencies among roads

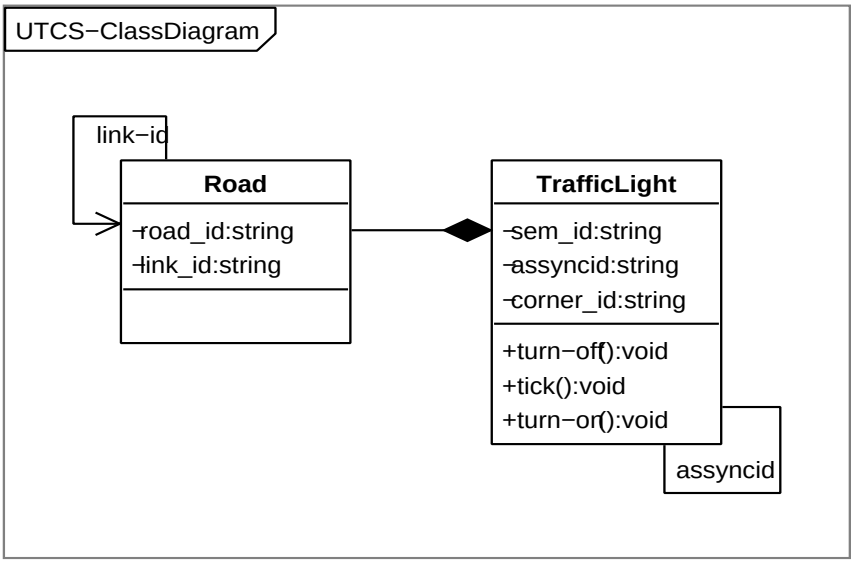



**Figure 3.6:** Class diagram of the urban traffic control system.

and traffic lights. Therefore the information derived by these diagrams must go together with the system code as meta-data.

The design information realizes the structure explicit view that describes the implicit components and their connections in form of *class diagram*. The class diagram in figure 3.6 could represent the structure part of the simplification of a UTCS case1. Notwithstanding that, it can help us in abstract global view how the concrete-level component structured and the relation between them. That is, all crossroads in the map are connected by a path of roads. An urban traffic control system that respects such criteria is consistent with the basic requirements that a livable city must have. Therefore, the software engineer that designs an UTCS should guarantee that such criteria will be respected. These requirements define the *consistency* of the system design.

The structure explicit data the describe how the instances connect, can be described by the deployment diagram as shown in figure 3.7, that defines the interconnections among roads and crossroads. A statechart expresses the dependencies among traffic lights as shown in figure 3.8. These diagrams well describe the system structure and behavior and its evolution should pass through these data to be well planned and integrated with the existing code. Therefore the information derived by all these diagram must go together with the system code as meta-data.

In this subsection we introduced the successful behavior by presenting an explicit global view of the UTCS case1 by using design information. The successful behavior describes the implicit information in the implementation code of motivation case into three UML diagrams for simplicity as follows:

**class diagram** - explicit information that describe the general static view of the whole sample case.

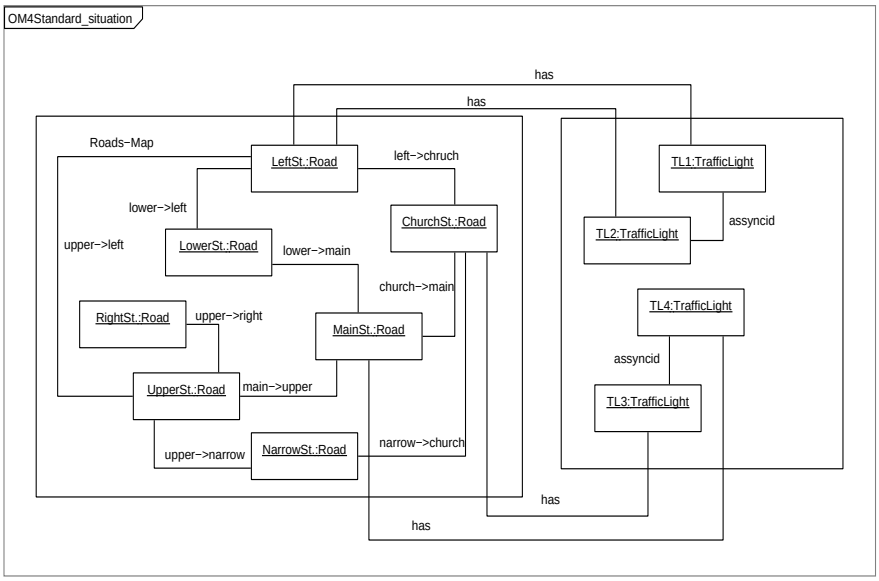



**Figure 3.7:** Normal layout: a) the static view .

**deployment diagram** - provides an global view for the instance connections

**statechart diagram** - explicit the dynamic behavior for specific class in the system.

**activity diagram** - explicit the dynamic information in form of activity and action for the whole system at particular state.

## 3.4. Lightweight Formalisation of Design Information

In the following we describe the design information formalism that is used as a cornerstone for our approach. Our approach to evolution uses *design information* as knowledge bases for getting system evolution. Design information is the data related to the design of the system we want to evolve. UML is the adopted formalism for representing design information.

The taxonomy of the design information is composed of two sub component *structural information* and *behavioral information* as shown in figure 3.9. *Structure information* describes the realized static structure of the concrete-level in the form of *class diagram* and *deployment diagram*. The *behavior information* describes the realized dynamic behavior of the concrete-level in form of three UML diagrams (*statechart*, *sequence* and *activity*). To this aim, we assume that the design information realizes as follows:

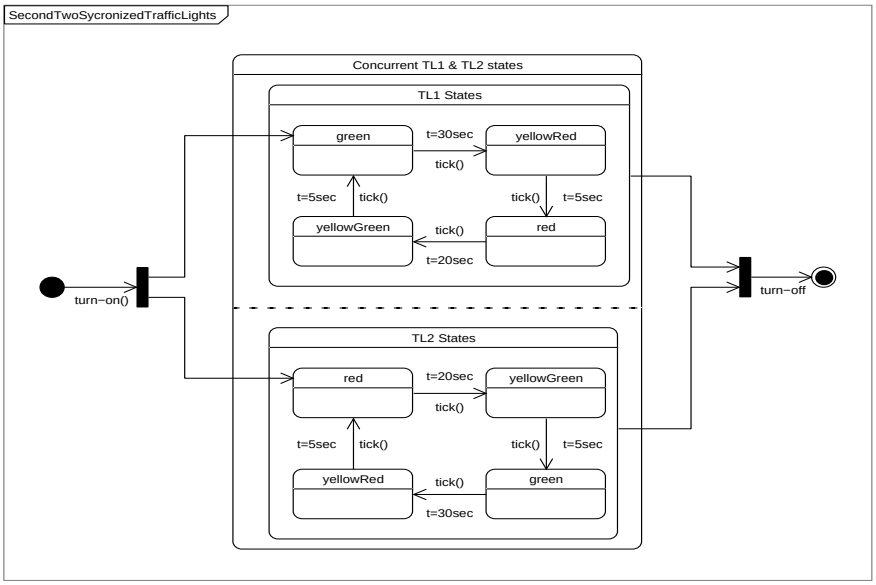



**Figure 3.8:** Normal layout: the behavior view.

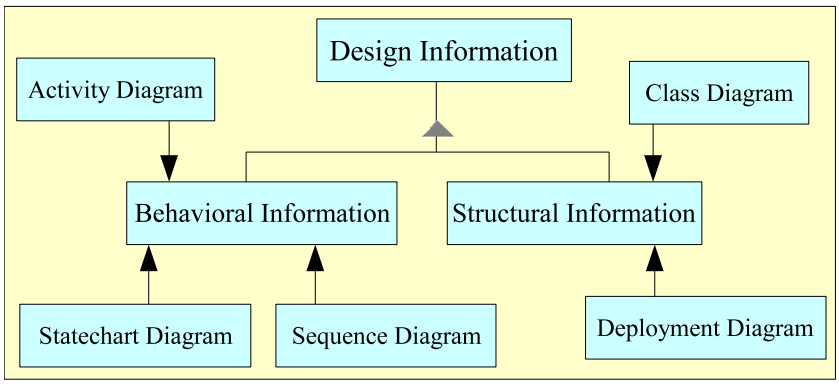



**Figure 3.9:** The taxonomy of design information-abstract level.

**Definition 3.4.1.** Design information ($DI$) is represented as a set of structural and behavioral UML diagrams: DI=$\{< SDI >, < BDI >\}$.

***Where.*** $SDI$ is the structural design information and $BDI$ is the behavior design information. ◊

## 3.4.1. Structural design information

*Structural design information* is an explicit description of the structure of the base-level objects. This includes the number of attributes and their data type. In the following definitions we describe the two structural diagrams as described in figure 3.9.

**Definition 3.4.2.** Structural design information $< SDI >$: $\{CD, DD\}$.

***Where.*** $CD$ is the class diagram and $DD$ is the deployment diagram. ◊

The two types of the structural design information are the *class diagram* and *deployment diagram*. In the following definitions, we define the formalism of both types, then applying the definitions to the structural diagrams realized from the running example.

**Definition 3.4.3.** Class diagram ($CD$) is the formula that describes the system in an abstract level using two components: classes and the relations between them as follows:

$$\text{CD} = \{\Sigma_{i=1}^{m}\ cls_i, \Sigma_{j,k,l}^{n}\ ass_j(cls_k, cls_l)\}.$$

***Where.*** $\forall\ cls_i \in CD$, the $cls_i$ is defined as follows:

$$cls_i =< clsN, \Sigma_{w=1}^{t}\ att_w, \Sigma_{i=1}^{m}\ op_i >$$

For association or relation, $\forall\ ass_i \in$ CD, the $ass_i$ is defined as follows:

$$ass_i =< asstype, \Sigma_{i=1}^{m}\ cls_i, cardinality >, \forall\ cardi_k \in cardinality,$$

the complete description of the $cardi_k$ represents as:

$$cardi_k =< cardtype, \Sigma_{l=1}^{n}\ (cls_l, cls_l.cardno) >$$

**Note:** *cls* is the class and *ass* represents the association between two or more classes.◊

***Example*** 3.4.1. The classes *Road (R)* and *TrafficLight (TL)* as modelled in Figure 3.6 represent the static structure of the running *UTCS* case study. The both classes can specified using the following formal definition:

$$\begin{aligned} UTCS\text{-}CD \ = \ & \{< cls_R, cls_{TL} >, \\ & < ass_{has}(cls_{R_{ins}}, cls_{TL_{ins}}), \\ & ass_{assyncid}(cls_{TL_{ins}}, cls_{TL_{ins}}), \\ & ass_{link-id}(cls_{R_{ins}}, cls_{R_{ins}}) >\}. \end{aligned}$$

△

**Definition 3.4.4.** Deployment diagram ($DD$) is the formula that describes how the system instances interact in the abstract level using two components: class instances (objects) and the object connections as follows:

$$\text{DD} = \{\Sigma_{i=1}^{m}\ obj_i,\ \Sigma_{j,k,l}^{n}\ objrel_j(obj_k, obj_l)\}.$$

***Where.*** *obj* is the objects and *objrel* is the relations between objects. ◊

***Example*** 3.4.2. Figure 3.7 represents another view of static structure of our case study in form of *deployment diagram*. We defined the objects and their relations according to the definition (3.2.4) as follows:

$$\text{UTCS} - DD = \{UTCS - DD_{objs}, UTCS - DD_{objrels}\}.$$

***Where.***

$$\begin{aligned}\text{UTCS} - DD_{objs} \quad = \quad & \{obj_{leftSt:road}, obj_{lowerSt:R}, obj_{churchSt:R}, \\ & obj_{rightSt:R}, obj_{mainSt:R}, obj_{upperSt:R}, obj_{narrowSt:R}, \\ & obj_{TL1:TL}, obj_{TL2:TL}, obj_{TL3:TL}, obj_{TL4:TL}\},\end{aligned}$$

and

$$\begin{aligned}\text{UTCS} - DD_{objrels} \quad = \quad & \{objrel_{leftSt \rightarrow churchSt}(leftSt, churchSt), \\ & objrel_{church \rightarrow main}(churchSt, mainSt), \\ & objrel_{narrow \rightarrow church}(narrowSt, churchSt), \\ & objrel_{upper \rightarrow narrow}(upperSt, narrow), \\ & objrel_{main \rightarrow upper}(mainSt, upperSt), \\ & objrel_{upper \rightarrow right}(rightSt, upperSt), \\ & objrel_{lower \rightarrow main}(lowerSt, mainSt), \\ & objrel_{lower \rightarrow left}(lowerSt, leftSt), \\ & objrel_{upper \rightarrow left}(leftSt, upperSt)\}\end{aligned}$$

**Note:** *UTCS-DD*$_{objs}$ is listing all the object instances described in figure 3.7, *UTCS-DD*$_{objrels}$ is listing all the possible relations between object instances described in *UTCS-DD*$_{objs}$, $R$ is the road class, and $TL$ is the traffic Light class. △

## 3.4.2. Behavioral design information

*Behavioral design information* describes the computations and the communications carried out by the base-level objects. It includes objects behavior, collaboration between objects, and the state of the objects. In the following, we define the syntax formula for the behavior diagrams, that we will use in the next chapters.

**Definition 3.4.5.** Behavioral design information is described as:

$$< \text{BDI} >: \{StD, SeD, AcD\},$$

***Where.*** *StD* is the statechart diagram , *SeD* is the sequence diagram, and *AcD* is the activity diagram. ◊

In the following, we define in details the behavioral design information components, Moreover, how this definition is realized by examples.

**Definition 3.4.6.** Statechart diagram (*StD*) describes the classes behavior through the movement of a class from a state to another under event/condition. The statechart diagram has the following elements:

$$\text{StD} = \{\Sigma_{i=1}^{m}\ St_i, \Sigma_{j}^{n}\ styp_j, \Sigma_{l=1}^{m}\ trans_l\}.$$

***Where***. the *St* is the states, *styp* is the state types, the possible types are (simple state, concurrent state, fork state, and join state), and *trans* is the transition from state to another.
**Note:**

$$\forall\ cls_i \in CD\ \exists\ std_i \in StD\ s.t.$$
$$(cls_i, std_i) = \ < \Sigma_{l,k,j=1}^{m}\ (cls_i.St_j, cls_i.styp_k, cls_i.trans_l) > .$$

◊

***Example*** 3.4.3. The statechart is shown in figure 3.8, specifies the synchronization between two traffic lights (*TL1*,*TL2*). After applying the definition 3.4.6, we get the following results:

$$\text{StD}_{con_{TL1\&TL2}} = \{St_{TL1}, St_{TL2}, styp_{concurrent}(TL1, TL2), trans_{TL1}, trans_{TL2}\}.$$

***Where***. The state of *TL1* consequently is organized as follows:

$$\text{St}_{TL1} = \{green, yellowRed, red, yellowGreen\}.$$

The possible transitions for the *TL1* states are realized in the following formula:

$$\begin{aligned}\text{trans}_{TL1} \ = \ & \{t_1(tick(), S_{green} \rightarrow S_{yellowRed}, t = 30sec),\\ & t_2(tick(), S_{yellowRed} \rightarrow S_{red}, t = 5sec),\\ & t_3(tick(), S_{red} \rightarrow S_{yellowGreen}, t = 20sec),\\ & t_4(tick(), S_{yellowGreen} \rightarrow S_{green}, t = 5sec)\}.\end{aligned}$$

Then, we define the consequence states and transitions of the TL2 as follows:

$$\text{St}_{TL2} = \{red, yellowGreen, green, yellowRed\}.$$

The possible transitions for the *TL2* states are realized in the following formula:

$$\begin{aligned}\text{trans}_{TL2} \ = \ & \{t_1(tick(), S_{red} \rightarrow S_{yellowGreen}, t = 20sec),\\ & t_2(tick(), S_{yellowGreen} \rightarrow S_{green}, t = 5sec),\\ & t_3(tick(), S_{green} \rightarrow S_{yellowRed}, t = 30sec),\\ & t_4(tick(), S_{yellowRed} \rightarrow S_{red}, t = 5sec)\}.\end{aligned}$$

Since we describe the statechat of figure 3.8, we extract the general form of that example as follows:

$$\begin{aligned}\text{StD}_{figure\ 3.8} &= \{S_{Sstate}, trans_{turn-on}(Sstate, Fstate),\\ &\quad S_{fork} concurrent_{(TL1_{green}, TL2_{red})},\\ &\quad S_{join}(TL1, TL2), trans_{turn-off}(S_{join}, Fstate)\}.\end{aligned}$$

△

**Definition 3.4.7.** Sequence diagram ($SeD$) describes the system behavior as a connection of system components through messages between them. This diagram has the following elements: *objects* and *messages* that describe the events flow between the objects.

$$\text{SeD} = \{\Sigma_{i=1}^{m}\ obj_i, \Sigma_{j}^{n}\ mess_j, \Sigma_{l=1}^{m}\ op_l\}.$$

***Where***. $obj$ is an object that participates in specific scenario, $mess$ represents the message between two object instances. Finally, the $op$ is the operation that will be fire in the specific object. ◊

**Definition 3.4.8.** Activity diagram ($AcD$) describes a particular behavior of the systems. The activity diagram has the following elements:

$$\text{AcD} = \{\Sigma_{i=1}^{m}\ astyp_i,\ \Sigma_{j}^{n}\ sl_j,\ \Sigma_{l=1}^{m}\ trans_l\}.$$

***Where***. the $astyp$ is the activity state types, the possible types are (action state, flow state, fork state, merge state, branch state, and join state), $sl$ is the swimlane area for specific object, and $trans$ is the transition from state to another or to object instances. ◊

***Example*** 3.4.4. The activity diagram in figure 3.5 shows the normal flow at a set of roads through set of synchronized traffic lights. We realize the formalism of the normal behavior of the activity diagram as follows:

$$\begin{aligned}AcD_{UTCS_{nb}} &= \{< astyp_{initialS} >, < astyp_{actionS} >, < astyp_{forkS} >,\\ &\quad < astyp_{joinS} >, < objNs >, < mergeNs >, < sl_{set} >,\\ &\quad < trans_{set} >, < astyp_{finalS} >\}.\end{aligned}$$

***Where***. In the following, the action states are realized as follows:

$$\begin{aligned}< \text{astyp}_{actionS} > &= \{AS_{go-upperst}, AS_{turn-left(narrowst)}, AS_{turn-left(leftst)},\\ &\quad AS_{turn-right(rightst)}, AS_{turn-right(lowerst)}, AS_{turn-right(leftst)},\\ &\quad AS_{turn(churchst)},\ AS_{turn(mainst)}\}.\end{aligned}$$

Next, the fork state is described in details as follows:

$$< \text{astyp}_{forkS} > = \{FS_{upper \rightarrow narrow},\ FS_{upper \rightarrow left},\ FS_{upper \rightarrow right}\}.$$

After that, this figure includes two join states, that are realized as follows:

$$< \text{astyp}_{joinS} > \;=\; \{JS_{(Uleft,Lleft)\rightarrow Church}, JS_{(Church,Lower)\rightarrow Main}\}.$$

There are four object nodes are distributed through four swimlanes as follows:

$$< \text{objNs} > \;=\; \{objn_{TL1},\ objn_{TL2},\ objn_{TL3},\ objn_{TL4}\}.$$

Moreover, there are four *merge states* described in details as follows:

$$\begin{aligned} < \text{mergeNs} > \;=\; & \{MNS_{left1-has-tflow}(TL1_{green} \& \ TL2_{red}), \\ & MNS_{left2-has-tflow}(TL1_{red} \& \ TL2_{green}), \\ & MNS_{church-has-tflow}(TL3_{green} \& \ TL4_{red}), \\ & MNS_{main-has-tflow}(TL3_{red} \& \ TL4_{green})\}. \end{aligned}$$

The swimlanes are realized as follows:

$$< sl_{set} > \;=\; \{sl_{tflow},\ sl_{TL1},\ sl_{TL2},\ sl_{TL3},\ sl_{TL4}\}.$$

All the possible transitions illustrated in figure 3.5, are realized as follows:

$$\begin{aligned} < \text{trans}_{set} > \;=\; & \{trans_{IS}(IS_{initialS}, AS_{upperst}), \\ & trans_{FS_1}(AS_{turn-left(narrowst)}, AS_{turn-left(leftst)}, AS_{turn-right(rightst)}), \\ & trans_{2AS}(AS_{turn-right(rightst)}, AS_{turn-right(lowerst)}), \\ & trans_{2AS}(AS_{turn-right(lowerst)}, AS_{turn-right(leftst)}), \\ & trans_{JS1}([AS_{turn-left(leftst)}, AS_{turn-right(leftst)}], AS_{turn(churchst)}), \\ & trans_{JS2}([AS_{turn-right(lowerst)}, AS_{turn(churchst)}], AS_{turn(mainst)})\}. \end{aligned}$$

**Note:** *IS* is the initial state, *AS* is the action state, *JS* is the join state, *MNS* is the merge node state, *sl* is the swimlane, *tflow is the traffic flow*, and *FS* is the final state. △

## 3.5. Summary

In this chapter we presented the role for using design information as a base for driving software evolution. Design information provides structural and behavioral views of the implementation code. Design information enables the developer to explicit the structural part of the concrete-level by providing the class diagrams. To explicit the behavior of the code then the design information provide the statechart, activity diagrams or collaboration diagrams. The main advantage of the design information is to give the system simpler as a base to understand, change, and maintain.

We have shown some methodologies and design information that are inadequate for system evolution either because they require a prior knowledge or they have no explicit mapping between design and runtime. This is the main gap that was addressed using UML diagrams that are explicitly mapped to source code and making the connection maintainable.

A first contribution, presented in this chapter, is the formalisation of the design information. Using this formalism, several explicit structural and behavior of the software systems can be precisely specified to drive the evolution and validation plans. This is the subject of the next chapter.

# 4 Evolution Planning: RAMSES Strategic Processes

In this chapter we describe how design information can be used to drive the software evolution and to maintain code consistency. We are going to explain how to automate the evolution by design information. To have a general middleware to deal with software evolution, the evolution planning cannot be hardcoded into the system. Therefore, the evolution planning aims for a knowledge-driven view of software development.

Actually, the design process is very important to the usability and understandability of the system. The design models and implementation code must be strictly connected. Usually, the early stages of the development, the specifications and the design of the system, are ignored once the code has been developed. This practice cause a lot of problems, in particular when the system must evolve. Recognizing changes at the higher level of abstraction and taking into automatically adapting the UML representation is exactly the motivation for this chapter.

This chapter is organized as follows: Section 4.1, overviews the design information based specification of the dynamic evolution. Section 4.2, illustrates the representation form of the UML at meta-level. Section 4.3, discusses the evolution planning through the evolutionary and consistency checker. Section 4.4 explains which operations can be done for the evolution planning. Finally, section 4.5 explains how to interpret the XMI schema by using script rules.

## 4.1. Introduction and motivation

In the last few years, methodologies to automate part of or the whole software life cycle has been widely studied in the software system development. These methodologies can be used to create and/or maintain software, i.e. they are applicable to all the phases of the software life cycle. Evolution and maintenance are becoming increasingly important in the software development. Automatic techniques to support these phenomena are fundamental to improve the management of unanticipated software evolution.

The design process is very important to the usability and understandability of the system. There are two views of design processes: functional and nonfunctional. Functional requirements present a complete description of how the system will function from the

user's perspective, while non-functional requirements dictate properties and impose constraints on the system.

We summarize the objective of this chapter in the following items:

- how the UML diagrams can be represented in a suitable form at runtime.
- how to plan the evolution to the design information for runtime changes, and how to check the consistency of the modified parts. This means how to provide an evolutionary plan for an event and check the design information after applying the plans.
- how XMI [1] can be managed by using operations-based evolution. How these operations are used to extract some parts of the XMI schemas and adapting them.
- how can the XMI schema interpreter the evolution by using script rules. How script language support the dynamic adaptation of the design information, and how these scripts interpret the dynamic evolution.

In the next section we first describe the representation of meta-data in the form of XMI schemas.

## 4.2. UML Diagrams as Meta-Data

The UML is de facto the standard (graphical) language used during the design process, therefore we consider its diagrams as a good representation of for the design information [11]. Dynamic events are hard to be captured at design-time whereas their occurrence surely involves design information. This problem causes complete redesigning of the software system. Our scope is to simplify the evolution/maintenance mechanism. That is, to render the changes required by the evolution immediately available both to the realized design information.

Software systems are expecting for a mechanisms to face changes in their environment and be able to self-adapt their code and design models when unanticipated events occur. This problem forces a complete redesigning of the software systems when changes occur. OMG[2] introduced a standard specification for UML models and their semantics to model the software systems. Moreover, the OMG [83] has introduced a standard representation for the UML, called XMI [81], that can be used between different software tools. For simplicity, if we consider the design information as a coin, then this coin has two views: (1) UML graphical representation and XMI internal representation (this view easy to computerize by program). The XMI can be used to transfer the design information data from a platform to another. XMI provides a translation of UML diagrams in a form more suitable for run-time manipulation.

1 XML Metadata Interchange, for more details http://www.omg.org/

2 www.omg.org

XMI provides a translation of UML diagrams in a text-based form more suitable for run-time manipulation as shown in the following schema that show the representation for the running example. The XMI standard gives a guideline for translating each UML diagram in XML. Each diagram is assimilated to a graph whose nodes are the diagram's components (e.g., classes, states, actions and so on), and arcs represent the relation among the components. The graph is decorated with XML tag describing the properties of the corresponding UML component.

After that we will illustrate the real result of the translation between UML diagram and XML by using the Poseidon tool. The result XMI schema is showed in the following listing.

```
<?xml version = '1.0' encoding = 'UTF-8' ?> <XMI xmi.version = '1.2'
xmlns:UML = 'org.omg.xmi.namespace.UML' xmlns:UML2 =
'org.omg.xmi.namespace.UML2'
  timestamp = 'Fri May 19 21:36:38 CEST 2006'>
  <XMI.header><XMI.documentation>
    <XMI.exporter>Netbeans XMI Writer</XMI.exporter>
    <XMI.exporterVersion>1.0</XMI.exporterVersion>
    <XMI.metaModelVersion>1.4.3</XMI.metaModelVersion></XMI.documentation>
  </XMI.header>
  <XMI.content>
    <UML:Model xmi.id = 'Im1153ee00m10b4c9feb4amm7e78' name = 'UTCS'
        isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
        isAbstract = 'false'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id = 'Im1153ee00m10b4c9feb4amm7e62' name = 'Road'
            visibility = 'public' isSpecification = 'false' isRoot = 'false'
            isLeaf = 'false' isAbstract = 'false' isActive = 'false'/>
        <UML:Class xmi.id = 'Im1153ee00m10b4c9feb4amm7e4f' name = 'TrafficLight'
          visibility = 'public' isSpecification = 'false' isRoot = 'false'
          isLeaf = 'false' isAbstract = 'false' isActive = 'false'/>

        <UML:Object xmi.id = 'Im1153ee00m10b4c9feb4amm7dfa' name = 'MainSt.'
            visibility = 'public' isSpecification = 'false'>
            .
            .
        <UML:Object xmi.id = 'Im1153ee00m10b4c9feb4amm7deb' name = 'ChurchSt.'
            visibility = 'public' isSpecification = 'false'>
        .
        .
        <UML:Object xmi.id = 'Im1153ee00m10b4c9feb4amm7d3e' name = 'TL2'
            visibility = 'public' isSpecification = 'false'>
        .
        <UML:Instance.linkEnd>
            <UML:LinkEnd xmi.idref = 'Im1153ee00m10b4c9feb4amm7d83'/>
```

```
          <UML:LinkEnd xmi.idref = 'Im1153ee00m10b4c9feb4amm7d6a'/>
          <UML:LinkEnd xmi.idref = 'Im1153ee00m10b4c9feb4amm7cd9'/>
          <UML:LinkEnd xmi.idref = 'Im1153ee00m10b4c9feb4amm7d76'/>
        </UML:Instance.linkEnd>

      .
      <UML2:Region xmi.id = 'Im1153ee00m10b4c9feb4amm7bd4' name = 'Region_4'
      visibility = 'public' isSpecification = 'false'>
          <UML2:Region.subvertex>
               <UML2:State xmi.id = 'Im1153ee00m10b4c9feb4amm7bc3'
                 name = 'red' visibility = 'public' isSpecification = 'false'>
                <UML2:Vertex.outgoing>
.
.
```

The above portion of XMI code translates part of the object diagram showed in Chapter 3 at figure 3.7. In particular, it describes the object named *TL2* and *Main St* and their inter-connection. The instances description of a class is grouped into the XMI tag UML.Object. The two occurrences showed in the above snippet describe respectively the object *TL2* and *Church St* in figure 3.7. The name of the instance is contained in the attribute name, whereas the type of the instance is contained in the sub-tag Class. The *xmi.idref* refers to description of the corresponding class into the class diagram. The has association is described through the tags *UML:Instance.linkEnd* that specify which instances are involved into the association and the tag *UML:Instance.ownedLink* that describes the nature of the association.

At system bootstrap, the XMI representation of design information extracts from the code. In this way we render accessible UML data-model to the meta-objects. Design information manipulation is yielded by creating and parsing XMI representation of the corresponding UML diagrams [49]. In this way we render accessible UML data-model. The XMI parser includes the following features:

- it transforms the UML design information related to the code in specific extracted schemas;
- it declares how to add extensions to the meta-data; and
- it allows self adaptation of the extracted schemas.

The XMI schemas are tightly linked with the implementation code. As described in [49], we can create new UML models from XMI schemas, therefore the evolutionary plan, which is a group of modified XMI schemas, can be reverted into UML diagrams. Basically, this reciprocity between UML diagrams and XMI schemas allows to maintain the connection between design and code.

## 4.3. Evolution and Validation Planning

In this section we explain the aspect of evolution planning and consistency validation, and their relation with design information. A plan describes the way to adapt the software system for runtime changes.

Evolutionary and validation planning is the strategy that is used by RAMSES meta-objects to dynamically evolve and validating the simulation of the software system in from of *design information* for runtime events. For that, we define the formalism for the main role of those objects: *evolution planning*, and *validation planning*. Below, we define these concepts as follows:

**Definition 4.3.1. Evolution planning (EP)**, given a system (**S**), S.t.:

$$\mathrm{S}_{comp} = \{S_{DI}, D_{domain}\},$$

Based on the design information definition, the evolution strategy of the system (S) is realized by the following formula:

$$\mathrm{S}_{ES} = \{S_{SS_{SDI}}, S_{DS_{BDI}}\}.\ \forall\ re_i \in D_{domain}\ \exists\ evolutionary\ plan\ (EP)$$

The generated evolutionary plan is a set of evolutionary actions(EA) related to the structural and behavioral design information of the system. The evolution plan of the runtime event ($\mathrm{re}_i$) is described as follows:

$$\mathrm{EP}_{re_i} = \{\Sigma\ EA_{SS_{SDI}},\ \Sigma\ EA_{DS_{BDI}}\},$$

The adopted system is realized by applying the proposed evolutionary plan (EP) to the original (S), the adopted system we refer as:

$$\mathrm{S}' = \{S_{SS_{SDI}}\ \times\ \Sigma\ EA_{SS_{SDI}},\ S_{DS_{BDI}}\ \times\ \Sigma\ EA_{DS_{BDI}}\}.$$

***Where.*** $S_{DI}$ is the design information of the system S, $D_{domain}$ is the running domain and environment of S, ($S_{ES}$) is the evolution strategy of S, $S_{SS_{SDI}}$ is the static evolution strategy of S, $S_{DS_{BDI}}$ is the dynamic evolution strategy of S. ◊

Evolutionary planning is the strategy to evolve the software system for the changes in it's environment domain. By generating an evolution plan for each runtime event, then apply the generated plan direct to the system and it's design information as shown in algorithm 1. The evolution planning algorithm in details, the inputs of evolution planning are: *the design information of system (S)*, and *the detect runtime event*. The evolution planning output is the new system includes the runtime changes we refer as ($S'$). The evolution planning core is: For each runtime event, the evolution planning generates: *preEVOPlan*[3], *an evolutionary action*, and *PostEvoPlan*[4]. Each evolutionary action has two components:

3 preEVOPlan is the precondition for the evolution planning process, that includes set of condition should be valid before applying the evolutionary action.

4 PostEVOPlan is the set of conditions that describe the evolutionary action applied to the system and leave the system consistent with the changes.

**static evolutionary action**: lists all the generated actions related to the structural design information;

**dynamic evolutionary action**: lists all the generated actions related to the dynamic design information.

The evolution planning checks that the *preEVOPlan* is valid or not. If *preEVOPlan* then apply the evolutionary action strategy types (static and dynamic) to the corresponding components of the system (S). In the following, we describe the *postEVOPlan* as the role of the validation planning and it's algorithm in details.

**Definition 4.3.2. Validation planning (VP)**, VP aims to check the evolved system ($S'$) is consistent thought two types of checking: *horizontal* and *vertical* consistency, and all the evolutionary actions applied. We suppose that, *evolution planning* is the map, s.t. *EP*: S $\longrightarrow S'$. Then, we assume the **VP** is a pair of two boolean components that check the functionality of the **EP** map is consistency evolving the system (S). The validation formula can be described as follows: $VP$=$(HVP, VVP)$, s.t.

$$\text{HVP} = \{IsCons(S'_{DI}), IsApp()\},$$

and

$$VVP = \{< SVVP >, < BVVP >\}.$$

The two elements of $VVP$ described in more details as follows:

$$\begin{aligned}
\text{SVVP} &= \{IsCons(S'_{CD}, UMLS_{CD}) \times \\
&\quad IsCons(S'_{DD},\ UMLS_{DD}), \\
&\quad IsApp(EP_{S'_{SDI}} : S_{SDI} \longrightarrow S_{SDI} \times EA_{SDI})\};
\end{aligned}$$

$$\begin{aligned}
\text{BVVP} &= \{IsCons(S'_{StD},\ UMLS_{StD}) \times \\
&\quad IsCons(S'_{SeD}, UMLS_{SeD}) \times \\
&\quad IsCons(S'_{AcD}, UMLS_{AcD}), \\
&\quad IsApp(EP_{S'_{BDI}} : S_{SDI} \longrightarrow S_{BDI} \times EA_{BDI})\};
\end{aligned}$$

***Where***. $HVP$ is the horizontal validation planning, $VVP$ is the vertical validation planning, $SVVP$ is the vertical validation planning for the structural components of the system, $BVVP$ is the vertical validation planning for the behavior components of the system, *IsCons* is the consistency boolean function, *IsApp* is the applied function of the evolutionary action to the system, and *UMLS* is the UML syntax standard for specific diagram. ◊

The validation planning of the evolved system ($S'$) aims to check the results of the evolution planning map. The validation planning is described through two validation types:

**Data** : $S_{DI}$, $re_i$
**Result**: $S'$ - Evolution planning

/*evolutionary planning processes of the system (S) */
**begin**

- **for each** *Runtime* $(re_i) \in$ *SDomain* **do**
  - generate preEVOPlan;
  - generate an Evolutionary action;
  - generate postEVOPlan
  - $EP_{re_i} = \{\Sigma\ EA_{SS_{SDI}},\ \Sigma\ EA_{DS_{BDI}}\}$;
  - **for each** $EP_{re_i}$ **do**
    - **if** *preEVOPlan valid* **then**
      - /*apply the evolution static strategy to the class diagram and deployment diagram */
      - **repeat**
        - **for** $m \in S_{SDI}$ **do**
          - $EP_{CD} = EA_{SS_{CD}} \times S.CD$;
          - $EP_{DD} = EA_{SS_{DD}} \times S.DD$
        - **end**
      - **until** *All* $\Sigma\ EA_{SS_{SDI}}$ *applied* ;
      - /*apply the evolution dynamic strategy to the statechart diagram, sequence diagram, and activity diagram */
      - **repeat**
        - **for** $m \in S_{BDI}$ **do**
          - $EP_{StD} = EA_{BS_{StD}} \times S.StD$;
          - $EP_{SeD} = EA_{BS_{SeD}} \times S.SeD$;
          - $EP_{AcD} = EA_{BS_{AcD}} \times S.AcD$;
        - **end**
      - **until** *All* $\Sigma\ EA_{DS_{BDI}}$ *applied* ;
    - **else**
      - Create (new preEVOPlan)
    - **end**
  - **end**
- **end**

**end**

**Algorithm 1:** Strategy of the evolution planning.

**syntax validation**: Syntax validation is the vertical validation for both structural and behavioral design information of the system. The *VVP* described in algorithms(2,3). The vertical validation plan checks each model of the system is consistent with the changes made by evolutionary planning. Finally, the syntax validation realized by the value of *VP*.
**Note:** $VP = \{VP_{SDI} \times VP_{BDI}\}$.

**semantic validation**: Semantic validation is the horizontal validation that describes if a model in the system is consistent then all the related models should be consistent too. For example, The *HVP* checks for any changes in the class diagram or the deployment diagram, Then this modification should be reflected into special scenario in case of *sequence diagram* or particular behavior in case of *activity diagram*, as shown in example (4.3.3).

After, defining the evolution and validation planning. We describe the evolution and validation plan describing the changes from the developer point of view for the normal layout described in the chapter 3. We suppose the detected runtime event is (closing *Church street*). Then, the developer suggests an evolution and validation plan to solve the problems related to this event, and with keeping satisfied all the constraint also to the modified plan.

In the following subsections, we describe the proposed evolutionary and validation plans for normal layout described in the chapter 3. The evolutionary and validation plan describe the developer point of view to adapt the realized design information.

### 4.3.1. Evolutionary planning of the UTCS example (3.2)

Informally, the evolutionary plan has to carry out the following actions:

- turn-off the traffic lights [$tf_1$, $tf_2$, and $tf_3$];
- delete the road *church st.* and its *roadlinks* towards the others;
- change the direction from [*upper* $\rightarrow$ *narrow*] to [*narrow* $\rightarrow$ *upper*];
- add new traffic light $tf_5$ to the *upper* at the crossroad between *upper* and *main*;
- move the traffic light $tf_4$ to end of *main* connected with *upper*;
- set new synchronization between $tf_5$ and $tf_4$; and
- delete the $tf_1$,$tf_2$ and $tf_3$ from the statechart and their states.

Applying our evolution planning definition for the above evolutionary plan as follows:

**Data** : $S'_{DI}$, $EP$, $S_{DI}$
**Result**: $S'$ - `Validation planning`

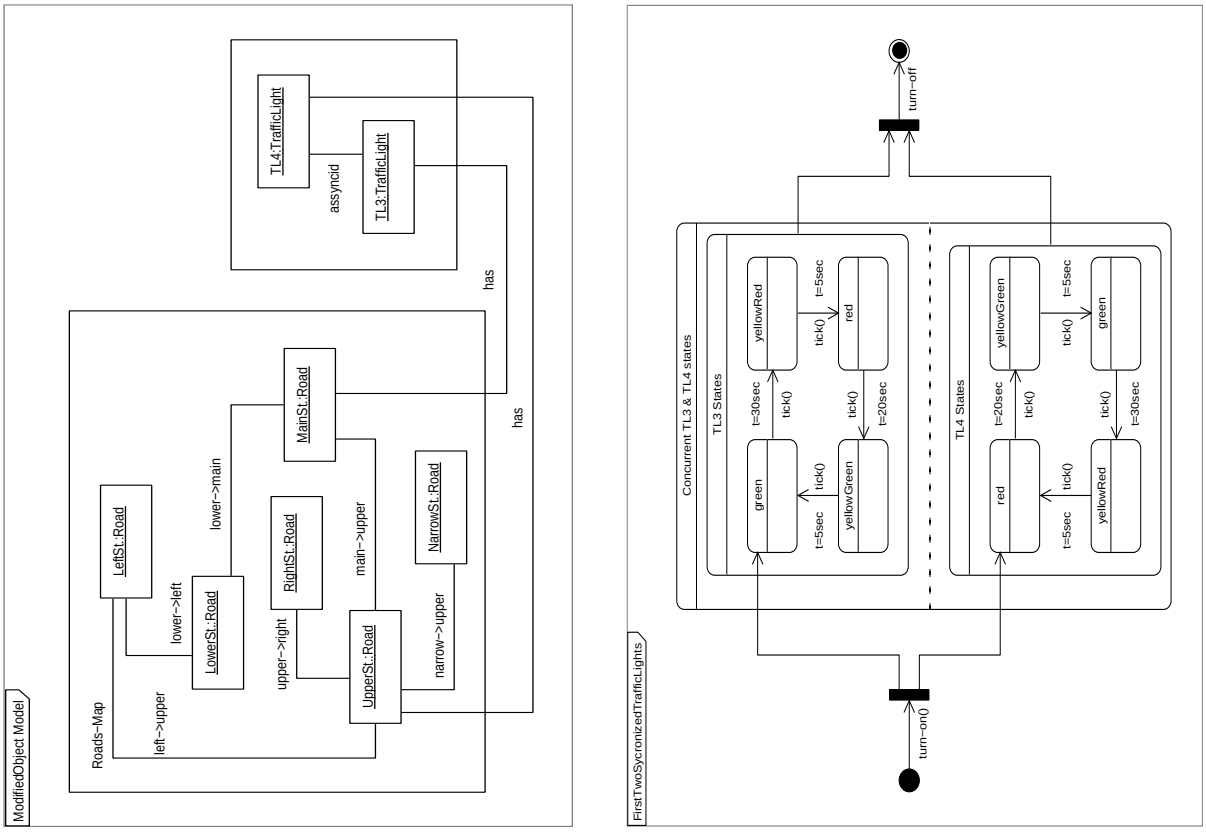
```
/*validation planning description of the evolved system */
begin
    for m ∈ S_SDI do
        if EP finished then
            /*Vertical validation of the static structural components */
            repeat
                for m ∈ S'_SDI do
                    /*SVVP of the class diagram */
                    if S'.CD = S.CD × EA_ssCD then
                        VP_S'CD = true
                    else
                        VP_S'CD = false
                    end
                    /*SVVP strategy of the deployment diagram */
                    if S'.DD = S.DD × EA_ssDD then
                        VP_S'DD = true
                    else
                        VP_S'DD = false
                    end
                end
                VP_SDI = VP_S'CD × VP_S'DD;
            until All S'_SDI scanned ;
        else
            check again EP_S'SDI
        end
    end
end
```

**Algorithm 2:** Strategy of the structural validation planning.

**Data** : $S'_{DI}$, $EP$, $S_{DI}$
**Result**: $S'$ - validation planning

/*validation planning description of the evolved system */
**begin**
  **for** $m \in S_{DI}$ **do**
    **if** *EP finished* **then**
      /*Behavior validation planning of the behavioral design information */
      **repeat**
        **for** $m \in S'_{BDI}$ **do**
          /*BVVP strategy of the statechart diagram */
          **if** $S'.StD = S.StD \times EA_{Bs_{StD}}$ **then**
            $VP_{S'_{StD}} = true$
          **else**
            $VP_{S'_{StD}} = false$
          **end**
          /*BVVP strategy of the sequence diagram */
          **if** $S'.SeD = S.SeD \times EA_{Bs_{SeD}}$ **then**
            $VP_{S'_{SeD}} = true$
          **else**
            $VP_{S'_{SeD}} = false$
          **end**
          /*SVVP strategy of the activity diagram */
          **if** $S'.AcD = S.SeD \times EA_{sB_{AcD}}$ **then**
            $VP_{S'_{AcD}} = true$
          **else**
            $VP_{S'_{AcD}} = false$
          **end**
        **end**
        $VP_{BDI} = VP_{S'_{StD}} \times VP_{S'_{SeD}} \times VP_{S'_{AceD}}$;
      **until** *All* $S'_{BDI}$ *scanned* ;
    **else**
      check again $EP_{S'_{BDI}}$
    **end**
  **end**
**end**

**Algorithm 3:** Strategy of the behavioral validation planning.




(a) Object Model (b) Statechart

**Figure 4.1:** Modified Layout: a) the modified static view b) the modified behavior view.

***Example*** 4.3.1. The evolution planning (EP) of the UTCS example is described in figure 3.1.b. The EP is realized through set of evolutionary actions that is applied to the design information:

$$\mathrm{EA}_{UTCS} = \{EA_{UTCS_{DD}}, EA_{UTCS_{StD}}\},$$

Such that the evolutionary actions related to the deployment diagram are defined as:

$$\begin{aligned}\mathrm{EA}_{UTCS_{DD}} \;=\; & \{UTCS_{DD}.\Sigma_{i=1}^{3}\, tf_i.turn\_off(),\\ & UTCS_{DD}.deleteClsIns(cls.churchSt.),\\ & UTCS_{DD}.deleteClsIns(cls_{churchst.}).roadLink(rl_{left\longrightarrow church},\\ & rl_{church\longrightarrow main}, rl_{narrow\longrightarrow church}, rl_{church-tl3}),\\ & UTCS_{DD}.addClsIns(tf_5, tf_{5_{roadLink}}(rlList)),\\ & UTCS_{DD}.changeDirClsIns(rl_{upper\longrightarrow narrow},\ rl_{narrow\longrightarrow upper})\}.\end{aligned}$$

Then, the evolutionary actions related to statecharts are:

$$\mathrm{EA}_{UTCS_{StD}} = \{createSync(twoGroupSync\ (tf_5,\ tf_4),\ \Sigma_{i=1}^{3}\ deleteTL.tf_i\}.$$

Finally the evolved diagrams described by the following formula:

$$\mathrm{UTCS}_{DI'} = \{UTCS_{DD} \times EA_{UTCS_{DD}}, UTCS_{StD} \times EA_{UTCS_{StD}}\}.$$

***Where.*** $EA_{UTCS_{DD}}$ is the evolutionary actions for the deployment diagram, $EA_{UTCS_{StD}}$ is the evolutionary actions for the statechart diagram, the method *deleteClsIns* is the deletion of a class instances, *addClsIns* is the addition of a class instances, *changeDirClsIns* is the changing road direction, *createSync* is the creation of synchronization protocol between traffic lights instances, and *twoGroupSync* is a type of synchronization between two traffic lights by defining the time cycle of each color for both instances. △

### 4.3.2. Consistency validation of the UTCS example (3.2)

The validation plan has to carry out the following actions:

- check that the remaining traffic lights are synchronized;
- check that every road Link has a connection between two roads;
- check that every association between road instances and traffic lights instances is one to one relation;

Applying our definition of the validation planning to check the syntax of *VVP* is described as follows:

***Example*** 4.3.2. The vertical validation planning of the UTCS example is shown in figure 3.1.b, is described by:

$$\mathrm{VVP}_{UTCS} \ = \ \{VVP_{UTCS_{DD}}, VVP_{UTCS_{StD}}\}$$

such that, the vertical validation planning of the deployment diagram is realized from the example as follows:

$$\begin{aligned} VVP_{UTCS_{DD}} \ = \ & \{\Sigma_{i,j,k}^{m}\ DD_{ClsIns.roadLink_i} = (r_j, r_k), \\ & \Sigma_{i,j,k}^{m}\ DD_{ClsIns.tFLink_i} = (tf_i, tf_k)\}. \end{aligned}$$

The validation reports of deployment calculated by:

$$\begin{aligned} \textbf{If} \quad & (\forall\ rL \in DD_{ClsIns.roadLink_i} \implies rL \neq null)\ \wedge \\ & (\forall\ tfL \in DD_{ClsIns.tFLink_i} \implies tfL \neq null) \\ & \textbf{Then}\ VVP_{DD} = true, (DD\ consistent) \\ & \textbf{Else}\ VVP_{DD} = false, (DD\ inconsistent). \end{aligned}$$

The vertical validation planning of the statechart is realized from the example as follows:

$$\mathrm{VVP}_{UTCS_{StD}} = \{\Sigma_i^m \; StD_{ClsIns.syncP_i} = tf_i.getSyncGroup()\}.$$

The validation report of the statechart is calculated by:

$$\begin{aligned} \textbf{IF} \quad & (\forall\, tSP \in StD_{ClsIns.syncP_i} \implies tSP \neq null) \\ & \textbf{Then}\, VVP_{StD} = true, (evolved\ statechart\ consistent), \\ & \textbf{Else}\, VVP_{StD} = false, (the\ statechart\ inconsistent). \end{aligned}$$

***Where***. $DD_{ClsIns.roadLink_i}$ is the validator operator used to check there is a road link, $DD_{ClsIns.tFLink_i}$ is the validator operator used to check the traffic link instances, and $StD_{ClsIns.syncP_i}$ is the validator operator used to check the traffic light belongs to group of synchronization or not. △

***Example*** 4.3.3. The Horizontal validation planning ($HVP$) of the example is shown in figure 4.1.

$$\begin{aligned} \forall\, tfl_k \in UTCS_{DD_{objrel=tFLink_k}} \; s.t.\; tfl_k = (tf_i, tf_j),\; \forall\, i \; and \; j \implies \\ \exists\, cstate \in UTCS_{StD} \; s.t.\; cstate = concurrentState(tf_i, tf_j). \end{aligned}$$

For instance as in figure 4.1, $HVP$ has the following processes:

$$\begin{aligned} \mathrm{tfl}_1 &= (tf_4, tf_5) \in UTCS_{DD'} \implies \exists\, cstate \in UTCS_{StD} \\ s.t.\; cstate &= concurrentstate\{ \\ & \quad (tf_{4_{green}}, tf_{4_{yellowRed}}, tf_{4_{red}}, tf_{4_{yellowGreen}}), \\ & \quad (tf_{5_{red}}, tf_{5_{yellowGreen}}, tf_{5_{green}}, tf_{5_{yellowred}})\}. \end{aligned}$$

△

## 4.4. Operation-Based Adaptation of Design Information

As said in the previous section, the XMI specifies how UML models are mapped into XML file. Besides this functionality XMI can also specifies how changes can be easily mapped into UML diagrams. Therefore, XMI is very good solution for solving some of the requested requirements for UML evolution.

In this section we define a set of (simple) operations to manipulate the UML diagrams through their XMI representation. These operations could be used to automate the dynamic evolution of the UML diagrams. Design information could be represented in a suitable form to be observed and manipulated. To parse and apply the evolution in XMI schemas, we need specific operations that are able to scan specific part of XMI to modify, delete or update it. In the following subsections we illustrate some operations and its roles for handling the design information.

| Operation | Description |
|---|---|
| **add class** | add an empty class |
| **add attribute** | the addition of an attribute to a class |
| **add method** | the addition of a method to a class |
| **add relation** | the addation of a relation between two classes |
| **delete class** | the deletion of a class |
| **delete attribute** | the deletion of an attribute from a class |
| **delete method** | the deletion of a method from a class |
| **delete relation** | the deletion of a relation between two classes |
| **rename class** | the change of the class name |
| **change attribute** | the changes of an attribute like new name or value |
| **change method** | the changes of a method name, method body |
| **change relation** | the change in the relation like the name the cardinality |

**Table 4.1:** Operation categories for changes in class diagrams

### 4.4.1. Operations taxonomy of structural design information

We realize the adaptation of the structure design information by describing in more details the general operations related to the structural design information components. In table 4.1, we summarize the operation taxonomy for adaptation in class diagram. In that table we have classified the operation categories into three type:

***add type***, the aim is to add new class diagram elements (class, attribute, method and relation[5].);

***delete type***, the aim to delete existing class diagram elements;

***change type***, the aim is to rename class, change attribute value, rename methods, and rename the name of the relations or change the cardinality values.

Now we are going to describe the operation taxonomy for adaptation the second component of the structural design information- *deployment diagram*. In figure 4.2, we illustrate the operations used to evolve this kind of static diagram. To avoid the concurrency we can classify the adaptation of the deployment diagram related to objects and relations as follows:

**objects** - the operations related to evolve objects are: adding new object, delete existing one, and rename object;

**relations** - the operations related to evolve relations are: adding new relation, delete existing one, and rename the relation.

5 We are using the term relation in general to describe all the possible relation for the class diagram such as: *dependency*, *association*, *directed association*, *aggregation*, and *composition*

| Operation | Description |
|---|---|
| **add object** | add an empty class instance (object) |
| **add relation** | the addition of a relation between two objects |
| **remove object** | the deletion of an object |
| **remove relation** | the deletion of a relation |
| **change object** | the change of the name of an object |
| **change relation** | the change the name of the relation |

**Table 4.2:** Operation categories for changes in deployment diagram

| Operation | Description |
|---|---|
| **add state** | the addition of a new state |
| **add transition** | the addition of a transition between two states |
| **add region** | the addition of a new region to the concurrent states |
| **add fork state** | the addition of a fork state to distributed the flow to many states |
| **add join state** | the addition of a join state to connect the distributed states to a state |
| **delete state** | the deletion of a state |
| **delete transition** | the deletion of a transition between two states |
| **delete fork state** | the deletion of a fork state |
| **delete join state** | the deletion of a join state |
| **change state** | the change of a state name |
| **change transition** | the change of a transition label and arguments |
| **change fork state** | the change of a fork state name |
| **change join state** | the change of a join name |

**Table 4.3:** Operations categories for changes in statechart diagram

### 4.4.2. Operations taxonomy of behavioral design information

In this subsection, we describe the required operations to evolve the realized behavior design information diagrams. Firstly, we summarize the operations taxonomy for adaption the statechart diagram. To drive the evolution for the statechart you need specific operations, that realize the different states types and the transition. In figure 4.3, we describe the evolution to the realized statechart diagram related to the main components of the statechart as follows:

**state-kind** - the required operations to evolve state-kind[6] in general are: create new state, delete a state or modify a state.

**transition** - The operations related to the transition are: create new transition, delete a transition or modify the arguments of exiting one.

6 We use term state-kind to describe all the existing state types in normal statechart such as: simple sate, composite state, concurrent state, fork state, and join state.

| Operation | Description |
|---|---|
| **add object** | add an object to the sequence platform |
| **message** | the addition of a message to an object |
| **delete object** | the deletion of a objects and all the connected message |
| **delete message** | the deletion of a message |

**Table 4.4:** Operations categories for changes in sequence diagram

Secondly, we describe the operations taxonomy for adaptation in the realized sequence diagram. In figure 4.4, we summarize the main operations related to the statechart elements as follows:

**object** - the operations related to the object are: adding new object or deleting existing one;

**message** - the operations related to the message[7] element are: adding new message or deleting a message.

Finally, we describe the operation taxonomy for adaptation in the realized activity diagram. In figure 4.5, we summarize the operation related to evolve the activity diagram elements as follows:

**activity** - the operation related to the activity object is: adding or deleting an activity state[8];

**decisions** - adding or deleting (branch state or merge state);

**signal** - the addition or deletion of the fork or join states;

**swimlanes**[9] - the addition or deletion a swimlane.

## 4.5. Interpreting the Evolution by Using Script Language

Adaptation in the design information is driven by adaptation rules as script. These adaptation rules can be related to either the evolution and validation. Evolutionary rules are specified and selected, according to the detected event. To detect which rule to apply, a set of guard statements may be included to the rule, to be evaluated when the rule is triggered. Consistency rules, which checks the consistency of the modified XMI schema, and a set of conditions, are triggered immediately upon parsing the evolutionary plans and the modified XMI schema.

7 The term message describe all the different kinds of message used in the sequence diagram such as: simple messages, special messages to create or destroy objects, and message responses.

8 the activity state term is used to describe the action state and concurrent action state.

| Operation | Description |
|---|---|
| **add action state** | add an empty class |
| **add object flow state** | the addition of an object flow state |
| **add transition** | the addition of a transition between two states |
| **add swimlane** | the addition of a swimlane |
| **add branch state** | the addition of a branch state |
| **add merge state** | the addition of a merge state |
| **add fork state** | the addition of a fork state |
| **add join state** | the addition of a join state |
| **delete action state** | delete an empty class |
| **delete object flow state** | the deletion of an object flow state |
| **delete transition** | the deletion of a transition between two states |
| **delete swimlane** | the deletion of a swimlane |
| **delete branch state** | the deletion of a branch state |
| **delete merge state** | the deletion of a merge state |
| **delete fork state** | the addition of a fork state |
| **delete join state** | the addition of a join state |

**Table 4.5:** Operations categories for changes in activity diagram

Ruby is an interpreted scripting language for quick and easy object-oriented programming. It has many features to process text files and to do system management tasks. It is simple, straightforward, and extensible. Moreover the Ruby code is easily integrable with C and C++ code [102, 75]. Ruby is an interpreted (immediately executable), scripting, pure object-oriented language, which can masquerade as a procedural language, portable, untyped, automatic garbage collection. Ruby includes advanced object-oriented concepts and features such as: singleton method, mix-in rather than multiple-inheritance, operator and method overloading, exception handling, iterators and closures, meta-class, built-in pattern-matching (like Perl).
The main advantages of Ruby are:

- the ability to examine (introspection) aspects of the program from within the program itself.
- dynamic-typing, modules and mix-in classes.
- everything is an object;
- variables are object attributes
- every function is a method

Ruby scripts are able to edit the rule set and have them reinterpreted to support the dynamic addition of new rules or changes. In the rest of this subsection we illustrate an example of an evolutionary and consistency rule. To automate the design information adaptation, the described rules have to be implemented as scripts (e.g., Ruby scripts)

that can be invoked during the system evolution. In the following, we present some portions of the Ruby scripts necessary for adapting our test case.

### Ruby rule for describing planning evolution

Informally, an evolutionary rule can be regarded as an instruction or authority for a manager to execute actions on design information to achieve an objective or execute a change. An evolutionary rule, in the form of Ruby script, is usually made up of an event specification that triggers the rule, which is often fired as a result of a monitoring operation, an action to perform in response to the trigger, and target object that is part of the XMI schema upon which the action performed.

In the following Ruby script we are interpreting the part of UTCS. The aims of this rule is to extract the nodes related to the object diagrams form the XMI schemas by using *getObjectDiagram*. Then it adapts the *link-id* of each road as specified by map by using two methods (*getAllInstancesOf* and *setAttributeValue*). The second part of the rule is used to add new instance of class *Traffic Light* to the new crossroads by asking for the name of the instance, adding the new instance to the object diagram nodes. Finally, by using *setAttributeValue* we fill the attributes of the added instance.

```
def plan_inaccessible_road(r, map, tls, xmi_schema)
   # Planning system adaptation when the part of road r is inaccessible .
   puts r
   puts map
   puts tls

  od = xmi_schema.getObjectDiagram()
   # Adapt the link-id of each road as specified by map.
  for rr in od.getAllInstancesOf("Road")
    if map[rr.getAttributeValue("road-id")] != nil then
      rr.setAttributeValue("link-id", map[rr.getAttributeValue("road-id")])
    end
  end

   # Traffic lights in tls must be added at new crossroads.
  tls.each_key {|name|
    theTL = od.addInstance(name, "Traffic Light")
    tls[name].each_key {|attribute|
      theTL.setAttributeValue(attribute, tls[attribute])
    }
  }
end
```

The plan consists of modifying the object model and the statechart of the system, so that, (i) all the roads next to the inaccessible road change their flow direction according to the planner information, (ii) all the traffic lights near the inaccessible street are removed, (iii) new traffic lights are introduced at new crossroads created after the changes in the flow direction, the behavior of such traffic lights will be synchronized, and (iv) at last the inaccessible road is removed from the system (it will be reintroduced when it is accessible again).

In our test case the closure of *Church street* is managed by the following invocation:

```
plan_inaccessible_road("Church St.",
  { "Left" => "Upper", "Upper" => "Right",
    "Narrow" => "Upper" },
  { "TL-Upper" => { "sem_id" => "s1-U1",
       "corner-id" => "Upper St. & Main St." },
    "TL-Main" => { "sem_id" => "s2-M12",
       "corner-id" => "Main St. & Upper St." }
  })
```

### Ruby rule for describing validation

The second format of the adaptation rules is the validation rules. These rules are very similar to the evolutionary rules described above, but the rule is used to check the consistency between the original UML diagrams and the modified UML diagrams.

In the following Ruby script we describe a consistency rule used to check the synchronization between two traffic lights. The class diagram extracted from the XMI schemas by using the method *getClassDiagram*, and the result stored in *CD*. Then by using method *getClass* the script rule extracts the *Traffic Light* type from the class diagram. After that, it checks the consistency by checking the synchronize attribute (*asssyncid*) value between every couple of traffic light. This rule returns a boolean to illustrate if the traffic lights are synchronized or not.

```
def all_synchronized?
    # Synchronized traffic lights are really synchronized?
  CD = xmi_schema.getClassDiagram()
  TLClass = CD.getClass("TrafficLight")
  synchronized = true
  for TL1 in TLClass.getAllInstances()
    unless (tl = TL1.asssyncid[9]) then
      TL2 = TLClass.getInstance(tl)
      synchronized &&= synchronized?(TL1, TL2)
    end
```

```
  end
  return synchronized
end
```

We describe only how a script langauge can be used to construct rules for consistency validation and evolution. In the next chapter we will go more in details about these adaptation mechanisms.

## 4.6. Summary

Many challenges remain in the development of tactical planning systems that will enable automated changes. XMI is used to represent the UML diagrams. This represents the link between the running application and the design phase. The evolutionary planning can be used to dynamically reconfigure the UML diagrams as a reaction to external events, such as anomalies detected by electronic devices. Moreover, the evolutionary planning can be used to dynamically extend the design information with new features, components, and relations between them. The evolutionary can be used to adapt the behavior as well as the structure that represents the system in XMI. The consistency checker has to be used when we have to check the consistency of dynamic changes carried out by a system on a representative of another schema before effectively performing such changes. The consistency checker has to be used in critical environments to avoid the dire consequences of erroneous and inconsistent updates. We described a set of operations for dealing with the UML diagrams, these operations aims to extract some nodes, modify, and delete the diagrams. Finally, we used the scripting language to manage the evolution of the design information by describing two types of adaptation rules (evolutionary and consistency rules). The rules are written in Ruby.

# 5 The Reflective Middleware: RAMSES At Work

In this chapter, we present our middleware (RAMSES) for dynamically evolving and validating consistency of software systems against run-time changes. The RAMSES middleware is based on a reflective architecture which provides objects with the ability of dynamically changing their behavior by changing its design information, as specified in chapter 3 and in chapter 4. The meta-level is composed of cooperating components, and reifies deployment models, scenarios, activity and statecharts of the system to adapt; then it uses such data for dynamically adapting the software system against environmental changes. The evolution takes place in two steps, each is handled by a special component: a meta-object, called *evolutionary meta-object*, plans a possible evolution against the detected external events then another meta-object, called *consistency checker meta-object* validates the feasibility of the proposed plan before really evolving the system. The meta-objects use the system design information to drive the system evolution.

A software system with a long life span, must be able to dynamically adapt itself to face unexpected changes in its environment avoiding a long out-of-service period for maintenance. A software system consists of several components concurrently executed and exchanging messages. Two aspects control the evolution of such kinds of systems: *behavior*, that is, how a single component behaves, and *dependencies*, that is, the interactions among components. Both of them can be involved in system evolution to comply with changes to system requirements.

A reflective architecture represents the structure that allows the running systems to consistently evolve. In [20] we described a reflective architecture for the evolution of the running systems. In such a framework, the system running in the base-level is the one prone to be adapted, whereas the software evolution is the nonfunctional feature realized by the meta-level. Evolution takes place exploiting design information concerning the running systems.

The cooperative meta-object at meta-level consult the engines, adapting the reified objects for dynamic behavior. Changes to the reified systems can be made at runtime and are immediately reflected to its base-components. The reified components can be developed interactively and incrementally. The evolution and consistency are not hard-coded, neither are they generated. Instead, we build a reflective framework of the base-systems that can be automatically self-adapted for any changes to be active long-life span.

Software engineering however, still makes the unreasonable demand for the running system to be fully specify the changes in advance. Software development asked for the way to modify the base objects at runtime without going to rebuild the application again. It requires a new approach, which adapts the base application as well as on advances in software technology. This new perspective uses the design data for the base application to adapt it, for runtime changes by modifying the reified data. In the rest of this chapter we give a brief overview of the reflective architecture and of its components, we show how these components work and the manipulation of the design information. After that, at section 5.3, we present how the both *evolutionary* and *consistency checker* meta-objects are describing the behavior for the meta-level systems.

This chapter is organized as follows: Section 5.1, describes our reflective architecture (the evolutionary mechanism), and the evolutionary engine related with the architecture and its rules. Section 5.2, describes the reification library to constitute the (reified system) at the meta-level. Section 5.3, describes the behavior of the meta-level. Finally in section 5.4, we survey the main points highlighted in this chapter.

## 5.1. Software Evolution through Reflection

The goal of RAMSES middleware consists of evolving a software system to face environmental and requirement changes and validate the consistency of such an evolution. This goal is achieved by:

- adopting a reflective architecture which dynamically drives the evolution of the software system through its design information when an event occurs; this has been made possible by moving design information from design- to run-time.

- using two sets of rules: *evolutionary* and *consistency*, which describes how evolution takes place and when the system is consistent respectively; these rules are used by the decisional components of the reflective architecture but not by the system that must be evolved;

- adapting the design information of the system and reflecting the changes on the running system.

In the reset of this section we give a brief overview of the reflective architecture and of its components, we show how these components work and the manipulation of the design information.

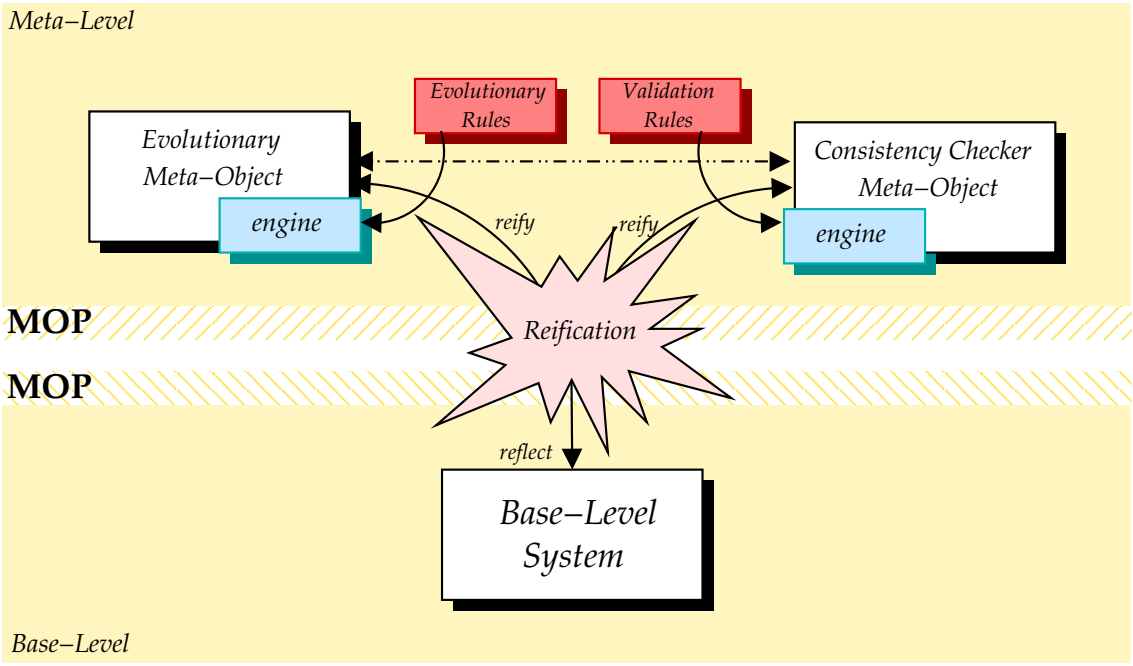



**Figure 5.1:** RAMSES designed for the evolution of software systems.

### 5.1.1. The reflective architecture

To render a system self-adapting[1], we encapsulate it in a two-layers reflective architecture as shown in Figure 5.1, and formally described in definition 5.1.1. The base-level is the system that we want to render self-adapting whereas the meta-level is a second software system which reifies the base-level design information and plans its evolution when particular events occur. By using a reflective architecture, thanks to the transparency and separation of concerns properties of reflection, we can render self-adapting every software system without changing its code. The RAMSES middleware formally is defined as follows:

**Definition 5.1.1.** RAMSES middleware is a reflective architecture, that is constituted of two levels: $RAMSES_{base-level}$ and $RAMSES_{meta-level}$. Through these levels the RAMSES middleware consists of three components as follows:
$\text{RAMSES}_{middleware} = \{\mathcal{R}\text{AMSES}_{base-components}, \mathcal{R}\text{AMSES}_{meta-components}, \mathcal{R}\text{AMSES}_{reification}\}$, s.t.
$\text{RAMSES}_{base-components} = \{BLS_{DI}, BLS_{code}\}$,
$\text{RAMSES}_{meta-components} = \{MObjs_{Evolutionary}, MObjs_{consistency-checker}\}$,
$\text{RAMSES}_{reification} = \{ReifLib, \text{XMI}\}$.

***Where.*** $\text{RAMSES}_{base-components}$ is the underlaying systems (implementation code) or their design information at the base level, $BLS$ is the base-level system (In our case, we assume

1 By the sentence *to render a system self-adapting* we mean that such a system is able to change its behavior and structure in according with external events by itself.

the base-level system is the design information), $\text{RAMSES}_{meta-components}$ is the meta-level components that monitor and supervisor the evolution of the base-level system,*Mobjs* is the meta-objects, that represents the main components of the RAMSES meta-level, $\text{RAMSES}_{reification}$ is an intermediate layer that used to reify the base-system components, and *ReifLib* is the RAMSES reification library, that represents the core components of the intermediate layer. ◊

This approach allows two kinds of dynamic evolution: *structural* and *behavioral* evolution. For instance, the following design information is related to the base-level system:

- *deployment model*, which describes objects and their relations; this model represents the structural part of the system;
- *sequence diagrams*, which traces system operations between objects (inter-object connection) for each use case at a time; and
- *statecharts*, which represents the evolution of the state of each object (intra-object connection) in the system.

The meta-level is responsible of dynamically adapting the base-level and it is composed of some special meta-objects, called *evolutionary meta-objects*. There are two types of evolutionary meta-objects: the *evolutionary* and the *consistency checker* meta-objects. Their goals consists of consistently evolving the base-level system. The former is directly responsible for planning the evolution of the base-level through adding, changing or removing objects, methods, and relations. The latter is directly responsible for checking the consistency of the planned evolution and of really carrying out the evolution through the causal connection.

Through the causal connection, the base-level system and its design information are reified by using the *reification library* in the meta-level (see section 5.2 for more details). Classic reflection takes care of reifying the state and every other dynamic aspect of the base-level system, whereas the *reification library* provides a reification of the design aspects of the base-level system such as its architecture and the collaborations among its components. The reification mechanisms content is the main difference of our architecture with respect to standard reflective architectures. Usually, reifications represent the base-level system behavior and structure not its design information. *Reification library* can build the representatives of the base-level system design information in the meta-level. Both evolutionary and consistency checker meta-objects directly work on such representatives and not on the real system, this allows a safe approach to evolution postponing every change after validation checks. As described in [19] when an external events occur, the evolutionary meta-object proposes an evolution as a reaction to the consistency checker meta-object which validates the proposal and schedules the adaptation of the base-level system if the proposal is accepted.

### 5.1.2. Decisional engines and evolutionary rule sets

Adaptation and validation components in the architecture are respectively driven by a set of rules which define how to adapt the system according to the detected event and the meaning of system consistency.

To give more flexibility to the approach, these rules are not hardwired in the corresponding meta-object rather they are passed to a sub-component of the meta-objects themselves, respectively called *evolutionary* and *validation engines*, which interpret them. Therefore, each meta-object has two components: (i) the core which interacts with the rest of the system (e.g., detecting external events/adaptation proposals, or manipulating the reified meta-data/applying the adaptation on the base-level system) and it implements the meta-object's basic behavior, and (ii) the engine which interprets the rules driving the meta-object's decisions.

In this chapter, we express both evolutionary and validation rules by using the Ruby scripting language as described in section 4.5. The engine is the Ruby interpreter, which executes the planed rules for evolution or check consistency.

The engines are passive sub-components of the respective meta-objects: evolutionary and consistency checker. The engines carry out the following actions:

- the engines are invoked by the meta-objects when needed, e.g., the validation engine is invoked by the consistency checker meta-object when a proposal of evolution is ready to be validated;
- in agreement with its behavior, the meta-object chooses a rule that the engine interprets;
- through reflective facilities used in the rules, the engine directly access and modify the base-level representation;
- the engine, or better the rules used by the engine, applied to the reifications.

Next, we will explain how these engines work. When an event occurs, the evolutionary engine receives from the evolutionary meta-object all the data related to the occurred event and the rule related to the adaptation required by the event. The rule will refer to the reified design information of the base-level system. The execution of the rule will create the evolutionary plan that the evolutionary meta-object will pass to the consistency checker meta-object. Analogously, the consistency checker meta-object will delegates the validation of the evolutionary plan to its engine. The engine uses the consistency rules and the XMI representation for testing if the proposed evolution can be rendered effective or not.

Both rules and engines working on meta-objects are tightly bound but completely unbound from the rest of the reflective architecture. Therefore, to adapt our approach to use rules specified with a different formalism is quite simple; we have just to substitute the engine with another able to interpret the chosen formalism. Of course, the engines

must be able to interact with the rest of the architecture as described by the following algorithm.

In general, adaptation takes place as follows:

❶ the reification library reifies the base-level design information into their meta-level;

❷ the evolutionary meta-object waits for an event that needs the adaptation of the base-level system; when such an event occurs it starts to plan evolution:

① through the design information of the base-level system, it detects which base-level components might be involved in the evolution; then

② it informs its engine about the occurred event and components involved in the evolution;

③ the evolutionary engine decides which evolutionary rule (or which group of evolutionary rules) is better to apply; then

④ it designs the evolutionary plan by applying the chosen evolutionary rule (or group of rules);

❸ the evolutionary meta-object passes the evolutionary plan to the consistency checker meta-object which must validate the proposed evolutionary plan before rendering the adaptation effective:

① the consistency checker meta-object demands the validation phase to the validation engine;

② the validation engine validates the proposed evolutionary plan by using its validation rules and the base-level system design information.

❹ if the proposed evolutionary plan is considered sound the consistency checker meta-object schedules the base-level system adaptation accordingly with such an evolutionary plan; otherwise the consistency checker meta-object returns an error message to the evolutionary meta-objects restarting the adaptation phase.

The evolutionary plan proposed by the evolutionary meta-object is a manipulation of the design information of the base-level system. The causal connection is responsible of modifying the model of the base-level system according with the proposed evolution. The most important advantage of this approach is that adaptation can take place on non-stoppable systems because it does not require that the base-level system stops during adaptation, but it only needs to define when it is safe to carry out the adaptation.

## 5.2. Reification and Reflection by Using Design Information

We have talked about reifying and reflecting design information of the base-level system. The design information simply feed the meta-level system during system bootstrap and drive its meta-computations during the evolution of the base-level system. Design information was described in chapters 3 and it is used by the middleware described in this chapter.

When an event occurs, the design information related to the base-level entities, that can be involved by the event, are used by the evolutionary and the consistency checker meta-objects for driving the evolution of such base-level entities (as described in the previous algorithm).

Design information identifies which entities are involved by the event (class and state diagrams), their behavior (sequence diagrams) and how the event can be propagated in the base-level system (collaboration diagrams). Therefore introspection and intercession on large systems become simpler than using standard reflective approaches because the design information provide a sort of index on the base-level entities and their interaction.

Moreover design information is the right complement to the base-level system reification build by the standard causal connection. Meta-objects consult and manipulate the design information to get information that otherwise are not easily accessible from the running system, e.g., the collaboration among objects. Design information are also used as a testbed for manipulation because they give an easily accessible overview of global features as inter-objects collaborations.

The reification library is the core component of the whole framework. It is necessary to provide the ability of manipulating the design information of the software system which is abstracting from the XMI details and without coping with the graphical representation. It loads the XMI description of the design information, it allows to extend and modify the UML diagrams, and finally save the modified design information again as a XMI file. We formally define the RAMSES-library as follows:

The reification library provides a uniform approach to the design manipulation. Changing the design representation, the application does not change. The central part of the reification library is the class *Reification* which represents the design information and provides access to the UML diagrams. To use it, create an instance of this class by calling the constructor with the name of the XMI file to load. Then access the UML diagrams by calling *getAllClassDiagram()* which returns an array of all class diagrams, or by retrieving a specific diagram by name using *getClassDiagram(String name)*. The same pattern is used to access object and statechart diagrams. After all modifications have been applied, you can save the reification by calling the method *save()* which writes the XMI representation to a file named according to the name given to the constructor

pre-pended with the string Ramses_.

**Definition 5.2.1.** RAMSES reification library is the link between the RAMSES middleware levels. The RAMSES reification is defined as follows:

$$
\begin{aligned}
\mathrm{RRL}_{\mathcal{R}\texttt{AMSES}.reification} &= \{RRL_{XMI}(CD),\ RRL_{XMI}(DD),\ RRL_{XMI}(SeD),\\
&\quad RRL_{\mathsf{XMI}}(StD),\ RRL_{XMI}(AcD)\},\ s.t.\\
RRL_{\mathsf{XMI}}(CD) &= \{\mathsf{XMI}_{reification}(\mathsf{XMI}CD.getAllClasses()),\\
&\quad \mathsf{XMI}_{reification}(\mathsf{XMI}CD.evolveCD())\},\\
RRL_{\mathsf{XMI}}(DD) &= \{UTCS\mathsf{XMI}_{reification}(\mathsf{XMI}DD.getAllInstances()),\\
&\quad \mathsf{XMI}_{reification}(\mathsf{XMI}DD.evolveDD())\},\\
RRL_{\mathsf{XMI}}(StD) &= \{\mathsf{XMI}_{reification}(\mathsf{XMI}StD.getStateChartDiagram()),\\
&\quad \mathsf{XMI}_{reification}(\mathsf{XMI}StD.evolveStD())\},\\
RRL_{\mathsf{XMI}}(SeD) &= \{UTCS\mathsf{XMI}_{reification}(\mathsf{XMI}SeD.getSequenceDiagram()),\\
&\quad \mathsf{XMI}_{reification}(\mathsf{XMI}SeD.evolveSeD())\},\\
RRL_{\mathsf{XMI}}(AcD) &= \{\mathsf{XMI}_{reification}(\mathsf{XMI}AcD.getActivityDiagram()),\\
&\quad \mathsf{XMI}_{reification}(\mathsf{XMI}AcD.evolveAcD())\},
\end{aligned}
$$

***Where***. $RRL_{XMI}(CD)$ is the reification for the class diagram, $\mathsf{XMI}_{reification}(\mathsf{XMI}CD.evolveCD())$ with *evolveCD* we mean all the possible operations related to the class diagram such as (*removeClass()*, *createNewClass*, *getClass*), $RRL_{\mathsf{XMI}}(DD)$ is the reification of deployment diagram, $\mathsf{XMI}_{reification}(\mathsf{XMI}DD.evolveDD())\}$ is all the possible operations related to the deployment diagram such as (*getInstance*, *removeInstance*, *connect Instance*), $RRL_{\mathsf{XMI}}(StD)$ is the reification for the statechart, $\mathsf{XMI}_{reification}(\mathsf{XMI}StD.evolveStD())\}$ is the all possible operations related to the statechart such as(*getState()*, *removeState()*, *connectState()*), $RRL_{\mathsf{XMI}}(SeD)$ is the reification of sequence diagram, $\mathsf{XMI}_{reification}(\mathsf{XMI}SeD.evolveSeD())\}$ is the set of operations related to sequence diagram such as(*addInstanceObject()*, *removeInstanceObject()*, *addInstanceMessage()*, *removeInstanceMessage()*), $RRL_{\mathsf{XMI}}(AcD)$ is the reification of activity diagram, $\mathsf{XMI}_{reification}(\mathsf{XMI}AcD.evolveAcD())\}$ is the set of evolution operation related to add, remove or update the activity diagram elements (*activity state type*, *message*, *operations*). ◊

***Example*** 5.2.1. The class *ClassDiagram* provides several methods to get, create and remove classes, actors and interfaces. To add a new class to the diagram call *createNewClass(String name, String visibility, boolean abstract)*. To get a reference to an existing class by name, call *getClass(String name)*. If you want to remove a class from the diagram, call *removeClass* either with the name or reference to the class object. The same pattern is used to deal with actors and interfaces. △

***Example*** 5.2.2. The class *ObjectDiagram* can be used to add, remove and connect instances (objects). You can also access instances by name or type with *getAllInstancesOf(String type)* and *getInstance(String name, String type)*. To add a new instance call

*addInstance(String name, UMLType type)* with the name and type (class). To remove an instance call *removeInstance* either with the name and type or with a reference to the instance object. To connect instances in the diagram (an instance references another instance through an attribute) call *connectInstances(Instance$_1$, Instance$_2$, String name)*. Such a connection can be removed by calling *disconnectInstances* with the same parameters. △

***Example*** 5.2.3. The class *StateChartDiagram* provides lots of methods to get, add and remove States, *FinalStates*, *ForkStates*, *InitialStates*, and *JoinStates*. Each type of state can be removed by a given name or instance. You can get a reference to a state with the appropriate *getState* method and add a state to the diagram either by providing an existing instance or the name of a new state. △

## 5.3. Describing the Meta-level Behavior of RAMSES

In this section we describe the behavior of the meta-level by describing the role of the *evolutionary meta-objects*. We present a meta-objects for the evolution of software systems. The role of evolutionary meta-objects proposed in this chapter, shows how to adapt a software system to reflect changes in its running environment. These *meta-objects* depend on well-known techniques for programs to dynamically analyze and modify their own structure. Our meta-objects go together with common reflective software architectures. These meta-objects provide a solution for changing the base-level systems behavior according with changes that occur in its running environment. We present special meta-objects in meta-level at the reflective architecture for providing unusual solution to self-adapt base-level systems runtime behavior. The meta-level consists of two meta-objects: first, an *evolutionary meta-object*, enables a software systems to adapt itself to dynamic changes to its requirements. Second, *consistency checker meta-object*, the responsibility of this meta-object is to check the meta-data is consistent with the evolutionary plans executed by *Evolutionary meta-object*. In this section, we define in details the two *evolutionary meta-objects* to self-adaptation reflective architecture at runtime. The role of the evolutionary and consistency checker meta-objects for describing the adaptation and checking the consistency of the modified meta-data for changes is shown in figure 5.3, in the next subsections, we describe the structure of these meta-objects in details.

The working meta-level behavior of RAMSES illustrates in figure 5.2. The meta-behavior described by the following scenario:

- the meta-behavior components observes the UTCS environments or new requirements, if they capture a runtime event, in our case for example *closing churchSt.* for road maintenance or accident;
- the meta-behavior uses the *RAMSES reification library* for manipulating the base-level design information;

- the meta-level evolutionary processes described in the following example:

***Example*** 5.3.1. Meta-level evolutionary processes described by the role of ($EMObj$). It uses it's evolutionary engine ($EE$) for consulting the following adaptation rules:

$$\text{EMObj}_{EE}(UTCS) = \{< ER1 >, < ER2 >, < ER3 >, < ER4 >, < ER5 >\},$$

The $EMObj$ uses the RAMSES reification library $RRL$ for manipulating and apply the evolution to the UTCS.XMI.

***Where.***

$$\begin{aligned}
< \text{ER1} > &= \{UTCS_{\text{XMI}.DD}(ClsIns.deleteIns(tf_1, tf_2, tf_3)),\\
&\quad UTCS_{\text{XMI}.StD}(ClsIns.setAllOffState(tf_1, tf_2, tf_3)),\\
&\quad UTCS_{\text{XMI}.DD}(ClsIns.deleteAlltFLinkTo(tf_1, tf_2, tf_3),\\
&\quad UTCS_{\text{XMI}.DD}(ClsIns.deleteAlltFLinkFrom(tf_1, tf_2, tf_3))\};\\
< \text{ER2} > &= \{UTCS_{\text{XMI}.DD}(ClsIns.deleteIns(churchSt))\};\\
< \text{ER3} > &= \{UTCS_{\text{XMI}.DD}(ClsIns.deleteAllRoadLinkTo(churchSt)),\\
&\quad UTCS_{\text{XMI}.DD}(ClsIns.deleteAllRoadLinkFrom(churchSt))\};\\
< \text{ER4} > &= \{UTCS_{\text{XMI}.DD}(ClsIns.addIns(tf_5)),\\
&\quad UTCS_{\text{XMI}.DD}(ClsIns.settFlink(tf_5, tf_4))\};\\
< \text{ER5} > &= \{UTCS_{\text{XMI}.StD}(ClsIns.createCState(tf_1, tf_2))\}.
\end{aligned}$$

△

- the meta-level validation processes of the evolved design information described in the following example:

***Example*** 5.3.2. The meta-level validation described by the role of $CCMObj$. The $CCMObj$ consults the following validation rules by using its validation engine($VE$) such as:

$$\text{CCMObj}_{VE}(UTCS) = \{< VR1 >, < VR2 >, < VR3 >\}.$$

***Where.***

$$\begin{aligned}
< \text{VR1} > &= \{\forall\, tf_i, tf_j \in TFLink[]\ s.t.\ tFLink(tf_i, tf_j) \in UTCS_{DD}\}\\
&\quad \Longrightarrow\ \exists\ (tfCS)\ \in\ concurrentstatelist(TFCS[])\\
&\quad s.t.\ tfCS(tf_i, tf_j)\ \in\ UTCS_{StD};
\end{aligned}$$

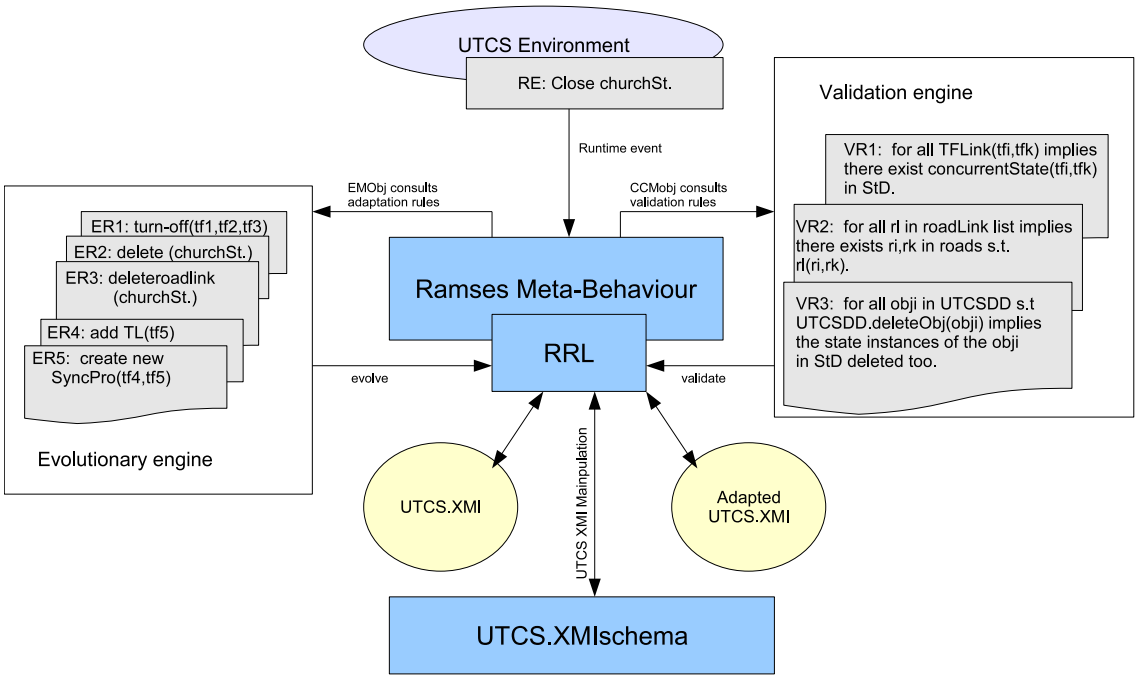



**Figure 5.2:** The RAMSES meta-behavior using UTCS motivation example 3.1.

For instance,

$$\text{TFLink}(tf_4, tf_5) \in UTCS_{DD} \implies \exists\, tfCS(tf_4, tf_5) \in UTCS_{StD};$$

$$< \text{VR2} > = \{\forall\, rl \in UTCS_{DD} \implies \exists\, r_i, r_j \in Roads[] \; s.t.\; rl(r_i, r_j) \in UTCS_{DD}\};$$

$$< \text{VR3} > = \{\forall obj_i \in UTCS_{DD}, for\ each\ evolution\ of\ obj_i.evolveCD \in UTCS_{DD} \implies obj_i.evolveStD \in UTCS_{StD}\};$$

for instance, the traffic light instances $(tf_1, tf_2, tf_3)$ are deleted and their links from the $UTCS_{DD}$, and also, their concrete states and simple states deleted from the $UTCS_{StD}$ as shown in figure 4.1. △

### 5.3.1. Evolutionary meta-object

The main aim of *evolutionary meta-object* is to enable a non-stoppable software systems to self-adapt for the new changes in their requirements and environments.

Several times a (non-stoppable) software system must evolve to adapt itself to the evolution of the environment it is modeling. For running example, in a software system as

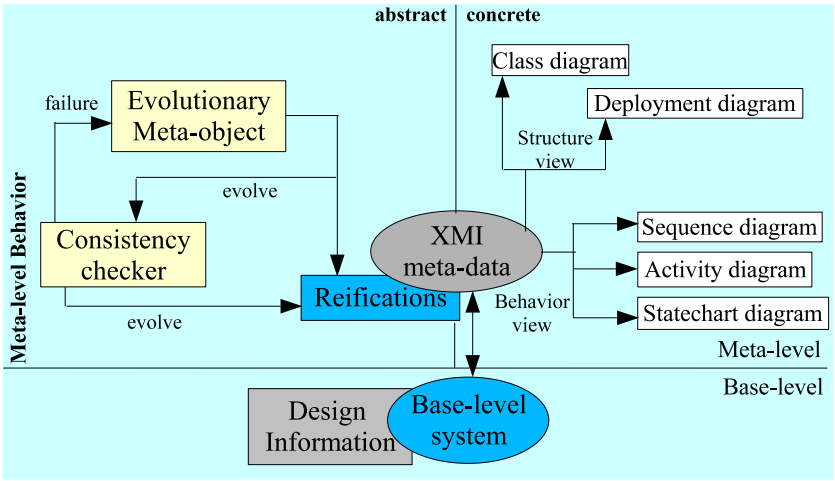



**Figure 5.3:** Evolutionary meta-objects for describing the meta-level behavior of RAMSES middleware.

the UTCS, this involves changes in the overall structure and behavior of the system, i.e., new components to interact with and a reorganization of the components interactions. If changes are planned a lot of time in advance, it is not a problem to take advantage of a moment when the traffic is low, for stopping the UTCS for a while, just the time for the reconfiguration. This is not a feasible solution when the change suddenly happens such as in case of a road interruption due to a road accident or something similar. In a similar case, we cannot stop the normal execution of the UTCS, creating many problems in other parts of the system, to face the unexpected situation.

The roles of *evolutionary meta-object* is involved in the dynamic evolution of a non-stoppable system, obviously the most important is formally described in definition 5.3.1 and is represented by the fact that:

- keeping still for a while (during the reconfiguration) a non-stoppable system could have dire consequences up to and including death.
- the system has to change whenever the environment it is modeling changes;
- changes to the environment can happen at all times, they are outside the control of the system and can not be foresee during the system design;
- reconfiguring the system in accordance with the changes in the environment is not easy and always feasible; moreover the cost of errors can be very high;
- to limit the problems, stopping to the system have to be planned in advance (e.g., roads unpracticability is notified to drivers weeks in advance) and have to be scheduled in noncritical moments (e.g. signals maintenance is carried out during night).

**Definition 5.3.1.** The Evolutionary meta-object has the following roles: (1) detect the runtime events; (2) by using $RRL_{\mathcal{R}\text{AMSES}.reification}$ reify the XMI representation into the meta-level; (3) build an evolutionary rules list for each event, then apply this list to the reified XMI.

$$\begin{aligned}
\mathcal{R}\text{AMSES}_{mc_{EMObj}} &= \{EMObj.observer(runtimeEvent?),\\
&\quad EMObj_{RRL_{\mathcal{R}\text{AMSES}.reification}}(RDI.\text{XMI}),\\
&\quad EMObj.create(ERlist)\}\\
s.t.\ \forall\ re\ \exists&\\
ES_{re} &= \{EMObj.createPreCond(re),\\
&\quad EMObj.createPostCond(re),\\
&\quad EMObj_{EvoEngine}.createERList(ERList)\}.
\end{aligned}$$

The role of meta-object for driving the evolution is realized as follows:

$$\begin{aligned}
&\textbf{If}\quad (EMObj.createPreCond(re) = true)\\
&\quad \textbf{Then MDI} = EMobj.doAdaptation(RDI, ERList)\\
&\quad \textbf{Else}\ EMObj.reCreate(ES').
\end{aligned}$$

***Where.*** *mc* is the meta-level components, *EMObj* is the Evolutionary meta-object, *RDI* is the reified design information by using $RRL_{\mathcal{R}\text{AMSES}.reification}$, *ERList* is the Evolutionary rules list related to the runtime event (re), *ES* is the evolution strategy, *MDI* is the modified design information, ES′ new evolution strategy. ◊

It is fairly evident that to render a non-stoppable system in compliance with the above roles a specific mechanism for adapting the system to environmental changes is needed. Adaptation takes place on a representative of the system. In this way, the adaptation mechanism does not interfere with the current execution of the system it is adapting, preserving the non-stoppable property of the system. Once the adaptation has been completed, the synchronization of the representative with the original system is delegated to the *consistency checker meta-object* for verifying the soundness of the adaptation. Examples of this approach can be found in [103]. *The evolutionary meta-object* describes how to modify the base-objects through their representatives, i.e., the reifications. It directly controls and manipulates the reifications adapting their content in accordance with environmental changes, e.g., adding new operations/components or altering the state of an object. The reifications represent the interface between the *evolutionary* and the *consistency checker*. In the details, when an environmental change takes place this meta-object, following the evolution rules, adapts to the occurred change the representative of the system. The evolution rules adopted by the evolutionary meta-object system design information.

Here we show an algorithm illustrating the basic steps carried out by the evolutionary meta-object. This algorithm is realized by using the pseudo code described in algorithm [4]. The evolutionary meta-object is parametric on the reified system. This

**Data** : base-level system - Reification
**Result**: adapted design information - `evolutionaryMetaObject`

```
/*representative of the base-level aspect */
_representative ← ramses.reification(designInformation);

/*the rules it follows for adaptations */
plans ← evolutionaryMetaObject(selectedRules);
metaData ← ramses.reification(designInformation, _representative (a) );
e ← getRunTimeEvent(runTimeEvent);

/*it retrieves from the plan the rule to face with the event. */
begin
    /*when an external event has occurred returns true then its argument
      refers to the occurred event */
    bool ← on_external_event (event & e);
    while true do
        if on_external_event (e) then
            _plan ← evolutionaryMetaObject(evolutionaryEngine, e);
            newstr ←doAdaptation(structureAdaptation, _plan );
            newbeh ←doAdaptation(behaviorAdaptation, _plan );
        end
    end
    adaptMetaData (newstr,newbeh);

    /*when called an inconsistency has been detected it tries to solve
      such an inconsistency exploiting its plan  */
    inconsistency_detected() ;
    doAdaptation(_representative (a), e);

    /*apply the role of events to representative data */
    adaptation ← _planGetAction(e);

    /*it carries out the adaptation. */
    adaptation(_representative (a), e);

    /*it notifies the attempt of evolution. */
    _representativeChanged ()
end
```

**Algorithm 4:** The role of evolutionary meta-object

means that a class describing the aspect to adapt has also to be provided and to be used to instantiate the evolutionary meta-object. It works on the reifications, i.e., on a representative of the software system. The representative, of course, depends on the aspect of the system this meta-object will deal with.

Another important element are the rules adopted by the algorithm for adapting the representative. These rules are represented by an instance `_plan` of the `plans` class. The `do_adaptation()` of the evolutionary asks `_plan` for the adaptation rule to apply when an event happens. Then the evolutionary applies the adaptation rule to the representative. The evolution takes place when an external event, i.e., an event that has not been generated by the non-stoppable system, happens. The `inconsistency_detected()` method is invoked by the implementation of the consistency checker meta-object.

In a complex system, as shown in the pseudo code above and in figure 5.3, there will be as many instances of the evolutionary meta-object as many aspects of the system have to be adapted. The evolutionary meta-object can be used to dynamically reconfigure a system (not necessarily a non-stoppable system) as a reaction to external events, such as anomalies detected by electronic devices. Moreover, it can be used to dynamically extend a running system with new features, components, and relations between them.

The evolutionary meta-object is applied in the UTCS for creating new synchronization protocol for the list of traffic lights in accordance with adaptation as shown in Figure 5.4. This meta-object interacts with elements of the reifications like roads, traffic lights, synchronization manager, synchronization protocol, and elaborates the photography survey at real-time. The evolutionary uses reification library for reifying the design information of the base-level system in form of XMI. After that, the evolutionary detects an event to turn of same traffic lights at specific traffic node and create new synchronization protocol for the existing traffic lights. Then, evolutionary creates an evolutionary plan to adapt the reifications.

As explained, the evolutionary meta-object observes the environment changes and adapts the base-level representative. The consistency of the representative is validated by the consistency checker meta-object when the evolutionary finishes the adaptation. If the validation fails the control returns to the evolutionary for fixing the problem. The representative adapted by the evolutionary is kept up to date by the reification library.

The evolutionary meta-object provides the following advantages:

- provides an implicit mechanism for dynamically evolving a system;
- provides a uniform way to evolve every aspect of a system, they could also be evolved separately;

and drawbacks:

- the non-stoppable system has an overhead when external events occurs and adaptation is needed;

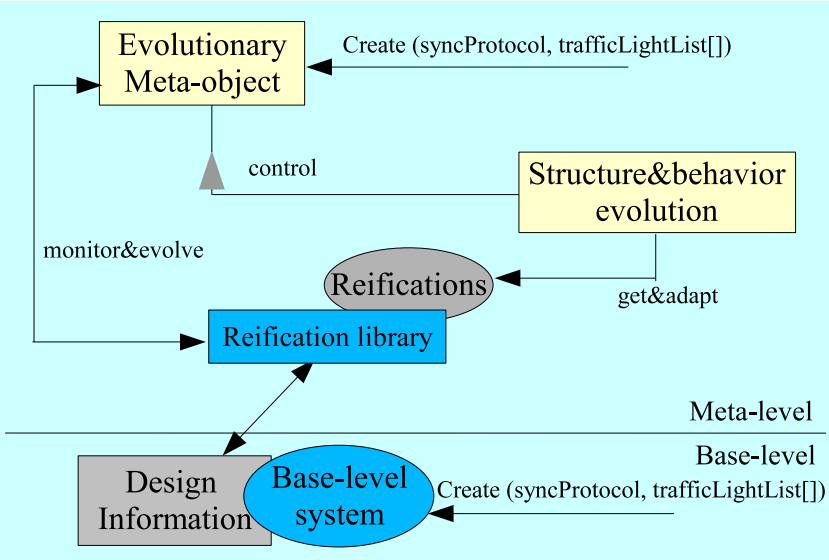



**Figure 5.4:** The application of the evolutionary meta-object in the UTCS

- the system need extra code and data structures representing the system, its behavior and the evolutionary rules.

A critical role is played by the adaptation rules, they are the core of the evolutionary and their realization is very hard because adaptations badly designed or applied at the wrong time could have very dire consequences, e.g., consider the chaos generated by stopping the traffic lights in a very busy area during the rush hour.

### 5.3.2. Consistency checker meta-object

To verify the feasibility and the soundness of the changes applied by evolutionary meta-object. That is, to check if it is possible to apply such changes without rendering inconsistent the base-level system. The consistency checker meta-object role is formally described as follows:

**Definition 5.3.2.** The Consistency checker meta-object has the following roles: (1) check the consistency of the modified design information ($MDI$) based on the applied evolution strategy ($ES$); (2) using the $RRL_{\mathcal{R}\text{AMSES}.reflection}(MDI.\text{XMI})$ for manipulating the modified design information; (3) Create a $CCRList$ based-on the created postcondition by $EMObj$.

$$\begin{aligned} \mathcal{R}\text{AMSES}_{mc_{CCMObj}} &= \{CCMObj.checkConsistency(MDI,ES), \\ &\quad CCMObj_{RRL_{\mathcal{R}\text{AMSES}.reification}}(MDI.\text{XMI}), \\ &\quad CCMObj.create(CCRList)\} \\ s.t.\ \forall\ ES\ \exists & \\ CCS_{ES} &= \{CCMObj.get(MDI,ES),\ CCMObj_{consEngine}(CCRList)\}. \end{aligned}$$

The role of consistency checker meta-object is realized as follows:

**IF** $(CCMObj.applied(CCRList) = true)$
**THEN** $BDI.\mathsf{XMI} = RRL_{\mathcal{RAMSES}.reflection}(MDI.\mathsf{XMI})$
**Else** $CCMObj.inconsistencyDetected(), EMObj.reCreate(ES').$

***Where.*** *mc* is the meta-level component, *CCMObj* is the consistency checker meta-object, *MDI* is the evolved design information based on the evolution strategy (ES), *RRL* is the RAMSES reification library, *CCS* is the consistency checker strategy, *CCRList* is the list of the validation rules, ES′ is a new evolution strategy to avoid inconsistency state. ◊

The delicacy of dynamically changing (part of) a component of a system is fairly evident. Usually changes directly affect only (part of) a component rendering simple to verify the effects of the changes. In complex systems each component cooperates with, integrates/is integrated in, uses/is used by other components, therefore, the effect of changes performed on a component are propagated to many other components not directly involved by the modification. Hazardous changes to a component will conflict with the overall behavior of the system and such conflicts are quite difficult to be detected. This problem is further amplified by the fact that the system can not be stopped hindering an easy reconfiguration and validation of the complete system. For example, in the UTCS, at a crossroads we can not turn green a traffic light without considering the state of the correspondent traffic light for pedestrians. Therefore, it is important to verify that environmental changes impacting on many components do not generate conflicts in the overall system and the effect of these environmental changes has only to be propagated to the system components when it is safe, i.e., when the propagation do not leave the system in an inconsistent state.

It is fairly evident from the problem description that every "proposed" change to a component of the system has to be well planned and validated against inconsistency. Hence we need a mechanism that applies the changes to the system only when the "proposed" changes are proved to leave consistent the system. Moreover, such a mechanism has not only to guarantee against inconsistencies due to erroneous updates but also to choose the right moment for applying the "proposed" changes.

The basic idea consists of gathering many "proposed" changes together on representatives of the base-level system, checking step by step that replacing such a pool of representatives with the corresponding aspects of the system will leave the system in a consistent situation. Then, the replacement will take effectively place when the system is in a state that can safely be carried out. Whereas, changes, that the consistency checker considers that could render the system inconsistent, are returned to the evolutionary for fixing.

The pseudo code that illustrates the basic steps carried out by the consistency checker for verifying and maintaining consistent the base-level system after evolution is shown in the algorithm 5.

**Data** : evolved meta-data - Reification, evoPlan - `evolutionaryMetaObject`
**Result**: consistencyBool - `consistencyCheckerMetaObject`

```
/*sample of consistency checker working only on modified meta-data */
strView ← ramses.reification(meta-data);
behView ← ramses.reification(meta-data);
plans ← evolutionaryMetaObject(plans);
begin
    /*reset the notification of a change received by evolutionary */
    plans ← CreateConsistencyPlan (evoPlan, meta-data);
    begin
        bool←check_consistency() (); strView.rest();
        behView.rest();
        plan.check_consistency() (strView,behView);
    end
    conChecker ← consistencyCheckerMetaObject(meta-data);
    while true do
        if strView.is_changed() then
            if conChecker.check_consistency() then
                _plan ← consistencyCheckerMetaObject
                (consistencyCheckerEngine, plans);
                structureAdaptation.inconsistency_detected() ;
                behaviorAdaptation.inconsistency_detected() ;
            else
                strView.adaptation ;
                behView.adaptation ;
            end
        end
    end
end
```

**Algorithm 5:** The role of the consistency checker

The consistency checker works on the whole system. It does not work only on a specific aspect but rather it has to maintain the consistency among meta-data of the system. Consistency rules are an important element managed by the consistency checker. These rules are represented by an instance `_plan` of the `plans<consistency>` class and are used to determine if the representatives (in our pseudo code are represented by a be-

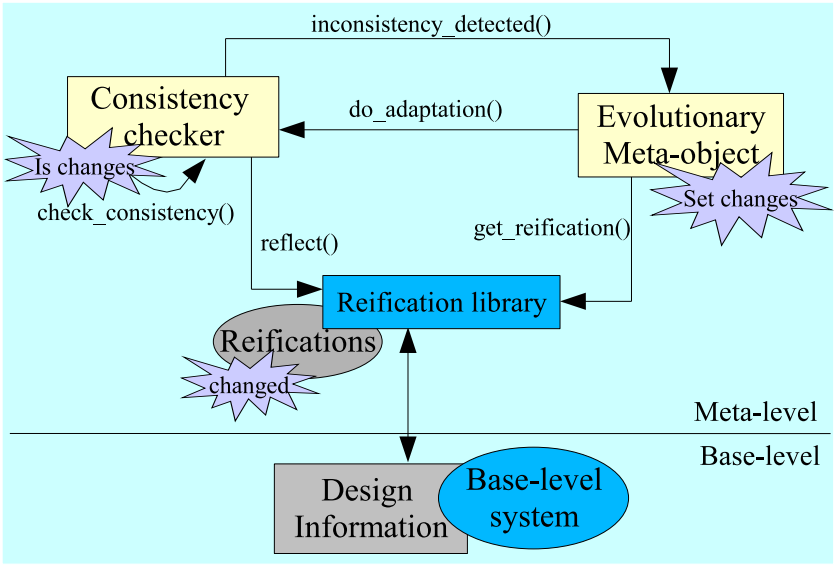



**Figure 5.5:** The role of consistency checking against evolution of the system structure.

havior: `behView` and a structure: `strView`) are a consistent snapshot of the system. The method `check_consistency()` of the consistency checker delegates `_plan` for such a check on the representatives.
Both the evolutionary and the consistency checker work on system representatives. Evolutionary objects carry out their work when external events occur whereas the consistency checker performs its work when one of the representatives that it is monitoring is modified, that is, when an evolutionary object proposes a change. If evolutionary objects notify that they have carried out a change to the consistency checker, it is able to simply detect such a change in the representatives. Such a notification is performed through a boolean flag added to the shared representatives. Such a flag is set to true when an evolutionary object modifies the representative, to false when the consistency checker validates the proposed changes. Therefore, the consistency checking will take place when the consistency checker detects a change in the flags shared with the evolutionary objects.

In figure 5.5 we show the integration of the consistency checker with the evolutionary. The figure sketches consistency checker role in the RAMSES architecture by considering the role of the reification library.

The consistency checker has to be used when we have to check the consistency of dynamic changes carried out by a system on a representative of another system before effectively performing such changes. The consistency checker has to be used in critical environments to avoid the dire consequences of erroneous and inconsistent updates.

A feasible use of the consistency checker consists of checking the consistency of the base-level of a reflective system against changes performed by the meta-level system before reflection takes place. However this meta-object is not mined from existing systems.

The consistency checker can be used stand alone for checking the consistency of a system or in collaboration with evolutionary for safely evolving a system:

- it compares the consistency of the meta-data embodied by the reification library. It uses a set of predefined rules;
- it interacts with evolutionary to fix potential inconsistencies between the "proposed" adaptation and the referents;
- it delegates/authorizes the reflection to update the corresponding aspects after the validation of the "proposed" adaptation.

The consistency checker provides the following advantages:

- It checks the consistency of the meta-data after evolution and before updating the base-level system in accordance with the adaptation. That is, it checks that carrying out the adaptation proposed by evolutionary will not render the base-level system inconsistent.
- The control flow returns to the evolutionary for fixing the proposed evolution if the consistency check fails.
- It looks, in collaboration with the evolutionary, for the right moment to allow system evolution, i.e., the moment which guarantees that evolution leaves the evolved system working and consistent.

The consistency checker has a few drawbacks:

- It augments the run-time overhead due to its checking and to its cooperation with evolutionary meta-objects for fixing the inconsistencies.
- Its work is based on a rigorous set of rules establishing when the system can be considered inconsistent.
- Adaptation does not immediately occur.

An important point is represented by the quality of the rules composing the validation system. This requirements is a very delicate point which requires a highly skilled software architect because all the efficacy of the consistency checker is based on the quality of the validation system and a bad designed validation system can have dire consequences.

## 5.4. Summary

In this chapter we have proposed an infrastructure named RAMSES to dynamically adapt software systems using architecture reflection. As with common reflective systems, we have divided the architecture of software systems into base- and meta-level. The base-level consists of the running application as well as design information in form of XMI schemas. The meta-level is composed of an interpreter engine for managing the evolution

and validating consistency processes for runtime changes. The evolution and validation is based on graph transformation which take place on the reified design information (XMI schemas). The meta-level behavior is described by the role of two evolutionary meta-objects (*evolutionary* and *consistency checker*).

# 6 The UTCS: a Case Study

This chapter describes how design information, i.e., UML specifications, can be used to evolve a software system and validate the consistency of such an evolution. In the previous chapters we presented the RAMSES middleware for software evolution describing the role played by meta-data in the evolution of software systems. The whole chapter focuses on a case study; we show how the *urban traffic control system* (UTCS) or part of it must evolve when unscheduled road maintenance, a car crush or a traffic jam block normal vehicular flow in a specific road. The UTCS case study perfectly shows how requirements can dynamically change and how the design of the system should adapt to such changes. Both system consistency and adaptation are governed by rules based on meta-data representing the system design information. As we show by an example, such rules represent the core of our evolutionary approach driving the *evolutionary* and *consistency checker meta-objects* and interfacing the meta-level system (the evolutionary system) with the system that has to be adapted.

This chapter is organized as follows: Section 6.1, provides a brief overview of the dynamic application UTCS. Section 6.2, we described three different cases of three different unanticipated events. Section 6.3, presents the design information for case (A). Section 6.4, presents the UTCS evolution prototype. Section 6.5, briefly summarizes main points highlighted in this chapter.

## 6.1. The Specification and Components of the UTCS

We describe the specification for one of the systems that have a continuously changing nature, the system we deal with is the *urban traffic control systems* (UTCS). The software engineers and city planners have to face many criteria for specifying the design issues of these systems. Moreover, part of these criteria should be able to deal with unanticipated events. When designing (UTCS) of a modern city, the software engineer cannot face all the unanticipated events. These systems should be flexible for new functional and non-functional requirements. The UTCS have to deal with a lot of unexpected and hard to plan such as traffic lights disruptions, roads maintenance, car crashes, traffic jam and so on. These considerations necessitate the need of rendering the UTCS dynamically adaptable to the external events.

We use the UTCS as a case study for many reasons, as follows:

- The environment of UTCS is dynamic;
- This case study is rich with unanticipated events. At design-time, we can not imagine and implement all these events;
- This system is flexible to include new functional and non-functional requirements;
- The area of UTCS is rich with many fix components and mobile components, the relation among these components can dynamically change to get a new behavior for the system;

The existing UTCS systems lack in adaptability to evolve and validate the traffic flow of some areas for unanticipated events such as: adding new traffic lights, changing the roads directions, changing the synchronization protocol between the opposite traffic lights, re-engineering the traffic flow of some area and so on.

### 6.1.1. UTCS components

This section illustrates the main components of the urban traffic control systems (UTCS). The UTCS consists of the following components, that provide a simplified vision of their software systems. Moreover, these components abstract the real world:

- *Traffic Nodes*: represent the road intersections. Every traffic node has a set of road sections, and the flow through its connected roads is managed by using a set of synchronized traffic lights. We consider the traffic nodes class has the following methods: (1) give the ability to create a new traffic node; (2) specify the connected road sections to this traffic node; (3) detect the direction of these roads; (4) provide how to change the synchronization protocol for specific traffic node and changes the flow trough the connected road sections. This component is the main in *UTCS* and has a relation to the other components.
- *Traffic Lights:* are a signalling device positioned at a road intersection or pedestrian crossing to indicate when it is safe to drive, ride or walk, using colored lights, typically red for stop, green for go, and yellow for proceed with caution. The system planner should specify the synchronized traffic lights, and the synchronization protocol between them. The traffic light class has a set of methods such as: (1) create new traffic light; (2) get the current state of the traffic light; (3) change from state to another; (4) get and specifies the group of synchronization that the traffic light belongs to; (5) turn-on the initial states of the traffic light; (6) turn-off specific traffic lights.
- *Road Sections:* every city consists of a network of roads, that connects any place to the other. *UTCS* represents the road network as a graph (roads represent the arcs and roads intersection (crossroads) represent the nodes). The system planner should specify for each road the name, direction of flow, the identification code, the position of road in the map, and detect the connection with the other road

sections. Moreover, the maximum speed through each road. This class has the following operations: (1) create a new road section; (2) specify the directions for road sections and lanes; (3) specify the link of the road with the others; (4) associate connection with the traffic light class.

- *Synchronization Manager:* specifies the synchronization protocol for the traffic lights enclosed to a traffic node for controlling the flow at this traffic node. The synchronization manager has the following operations: (1) create a synchronization group for the set of traffic lights; (2) switch from one group of synchronization to another.

- *Synchronization Protocol:* the synchronization protocol specifies different phases of synchronization at each traffic node. Each phase specifies the allowed direction and the color required to allow this direction. The synchronization protocol has the following operations: (1) define the traffic flow phases for each group of synchronization; (2) define the initial states for each traffic light in the synchronization group; (3) generalize the synchronization type which create the suitable type of synchronization for each group.

- *Traffic Link:* the main propose is to specify the link between road sections and their traffic lights. The main operations for this class are: (1) create the relation between the traffic lights and the roads; (2) specify the start and end for each road section and the traffic lights.

- *Directions:* specify the directions for each road sections connected to the crossroads (traffic node). The main operations for this class are: (1) create a new direction; (2) change the existing one; (3) specify the relations between road sections and traffic nodes.

### 6.1.2. Specification of the UTCS

The map in the figure 6.1, represents a simplification of part of the Berlin downtown. We marked on the map some special area that we use to explain how RAMSES work. Moreover, for each case we sketch the original map for that area and the modified map, more details about these events are in the next sections.

We discuss the specification of the *UTCS* in general. In this system we have a map that describes the flow in normal case as follows:

- There are a set of roads with different types (one way, two ways, and many lanes);

- There is a circular axes that links four roads;

- The traffic lights at a crossroad must be synchronized. There are many cases of synchronization: two traffic light, three traffic light and four traffic light.

In such systems, the urban manager supervises normal operations, that describe anticipated events like: controls the roads, and controlled the traffic lights. The system planner address a set of rules, that should be considered to validate these respect of a given set of constraints. Some of the issues considered in the case study are:

- Reachability: means cars must be able to reach every road from every where;
- Every opposite traffic lights at the crossroad must be synchronized.
- For the cycling connection there is a priority for the cars that pass in cycle way (priority conditions).
- Tram circulations cannot be interrupted, it is close path.

In the following section we describe in abstract-level three different cases of UTCS system, then, we discuss one of these cases in more detailed.

## 6.2. UTCS Cases

This section describes the three different cases marked in the Berlin map as shown in figure 6.1.

### 6.2.1. Case (A): closing a lane or part of lane

We have analyzed the map of area shown in figure 6.2, that is simplification of the map of the area (A) marked in figure 6.1. The normal layout of area (A) consists of two traffic nodes ($\mathrm{tn}_1$ and $\mathrm{tn}_2$), each traffic node represents the crossroads (roads intersections). The traffic flow at each traffic node is controlled by using a set of traffic lights. In the normal layout, traffic node ($\mathrm{tn}_1$) is controlled by four traffic lights, and traffic node ($\mathrm{tn}_2$) is controlled by three traffic lights. Each set of traffic lights has the same type of synchronization protocol named *TwoGroupsSync*. The formal representation of the normal layout of case (A) is as follows: Given the following nodes:

$\mathrm{TNset}_{CaseA} = \{\mathrm{tn}_1, \mathrm{tn}_2\}$, s.t.

$$\begin{aligned} \mathrm{TL}_{\mathrm{tn}_1} &= \{\mathrm{tf}_1, \mathrm{tf}_2, \mathrm{tf}_3, \mathrm{tf}_4\}, \\ \mathrm{TL}_{\mathrm{tn}_2} &= \{\mathrm{tf}_5, \mathrm{tf}_6, \mathrm{tf}_7\}, \\ \mathrm{RS}_{\mathrm{tn}_1} &= \{\mathrm{rs}_1, \mathrm{rs}_2, \mathrm{rs}_3, \mathrm{rs}_4\}, \\ \mathrm{RS}_{\mathrm{tn}_2} &= \{\mathrm{rs}_5, \mathrm{rs}_4, \mathrm{rs}_6\}. \end{aligned}$$

The *syncManager* specifies the following groups of synchronization for case (A):

$$\mathrm{TwoGroupsSync}((\mathrm{A},\mathrm{B}), [(\mathrm{tf}_1, \mathrm{tf}_3), (\mathrm{tf}_2, \mathrm{tf}_4)]),$$

$$\mathrm{TwoGroupsSync}((\mathrm{A},\mathrm{B}), [(\mathrm{tf}_6), (\mathrm{tf}_5, \mathrm{tf}_7)]).$$

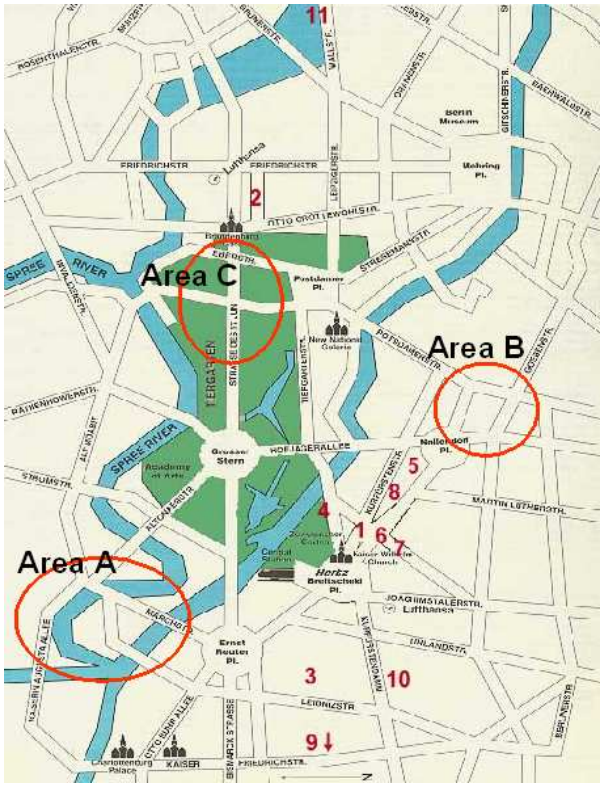



**Figure 6.1:** The original map for Berlin city

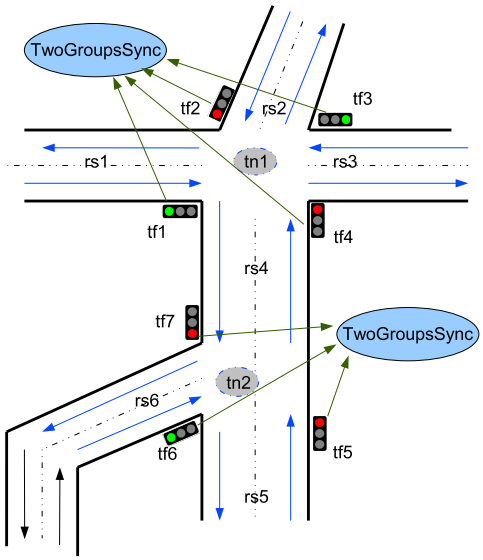



**Figure 6.2:** Layout of Case A.

#### Traffic jam at a lane of road section (rs4)

In figure 6.3, we illustrate two plans from the designer point of view to evolve the layout according to an unanticipate event (closing all or part of the left lanes of road section $\mathrm{rs}_4$). The formal representation of the first plan of modified layout of case (A) is described as follows: Given the following nodes:

$\mathrm{TNset}_{M1caseA} = \{\mathrm{tn}_1, \mathrm{tn}_2\}$, s.t.

$$\begin{aligned} \mathrm{TL}_{\mathrm{tn}_1} &= \{\mathrm{tf}_1, \mathrm{tf}_2, \mathrm{tf}_3\}, \\ \mathrm{TL}_{\mathrm{tn}_2} &= \{\mathrm{tf}_5, \mathrm{tf}_6\}, \\ \mathrm{RS}_{\mathrm{tn}_1} &= \{\mathrm{rs}_1, \mathrm{rs}_2, \mathrm{rs}_3, \mathrm{rs}_4\}, \\ \mathrm{RS}_{\mathrm{tn}_2} &= \{\mathrm{rs}_5, \mathrm{rs}_4, \mathrm{rs}_6\}. \end{aligned}$$

The *syncManager* specifies the following group of synchronization for case (*M1case1*):

$$\mathrm{TwoGroupsSync}((\mathrm{A,B,C}), [(\mathrm{tf}_1, \mathrm{tf}_3), (\mathrm{tf}_2), ,(\mathrm{tf}_5, \mathrm{tf}_6)]).$$

The formal representation of the second plan of modified layout of case (A) is as follows: Given the following nodes: $\mathrm{TNset}_{M2CaseA} = \{\mathrm{tn}_1, \mathrm{tn}_2\}$, s.t.

$$\begin{aligned} \mathrm{TL}_{\mathrm{tn}_1} &= \{\mathrm{tf}_1, \mathrm{tf}_2, \mathrm{tf}_3, \mathrm{tf}_4\}, \\ \mathrm{TL}_{\mathrm{tn}_2} &= \{\mathrm{tf}_5, \mathrm{tf}_6, \mathrm{tf}_7\}, \\ \mathrm{TL}_{\mathrm{tn}_3} &= \{\mathrm{tf}_8, \mathrm{tf}_9\}, \\ \mathrm{RS}_{\mathrm{tn}_1} &= \{\mathrm{rs}_1, \mathrm{rs}_2, \mathrm{rs}_3, \mathrm{rs}_7\}, \\ \mathrm{RS}_{\mathrm{tn}_2} &= \{\mathrm{rs}_8, \mathrm{rs}_5, \mathrm{rs}_6\}, \\ \mathrm{RS}_{\mathrm{tn}_3} &= \{\mathrm{rs}_7, \mathrm{rs}_8\}. \end{aligned}$$

The *syncManager* specifies the following groups of synchronization for case (*M2caseA*):

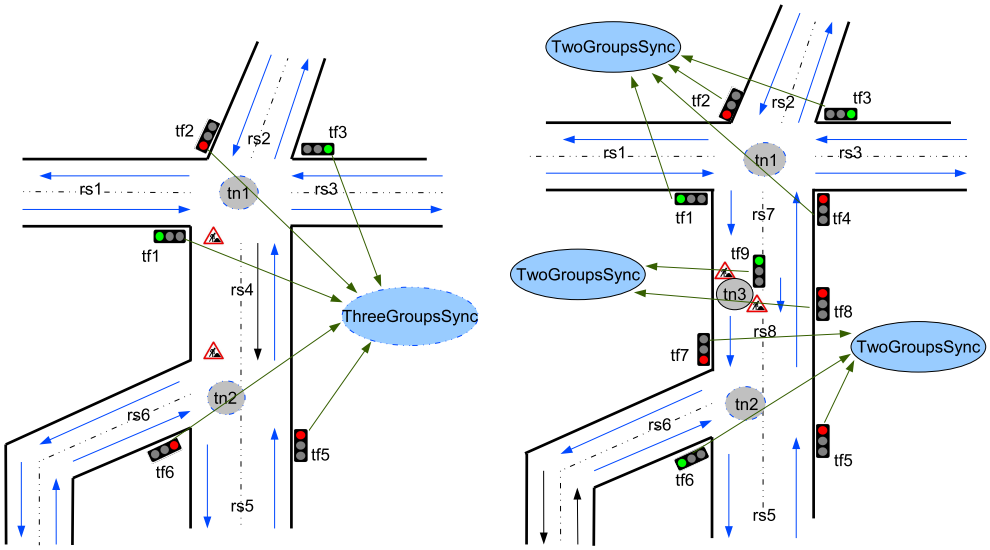
$$\mathrm{TwoGroupsSync}((\mathrm{A,B}), [(\mathrm{tf}_1, \mathrm{tf}_3), (\mathrm{tf}_2, \mathrm{tf}_4)]),$$

$$\mathrm{TwoGroupsSync}((\mathrm{A,B}), [(\mathrm{tf}_6), (\mathrm{tf}_5, \mathrm{tf}_7)]),$$

$$\mathrm{TwoGroupsSync}((\mathrm{A,B}), [(\mathrm{tf}_8), (\mathrm{tf}_9)]).$$

### 6.2.2. Normal layout for case (B): an overview

In this section, we discuss the layout of area (B), in the original map. The normal layout of this case is described in figure 6.4. The normal layout consists of four traffic nodes. The traffic flow at the traffic node ($tn_1$ and $tn_2$) at the main road is controlled by four traffic lights. The traffic flow for the other traffic nodes depends on the priority of flow. The traffic lights at traffic node ($tn_1$ and $tn_2$) have the synchronization protocol named *TwoGroupsSync*. The formal representation of the normal layout of case (B) as follows: Given the following nodes:




(a) First Plan (b) Second Plan

**Figure 6.3:** Proposed evolution of Area A: a) adapted layout according first plan b) adapted layout according second plan.

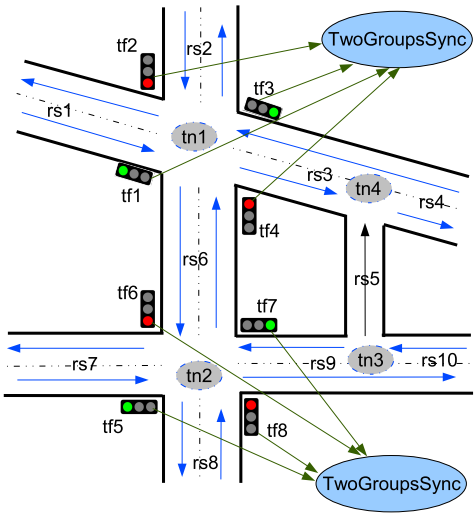



**Figure 6.4:** Layout of Case B.

$\text{TNset}_{CaseB} = \{\text{tn}_1, \text{tn}_2, \text{tn}_3, \text{tn}_4\}$, s.t.

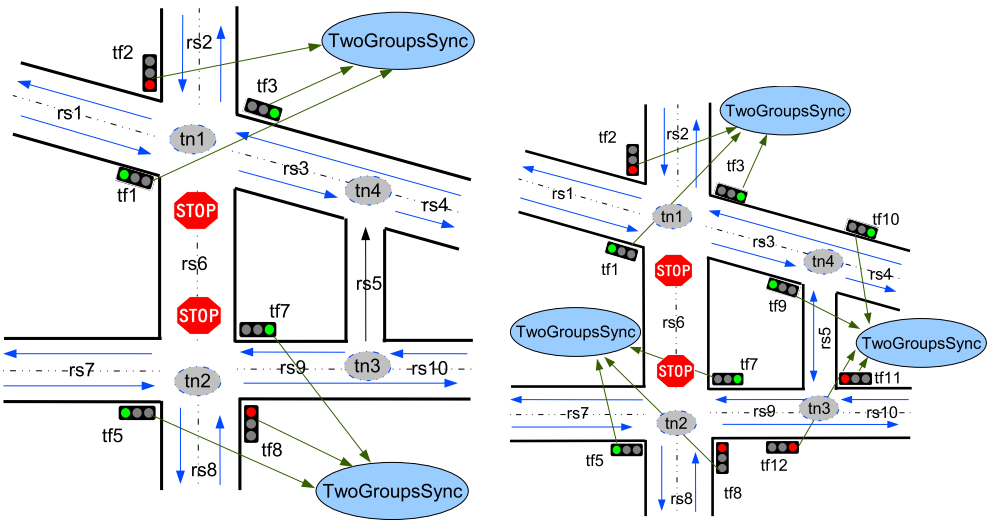
$$\begin{aligned}
\text{TL}_{\text{tn}_1} &= \{\text{tf}_1, \text{tf}_2, \text{tf}_3, \text{tf}_4\},\\
\text{TL}_{\text{tn}_2} &= \{\text{tf}_5, \text{tf}_6, \text{tf}_7, \text{tf}_8\},\\
\text{RS}_{\text{tn}_1} &= \{\text{rs}_1, \text{rs}_2, \text{rs}_3, \text{rs}_6\},\\
\text{RS}_{\text{tn}_2} &= \{\text{rs}_7, \text{rs}_6, \text{rs}_9, \text{rs}_8\},\\
\text{RS}_{\text{tn}_3} &= \{\text{rs}_5, \text{rs}_9, \text{rs}_{10}\},\\
\text{RS}_{\text{tn}_4} &= \{\text{rs}_3, \text{rs}_4, \text{rs}_5\}.
\end{aligned}$$

The *syncManager* specifies the following groups of synchronization for case (*CaseB*):

$$\text{TwoGroupsSync((A,B), [(tf}_1\text{, tf}_3\text{), (tf}_2\text{, tf}_4\text{)])},$$
$$\text{TwoGroupsSync((A,B), [(tf}_5\text{, tf}_7\text{), (tf}_6\text{, tf}_8\text{)])}.$$

### closing a road for maintenance or accident

In the figure 6.4, we illustrate the normal behavior of the flow in the (B) area, when the main road (rs6) is closed. In that case, we need to modify the flow to that road and change the behavior of the traffic lights, that control the traffic flow from and to this road.




(a) First Plan (b) Second Plan

**Figure 6.5:** Proposed evolution of area B: a) adapted layout according first plan b) adapted layout according second plan.

In figure 6.5, we illustrate two plans from the designer point of view to evolve the layout according to an unanticipate event (closing road rs6). The formal representation of

the first plan of modified layout of case (B) is as follows: Given the following nodes: $\text{TNset}_{M1caseB} = \{\text{tn}_1, \text{tn}_2, \text{tn}_3, \text{tn}_4\}$, s.t.

$$\begin{aligned}
\text{TL}_{\text{tn}_1} &= \{\text{tf}_1, \text{tf}_2, \text{tf}_3\},\\
\text{TL}_{\text{tn}_2} &= \{\text{tf}_5, \text{tf}_7, \text{tf}_8\},\\
\text{RS}_{\text{tn}_1} &= \{\text{rs}_1, \text{rs}_2, \text{rs}_3\},\\
\text{RS}_{\text{tn}_2} &= \{\text{rs}_7, \text{rs}_9, \text{rs}_8\},\\
\text{RS}_{\text{tn}_3} &= \{\text{rs}_5, \text{rs}_9, \text{rs}_{10}\},\\
\text{RS}_{\text{tn}_4} &= \{\text{rs}_3, \text{rs}_4, \text{rs}_5\}.
\end{aligned}$$

The *syncManager* specifies the following groups of synchronization for case (*M1caseB*):

$$\text{TwoGroupsSync((A,B), [(tf}_1\text{, tf}_3\text{), (tf}_2\text{)]),}$$

$$\text{TwoGroupsSync((A,B), [(tf}_5\text{, tf}_7\text{), (tf}_8\text{)]).}$$

The formal representation of the second plan of modified layout of case (B) is as follows: Given the following nodes: $\text{TNset}_{M2caseB} = \{\text{tn}_1, \text{tn}_2, \text{tn}_3, \text{tn}_4\}$, s.t.

$$\begin{aligned}
\text{TL}_{\text{tn}_1} &= \{\text{tf}_1, \text{tf}_2, \text{tf}_3\},\\
\text{TL}_{\text{tn}_2} &= \{\text{tf}_5, \text{tf}_7, \text{tf}_8\},\\
\text{TL}_{\text{tn}_3} &= \{\text{tf}_9, \text{tf}_{10}, \text{tf}_{11}, \text{tf}_{12}\},\\
\text{RS}_{\text{tn}_1} &= \{\text{rs}_1, \text{rs}_2, \text{rs}_3\},\\
\text{RS}_{\text{tn}_2} &= \{\text{rs}_7, \text{rs}_9, \text{rs}_8\},\\
\text{RS}_{\text{tn}_3} &= \{\text{rs}_5, \text{rs}_9, \text{rs}_{10}\},\\
\text{RS}_{\text{tn}_4} &= \{\text{rs}_3, \text{rs}_4, \text{rs}_5\}.
\end{aligned}$$

The *syncManager* specifies the following groups of synchronization for case (*M2caseB*):

$$\text{TwoGroupsSync((A,B), [(tf}_1\text{, tf}_3\text{), (tf}_2\text{)]),}$$

$$\text{TwoGroupsSync((A,B), [(tf}_5\text{, tf}_7\text{), (tf}_8\text{)]),}$$

$$\text{TwoGroupsSync((A,B), [(tf}_9\text{, tf}_{10}\text{), (tf}_{11}\text{, tf}_{12}\text{)]).}$$

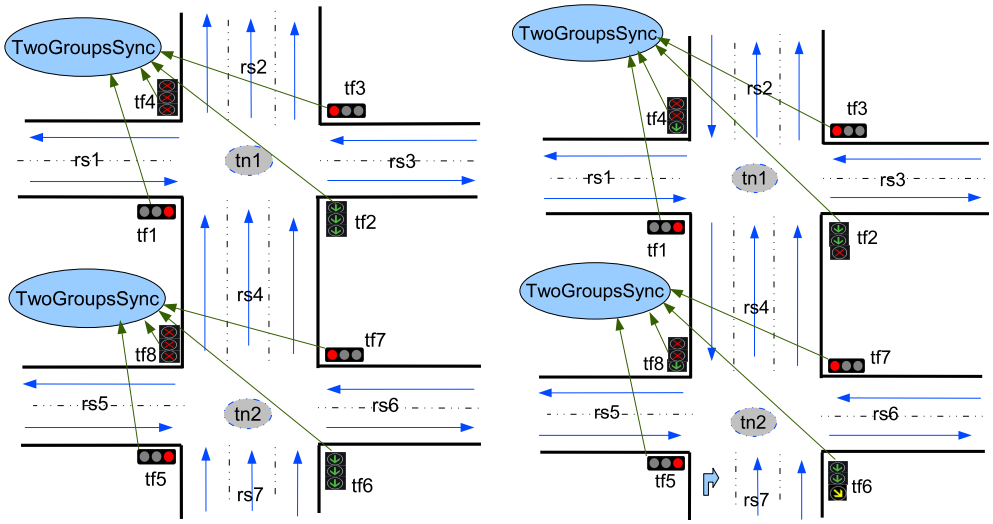
### 6.2.3. Normal layout for case (C): an overview

The layout of the considered area (C) consists of two traffic nodes ($\text{tn}_1$ and $\text{tn}_2$); each traffic node represents a crossroads as shown in figure 6.6.a. The traffic flow at each traffic node is controlled by a set of traffic lights. In details, the traffic at the traffic nodes $\text{tn}_1$ and $\text{tn}_2$ are respectively controlled by four traffic lights. Both sets of traffic lights adopt the same synchronization protocol (named *TwoGroupsSync*): opposite traffic lights have always the same color, if a couple is red the other one is green or vice versa. The formal representation of the normal layout of case (C) is:

$\mathrm{TN}_{CaseC} = \{\mathrm{tn}_1, \mathrm{tn}_2\}$, s.t.

$$\begin{aligned}
\mathrm{TL}_{\mathrm{tn}_1} &= \{\mathrm{tf}_1, \mathrm{tf}_2, \mathrm{tf}_3, \mathrm{tf}_4\} \\
\mathrm{TL}_{\mathrm{tn}_2} &= \{\mathrm{tf}_5, \mathrm{tf}_6, \mathrm{tf}_7, \mathrm{tf}_8\} \\
\mathrm{RS}_{\mathrm{tn}_1} &= \{\mathrm{rs}_1, \mathrm{rs}_2, \mathrm{rs}_3, \mathrm{rs}_4\} \\
\mathrm{RS}_{\mathrm{tn}_2} &= \{\mathrm{rs}_5, \mathrm{rs}_4, \mathrm{rs}_6, \mathrm{rs}_7\}
\end{aligned}$$

The *syncManager* specifies the following groups of synchronization for case (*CaseC*):




(a) Layout of Case (C) (b) Emergency plan

**Figure 6.6:** Case (C): (a) normal layout, (b) emergency plan.

TwoGroupsSync((A,B), [($\mathrm{tf}_1$, $\mathrm{tf}_3$), ($\mathrm{tf}_2$, $\mathrm{tf}_4$)])

TwoGroupsSync((A,B), [($\mathrm{tf}_5$, $\mathrm{tf}_7$), ($\mathrm{tf}_6$, $\mathrm{tf}_8$)])

Note that, in the considered area we have a large avenue (the road composed by the sections $\mathrm{rs}_2$, $\mathrm{rs}_4$ and $\mathrm{rs}_7$) with three lanes, the traffic lights steering the traffic flow in this avenue have three lights as well:

$$\begin{aligned}
\mathrm{tf}_2 &= \{\mathrm{tf}_{2_{\mathrm{L}_1}}, \mathrm{tf}_{2_{\mathrm{L}_2}}, \mathrm{tf}_{2_{\mathrm{L}_3}}\} \\
\mathrm{tf}_4 &= \{\texttt{off}, \texttt{off}, \texttt{off}\} \\
\mathrm{tf}_6 &= \{\mathrm{tf}_{6_{\mathrm{L}_1}}, \mathrm{tf}_{6_{\mathrm{L}_2}}, \mathrm{tf}_{6_{\mathrm{L}_3}}\} \\
\mathrm{tf}_8 &= \{\texttt{off}, \texttt{off}, \texttt{off}\}
\end{aligned}$$

When an anomalous situation is detected (e.g., a traffic jam in the rush hour or a gas tube explodes) the UTCS must adapt itself to solve or alleviate the emergency. Of

course, not all the anomalous situation can be foreseen at design-time and anyway the code and the design should not polluted with the management of these anomalous and seldom cases. Therefore the adaptation dynamically takes place and consequently also the design must be changed.

Consider the case of the emergency plan, showed in Fig. 6.6.b, for alleviating the congestion at the rush hour in the large avenue. In the plan the first lane of the avenue change will be run in the other direction and consequently some traffic lights change their behavior and synchronization protocol.

In particular the traffic lights in the large avenue are characterize by:

$$\begin{aligned} \mathrm{tf}_2 &= \{\mathrm{tf}_{2_{\mathrm{L}_1}}, \mathrm{tf}_{2_{\mathrm{L}_2}}, \texttt{off}\} \\ \mathrm{tf}_4 &= \{\mathrm{tf}_{4_{\mathrm{L}_1}}, \texttt{off}, \mathrm{tf}_{2_{\mathrm{L}_2}}\} \\ \mathrm{tf}_6 &= \{\mathrm{tf}_{6_{\mathrm{L}_1}}, \mathrm{tf}_{6_{\mathrm{L}_2}}, \texttt{on/off}\} \\ \mathrm{tf}_8 &= \{\mathrm{tf}_{8_{\mathrm{L}_1}}, \texttt{off}, \texttt{off}\} \end{aligned}$$

## 6.3. Design Information Realization for Case (A)

In this section, we illustrate the class diagram and deployment models for the normal layout for the *Case(A)*. In figure 6.7, we realize the structure of the case (A) in form of class diagram, that presents the classes and their relations. In figure 6.8, we realize the instance road sections and its traffic lights of the normal layout instances as described in figure 6.2. In figure 6.9, we represent the first deployment object model of the modified layout as described in figure 6.8.a.

In our point of view, the statechart model opens a window of the system behavior. Since, the statechart represents the behavior of the system, then we illustrate different kinds of synchronization protocol. Figure 6.10, represents how four traffic lights synchronize to represent the state of traffic flow at traffic node (tn1). The behavior for the normal layout is described at figure 6.7, that has two types of synchronization protocols, (1) first protocol describes the synchronization between four traffic lights; (2) second protocol describes the synchronization between three traffic lights. To realize a new behavior that describes the modified layout as shown in figure 6.9, we create set of script rules to delete two instances of traffic lights and create new synchronization protocol as described in figures 6.11.

### 6.3.1. Evolutionary rules

RAMSES middleware has two types of meta-objects:(evolutionary and consistency checker) meta-objects. In this case, the evolutionary meta-object detects the runtime events, whose effect is to close the left lane of the road section $rs_4$. Then, the evolutionary meta-object proposes one of the following two evolutionary plans:

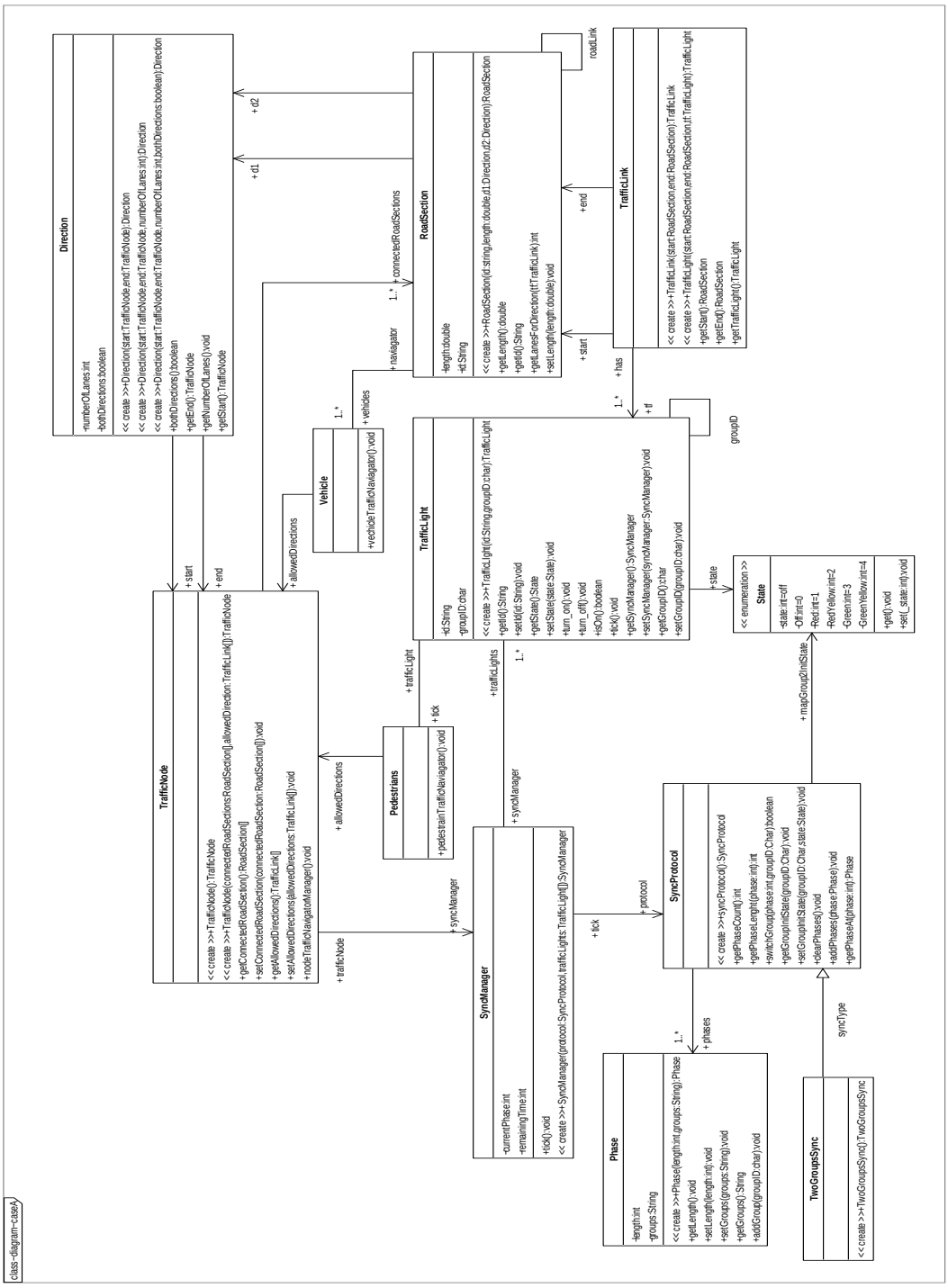
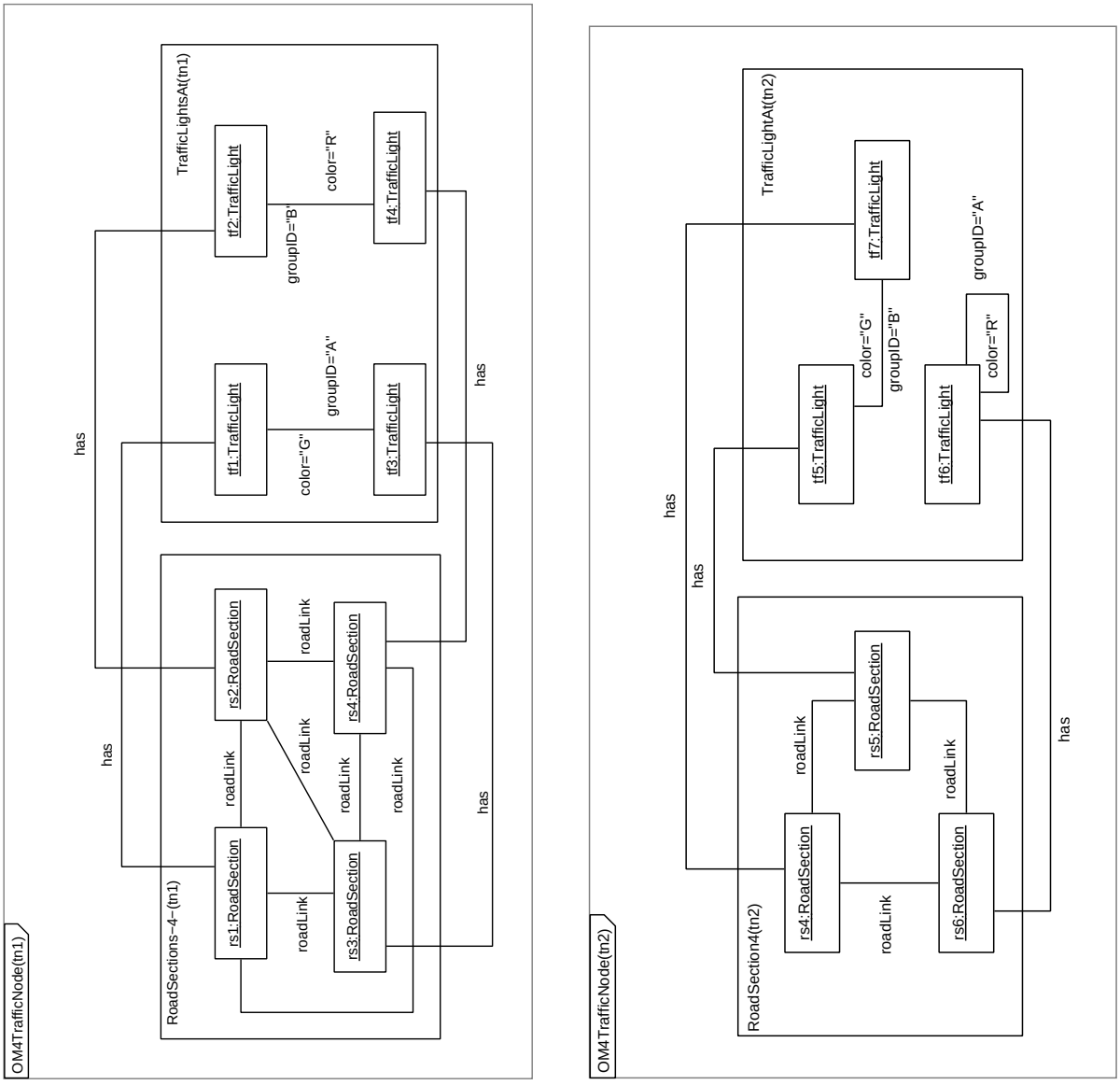



**Figure 6.7:** Class diagram for the normal layout.




(a) DM4tn1 (b) DM4tn2

**Figure 6.8:** Deployment diagram: a) object instances connection at traffic node (tn1) b) object instances connection at traffic node (tn2).

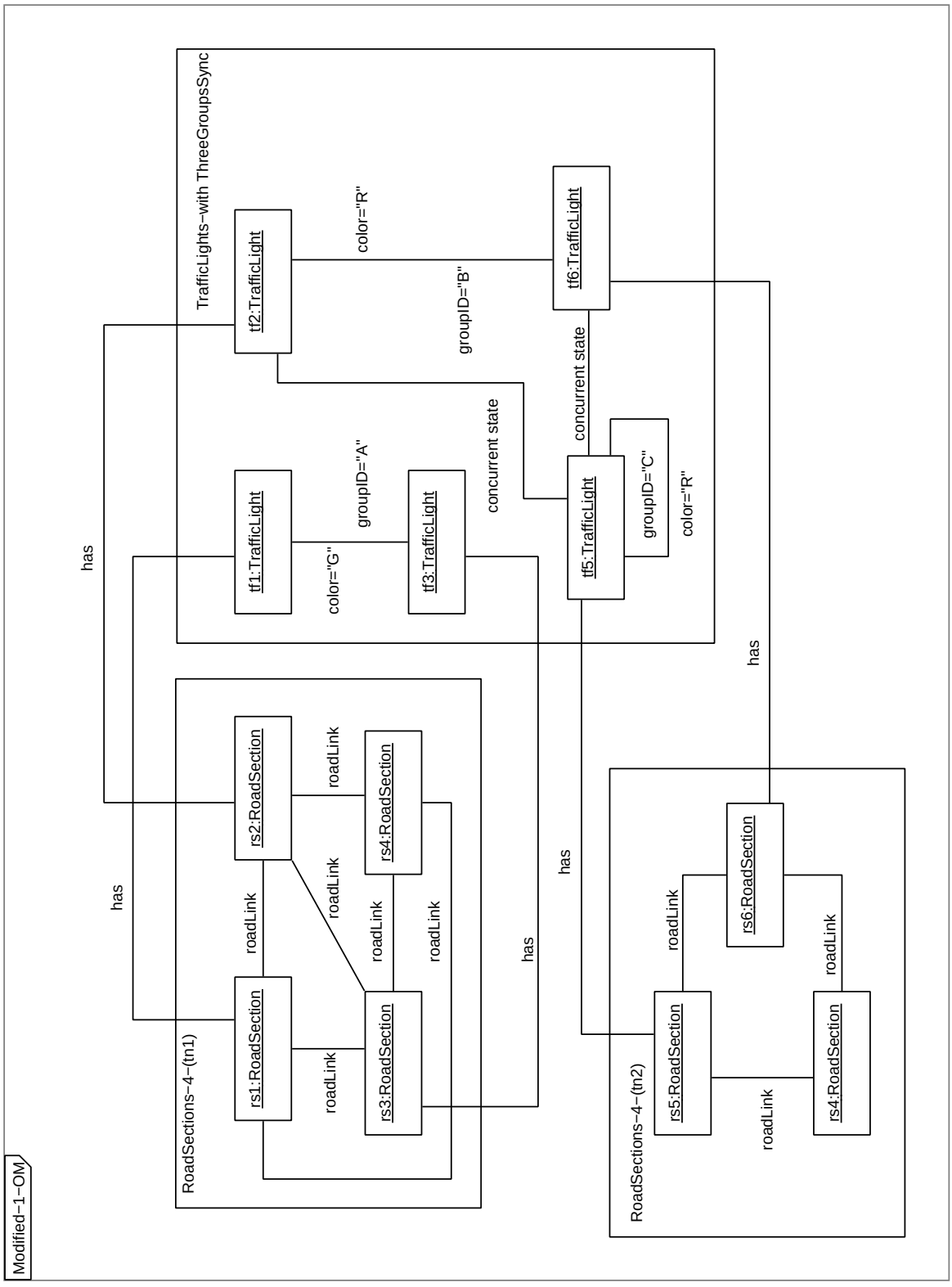



**Figure 6.9:** Adapted deployment diagram realized from first plan at figure 6.3.a

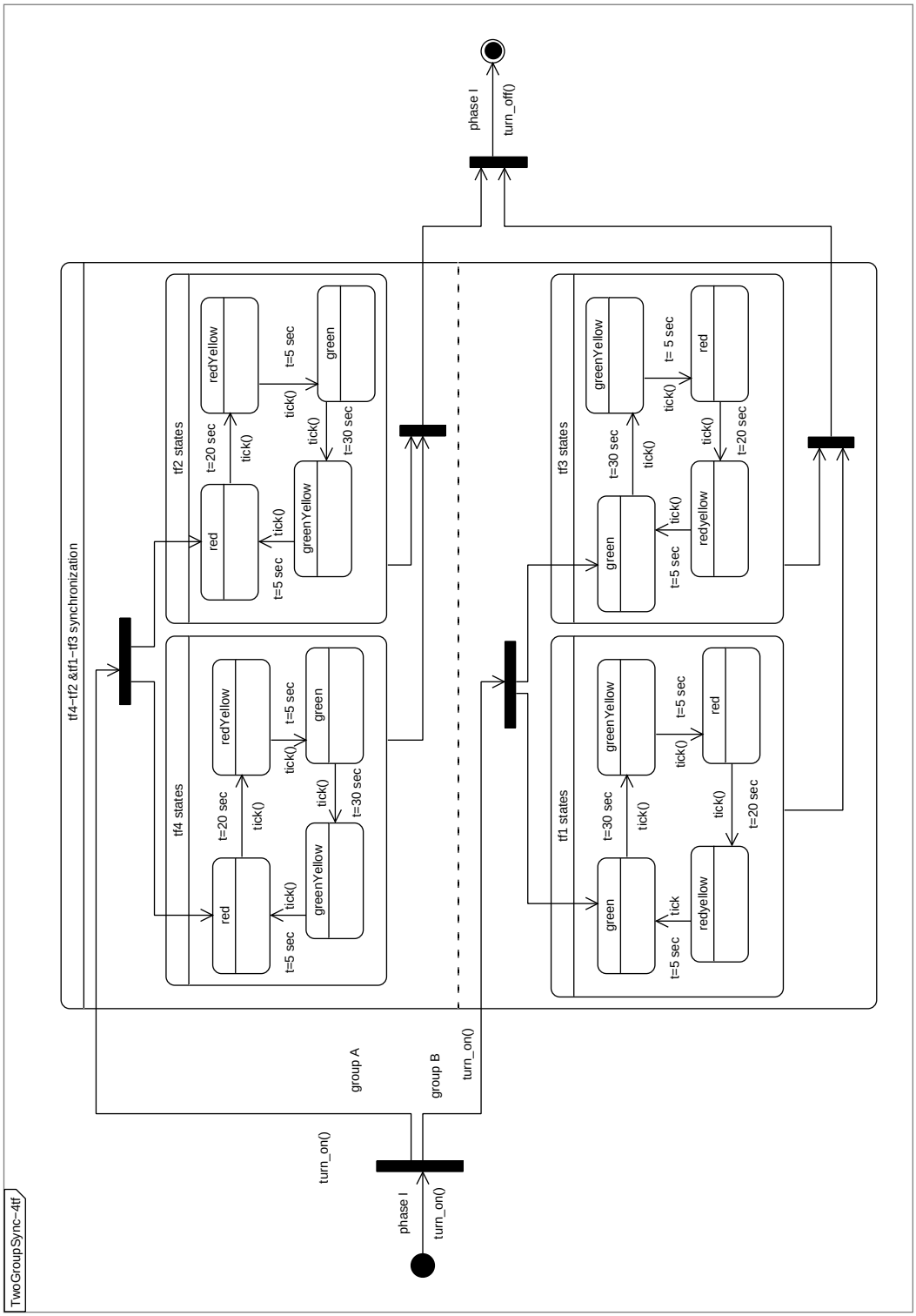



**Figure 6.10:** TwoGroupsSync with 4tf

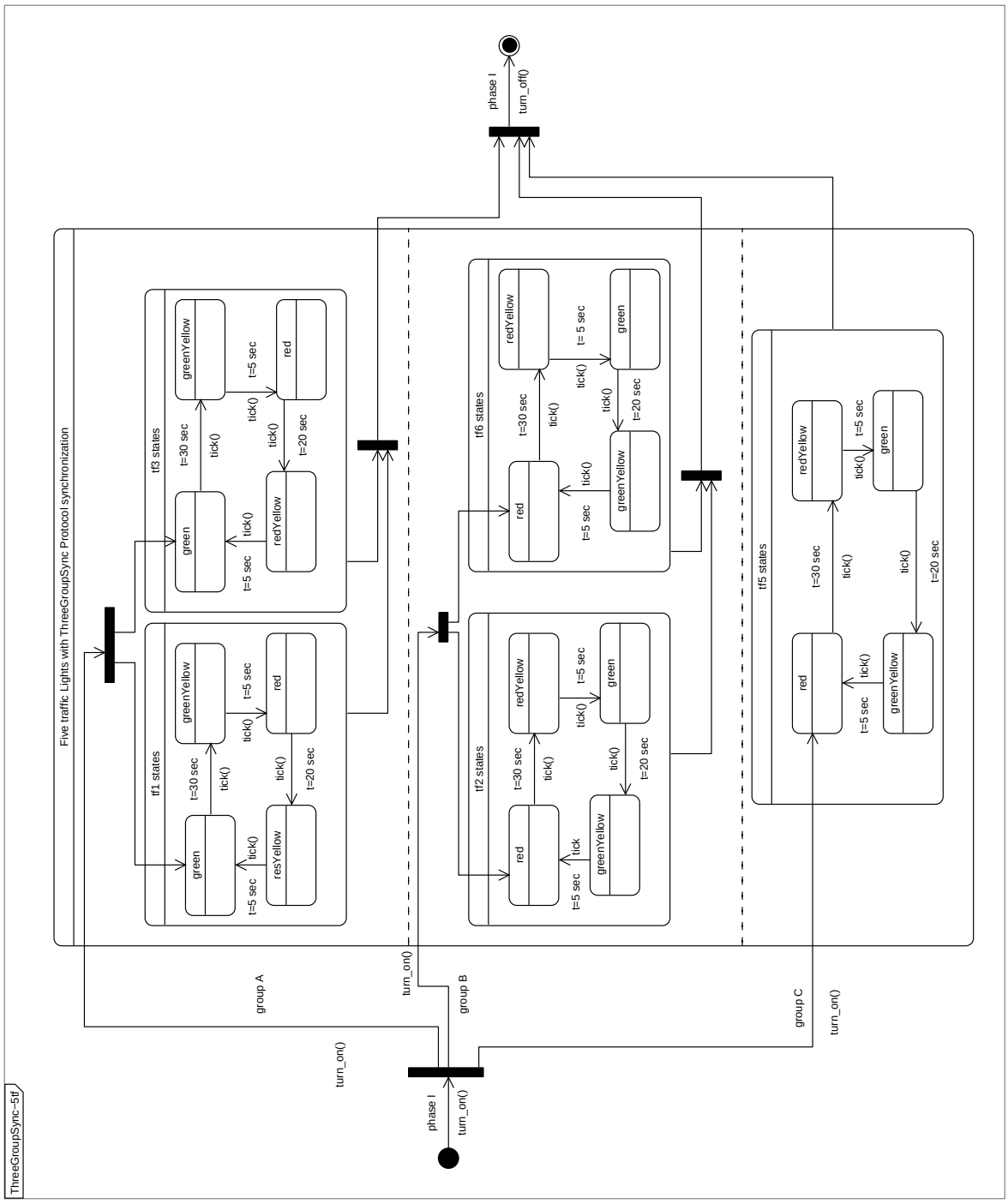



**Figure 6.11:** TwoGroupsSync with 5tf

**First plan close the whole left lane of road section (rs4):** This evolutionary plan composes of two steps:

- turn off the traffic lights $(tf_4, tf_7)$
- create a new synchronization protocol for the remaining traffic lights. $synManager(ThreeGroupsSync(A, B, C), [(tf_1, tf_3), (tf_2), (tf_5, tf_6)])$.

**Second plan close a part of the left lane of road section (rs4):** The second evolutionary plan composed of the following steps:

- delete road section (rs4),
- create new road section instances (rs7 and rs8)
- create new traffic node at the part of road section (rs7 and rs8), $tn_3$,
- add two traffic instances to the new traffic node, $TL_{tn_3} = \{tf_8, tf_9\}$,
- create a synchronization protocol for the new traffic lights, $syncManager(TwoGroupsSync(A, B), [(tf_8), (tf_9)])$.

### 6.3.2. Validation rules

The main role of the consistency checker meta object is to check that the internal representation remains consistent after evolutionary meta-object work.

**In case of the first plan:** The consistency checker has to verify two rules: (1) The initial state of both traffic lights $(tf_4, tf_7)$ is off. (2) every traffic light belongs to the *threeGroupsSync*.

**In case of the second plan:** The consistency checker has to verify the following rules:

- $tn_3 \in TNset_{M2caseA}$,
- rs7 and rs8 created and added,
- $tf_8$ and $tf_9$ created and added,
- synchronization protocol assigned for $tf_8$ and $tf_9$.

**Reachability:** Every traffic node by using set of traffic lights allows different kinds of direction and reachability of the road sections that connected to this traffic node. For example $tf_1$ at the traffic node $tn_1$ allows the following direction: $rs_1 \longrightarrow rs_2$, $rs_1 \longrightarrow rs_3$ and $rs_1 \longrightarrow rs_4$. From the original map for *CaseA*, we get all the direction flow between the connected road sections for both traffic nodes ($tn_1$ and $tn_2$). The role of the consistency checker is to check the adapted layout keeps all the directions flow that exist at the original one, by checking the value of *roadLink* attribute.

### 6.3.3. Samples of script rules for case (A)

In the following, we illustrate the Ruby script for evolving and validating the internal representation of the software systems.

The following code snippet describes how to remove instances from the object model in figure 6.8.

```
# removing tf4 and tf7
trafficNode1.removeInstance(trafficNode1.getInstance("tf4","TrafficLight"))
trafficNode2.removeInstance(trafficNode2.getInstance("tf7","TrafficLight"))
```

In this part of the script code, we illustrate how to adapt the synchronization between traffic lights by adding new region at the statechart, then we add the states for that traffic light instance and its states.

```
# adding the "tf5 states" to the Region_3
top1SiPrC.addState("tf5states")
top1SiPrCSiTl5=top1SiPrC.getAllSimpleState[0]
top1SiPrCSiTl5.addNewRegion("tf5")
top1SiPrCSiTl5r=top1SiPrCSiTl5.getRegion("tf5")
```

This code snippet of the rule adapts the required transitions between simple states for the traffic light instance ("tf5").

```
# add the transitions
top1SiPrCSiTl5rRe=top1SiPrCSiTl5r.getState("red")
top1SiPrCSiTl5rGr=top1SiPrCSiTl5r.getState("green")
top1SiPrCSiTl5rGY =top1SiPrCSiTl5r.getState("greenYellow")
top1SiPrCSiTl5rRY =top1SiPrCSiTl5r.getState("redYellow")

top1SiPrCSiTl5rRe.addTransitionTo (top1SiPrCSiTl5rRY, "", "t=30sec","tick()")
top1SiPrCSiTl5rRY.addTransitionTo (top1SiPrCSiTl5rGr, "","t=5 sec","tick()")
top1SiPrCSiTl5rGr.addTransitionTo (top1SiPrCSiTl5rGY, "", "t=20sec","tick()")
top1SiPrCSiTl5rGY.addTransitionTo (top1SiPrCSiTl5rRe, "","t=5 sec","tick()")
```

In the following code snippet, we check, whether the deployment diagram and statechart diagrams have anything left from the two deleted traffic lights tf4 & tf7.

```
# to check >> tf7 states << # puts
Ramses::StateChartDiagram.new.methods[1..-1]

top1SiPrB = top1.getAllSimpleState[0].getRegion("Region_2")
puts getNames(top1SiPrB) begin
```

```
  top1SiPrB.getState("tf7 States")
  faultReason << "tf7 states are still in the statechartdiagram\n"
  rescue Ramses::StateDoesNotExistsException
end
```

Finally, this part of code snippet checks whether the remaining traffic lights are in the appropriate groups.

```
top1SiPrA = top1.getAllSimpleState[0].getRegion("Region_1")
faultReason << "tf1 states are missing the group a\n" unless
    getNames(top1SiPrA).include?("tf1 states")
faultReason << "tf3 states are missing the group a\n" unless
    getNames(top1SiPrA).include?("tf3 states")
```

This part of script describes how to define the new behavior for the traffic lights that control the traffic flow at the traffic nodes tn3 and tn4.

```
# add the new states to the statechartdiagram twoGroupSync-4tf-4(tn3&tn4)

top3 =$reif.getStateChartDiagram("Statediagram_1")

initState = top3.addInitialState("init")
forkInit =top3.addForkState("fork")
finalState = top3.addFinalState("final")
joinState = top3.addJoinState("join")
initState.addTransitionTo(forkInit, "turn_on()", "", "Phase I")
joinState.addTransitionTo(finalState, "turn_off()", "", "Phase I")

top3Si = top3.addState("tf9-tf10 & tf11-tf12 synchronization states")

top3SiPrA = top3Si.addNewRegion("Region_1")
forkRegion1=top3SiPrA.addForkState("fork region1")
joinRegion1=top3SiPrA.addJoinState("join region1")
joinRegion1.addTransitionTo(joinState, "", "", "")
forkInit.addTransitionTo(forkRegion1, "", "group A", "")
```

## 6.4. Practical Results: A Dynamic Evolution and Validation Prototype

In this chapter, we describe the external libraries that are used by the evolution&validation prototype. Then, we give an overview that illustrates the functionality of the prototype to evolve and validate the internal representation of the software systems.

### 6.4.1. The external libraries required

The evolutionary prototype depends on several libraries, especially on the reification library which provides the functionality of loading, manipulating and saving the XMI design information. For loading and writing the XMI data, the reification library uses the Xerces XML parser. Besides that, some other external libraries are required to achieve the dynamic scripting support of the prototype. Since the manipulation and validation of the design information is done by rules (evolutionary and consistency checking rules) which may change during the lifetime of the application, it is self-evident to use a scripting language for composing these rules. There are many free Java based implementations of different scripting languages available.

#### JRuby

JRuby is a Ruby interpreter written in 100% pure Java and provides most built-in Ruby classes. It supports the interaction with and the definition of Java classes from within Ruby. The API is split into two halves, *low-level* and *high-level*. The low-level is implemented in Java and provides a thin wrapper over Java reflection classes. The high-level is built on top of this, implemented entirely in Ruby. For the prototype, the high-level capabilities are sufficient, for example to use the Java random class in a rule you just have to write:

```
require 'java'
include_class 'java.util.Random'
r = Random.new
puts r.nextInt
```

#### Bean scripting framework

The Bean Scripting Framework (BSF) is a Java library which provides scripting language support within Java applications, and access to Java objects and methods from scripting languages. BSF permits Java applications to be implemented in part (or dynamically extended) by a language that is embedded within it. Therefore it provides an API

that makes it possible to call scripting language engines (like JRuby) from within Java. In addition, it contains an object registry that exposes Java objects to these scripting language engines.

Before using the BSF you have to register the Ruby scripting engine and create an instance of the BSFManager class:

```
BSFManager.registerScriptingEngine("ruby",
    "org.jruby.javasupport.bsf.JRubyEngine", new String[] { "rb" };

BSFManager manager = new BSFManager();
```

The current implementation of the prototype uses global variables to provide access to the reified data from within the Ruby rule. The following line declares a global variable called reif, which refers to an instance of the Reification class.

```
manager.declareBean("reif", reif, Reification.class);
```

Then it can be used transparently in the Ruby rule like any other global Ruby variable, e.g.:

```
cds = $reif.getAllClassDiagram
```

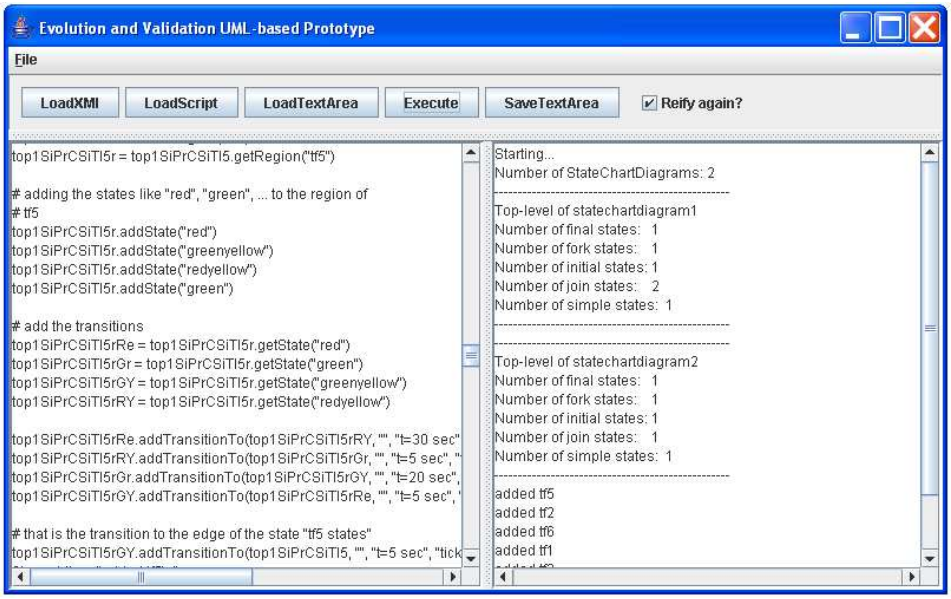



**Figure 6.12:** The evolution prototype interface.

## 6.4.2. The prototype: an overview

The evolution and validation prototype interface as shown in figure 6.12, that provides the ability to evolve and validate the internal representation of the software system.

The structure of UTCS prototype is composed of the five buttons. In the following, we illustrate the roles of these buttons:

**LoadXMI**: The role of this button is to load the internal design information of software system in form of XMI by using RAMSES reification library.

**LoadScript**: Provides the ability to load Ruby script, that describes the designer point of view to evolve or validate the design representation of the software systems.

**LoadTextArea**: Provides the ability to load the edit the script rules and update its before applying its to the loaded XMI schemas.

**Execute**: Applying the script to the loaded XMI schema and saving the modified schema to another file.

**SaveTextArea**: provides the ability to save the modified script to a new script file.

The UTCS interface has additional three components:

**Reify again?**: gives the ability to reify the XMI multi-times to evolve or validate.

**RightTextArea**: uses for traceability the execution of script as log window.

**LeftTextArea**: uses as a editor for scripts.

We have applied the UTCS prototype to the cases we have discussed in the previous sections. For each case, the prototype drives the evolution by loading the normal layout; then applying the suggested evolutionary plan to the XMI schemas. In case of check the consistency, the prototype loads the modified schemas; then applies the required validation plan.

Finally we illustrate in details the typical use of the prototype as follows:

- load XMI-file, by using the button *LoadXMI* or *open* submenu you can load the XMI file that describe the normal layout for the case you want to evolve;
- load script (evolution) in the script-text area. By using the button *loadTextArea*, you can load the evolution scripts in the *LeftTextArea* as shown in the GUI;
- For the scripts loaded at the *LeftTextArea* you have the ability to read it and make the new changes. After that, by using the button *SaveTextArea*, you can save the modified script;
- apply the loaded or modified evolution script to the XMI file by using the button *Execute*;
- sign the checkbox *Reify again?* to reify the XMI file many times;
- To check the XMI file is consistent with changes, we should loaded a consistency script as the same way for evolution script;
- execute the loaded consistency rule to the XMI file;

- you can check the traceability of the execution both the evolution and consistency script by monitoring the log-window (*RightTextArea*).

## 6.5. Summary

A good design is the basis of every good project, this is particularly true when design information should be used to verify and adapt a running system without stopping it. In this chapter we have shown the role of design information in the adaptation of software systems at run-time. The evolution is managed by a reflective middleware that reifies and, when events occur, manipulates system design information (the system meta-data).

Manipulation is realized by Ruby scripts that drive both the adaptation plan and the consistency checks. We have chosen to write evolutionary and consistency rules as Ruby scripts rather than as logic formulas because: they are expressive as well as (maybe more than) logic formulas; (ii) we don't have to write an interpreter for the rules; finally (iii) they can directly interact with the base-level representation through the reflective facilities of Ruby without extra efforts.

The chapter focuses on a case study but it should be clear that the approach is general and usable to evolve most of the software systems.

# 7 Comparison of RAMSES to Related Works

Today's software systems are mostly characterized by an ever increasing size and complexity, rapid changing of requirements, frequent upgrading of supporting technology, and varying contexts for use and deployment. These facts are more acute for information systems due to their long life span, market and economy volatility, aggressive competitiveness to survive increased by the emergence of virtual (dynamic) inter-organizational enterprises. To cope with this high volatility, both industry and academia acknowledge the overwhelming need for putting forward adequate conceptual techniques, methods and tools are able to directly construct self-adapting and dynamically evolving information systems instead of the development life-cycle of constructing and only afterwards considering maintenance and evolution.

There are currently several ongoing proposals dealing with this vivid area of research and practice, namely the sound combination of *computational reflection* issues with different *design information models* as well as the adaptation of such combinations for coping with runtime evolution. Therefore, any attempt toward an exhause comparison of such proposals in this area seems to be premature. Moreover, due to the usual divergence in their objective, its difficult to compare self-adapt setting and main application domains they are devoted to, even well-establish dynamic adaptation.

Nevertheless, it is more or less possible to assess existing dynamic approaches with respect to some adequately selected evolution criteria; where the more the main application domain and self-adapting setting of such framework are close, the more exhaustive and clear becomes such comparison. In this sense, as the RAMSES architecture is essentially dedicated for dynamically evolving the software systems for runtime changes based on its design information, we will restrict ourself only to the approaches that achieve dynamic adaptation. More precisely, as the RAMSES architecture stems from base-level to meta-level with computational reflection, script engines and design information issues, we will carry out our comparative study with respect to these two directions, namely (1) reflective approaches; and (2) analysis and evolution approaches.

On the light of these motivating choices, in the following subsections, firstly, we comment on the selected evolution criteria which are mainly adopted from [67] and slightly enriched with *advanced* criteria including runtime specification evolution. Secondly, we will apply these criteria to the related reflective approaches by classifying the RAMSES as

one of them. Finally, we apply the same criteria to the selected analysis and evolution tools.

## 7.1. Evaluation Criteria

In the following we report on the commonly agreed upon runtime evolution concepts, which assign the self-adaptation to any evolution framework. These concepts are classified into four categories: *meta-based criteria*, *design information-based criteria*, *reflection features*, *dynamic evolution and analysis features*. Besides that, we propose to include a fifth class of criteria more specific to complex information systems as an application domain. Referred to as *advanced features*, criteria of this class allows to assess a given architecture against the capability of handling runtime evolution.

**Meta-based criteria**: This criteria category states the capability of a given framework in fulfilling the following concepts: *monitoring base-system* as the ability of meta-level to monitor the execution of the base-level system; *detecting the runtime events* as the ability to get the runtime events and for each event build his own pre-condition and post-condition; *interpreting the changes* as the ability to adapt the systems by using special meta-objects; and finally, the ability to check the consistency of the modified components by using meta-objects.

**Design information-based criteria**: This includes the ability of driving software evolution and validation based on design-information as follows: first, *horizontal evolution and validation (E&V)* as the ability to evolve and validate the explicit and implicit related components; second, *vertical evolution and validation (E&V)* as the ability to evolve and validate each explicit or implicit components alone.

**Reflection features**: This criteria category specifies the link between the base-level and the meta-level in fulfilling the following concepts: *reifying explicit data* as the ability to reify the base components into the meta-level to constitute the representative system; *reflect the changes to its base-objects* as the ability to reflect all the changes have been done on the representative systems to its base components; and finally, *trap specific models* as the ability to reduce the cost of reification by reifying only the required objects according to the pre-condition of the runtime events.

**Dynamic evolution and analysis features**: This category addresses some analysis and evolution features. The criteria are: (1) *connection to the specification- and concrete- level*, as the ability to directly interact with the both view of the system; (2) *global view*, as the ability to explicit different views from the implicit concrete-level; (3) *dynamic evolution*, as the ability to drive new changes to both *structural* and *behavioral* representation.

**Advanced features**: The first criterion we consider in this category is the ability to apply the *feedback concepts* to the reified systems to include the new data and the

also in the inconsistent states; *continuing changes* as the ability to self-adapt the reified system whenever there is runtime events; *failure avoidance (robustness)* as the ability to avoid an inconsistence states at some level; and finally, *organisational stability* as the ability of the modified system to be consistent against the changes.

In the following, we overview some of the related work by focusing more on approaches which are of interest to our thesis. In the last few years, there has been a growing interest for dynamically evolving software systems. In the literature, there are several approaches related to building adaptive software systems by allowing system behavior to evolve after design time. Several of such approaches propose reflective architecture-based mechanisms for dynamic evolution. In addition, there are some related tools are used to support analysis and evolution of the software systems. The related topics could be summarized by categorizing them within the research directions as follows.

## 7.2. Reflective Approaches and RAMSES

In [5], software evolution is defined as a kind of software maintenance that takes place only when the initial development was successful. The goal consists of adapting the application to the ever-changing, and often in an unexpected way, user requirements and operating environment.

Software evolution, as well as software maintenance, is characterized by its huge cost and slow speed of implementation. Often, software evolution implies a redesign of the whole system, the development of new features and their integration in the existing and/or running systems (this last step often implies a complete rebuilding of the system). Besides, software systems are often asked for promptly evolving to face critical situations such as to repair security bugs, to avoid the failure of critical devices and to patch the logic of a critical system.

It is fairly evident the necessity of improving the software adaptability and its promptness without impacting on the activity of the system itself. This statement brings forth the need for a system to manage itself to some extent, to dynamically inspect component interfaces, to augment its application-specific functionality with additional properties, and so on. Non-stopping applications with a long life span are typical applications that have to be able to dynamically adapt themselves to sudden and unexpected changes to their environment. Therefore, the support for run-time adaptive software evolution is a key aspect of these systems. Software evolution of a generic system is usually carried out by stopping the system and manually, or with the aid of specific tools, changing the system behavior according to the required evolution. A more dynamic approach consists of encapsulating the system prone to be adapted in a monitoring system that waits for an event. When the event occurs, it plans a countermove that will imply the automatic and dynamic evolution of the monitored system. The monitoring system also takes care to grant the safety and stability of the monitored system against its evolution.

Most systems that offer computational reflection at runtime are based on the use of a meta-object protocol (MOP). MOPs give a system the ability to customize at runtime, but what may be adapted must be previously specified by the protocol. Different approaches modifying the MOP are commonly needed to make the system adaptable to a new characteristic. The system we consider in this overview are:

### 7.2.1. The K-Component architecture

*K-components* [38] uses asynchronous architectural reflection to build context-aware adaptive software. The adaptation logic that specifies the adaptive behaviour (adaptation policy) is written as adaptation contracts in a declarative programming language (ACDL). Adaptation occurs in response to adaptation events raised by either the application components or from the evaluation of adaptation rules themselves, e.g., anticipated failure to achieve a set goal. The meta-level configuration manager runs asynchronously and so periodically reflects on the need for adaptation, using polled adaptation events and the adaptation contracts, thereby greatly reducing reflective computation overhead. The reified software architecture is arranged as a typed directed component configuration graph, where changes to the configuration graph during dynamic adaptation are performed as transactional operations, so that the result is again a correct directed configuration graph. If adaptation is required, a component can be removed from the system configuration graph and another component, exposing the same interface, can be swapped in. A component's external interface cannot be changed by architectural reconfiguration since only reconfiguration operations on the configuration graph is allowed. This maintains correctness of the component configuration graph but severely restricts how the system can adapt.

New components can be loaded at runtime from a DLL or as a remote CORBA component, but their interface must be previously specified since the configuration graph is a static representation of the architecture of the system, and cannot be extended to support new component types at runtime. The system also requires that the adaptation event types are known to the configuration manager at compile-time, so very little support is included to initiate adaptations in response to unanticipated or un-typed events, as will likely occur in a mobile or pervasive computing environment.

### 7.2.2. Architectural reflection

Architectural Reflection, [27] presented a novel approach to reflection called *architectural reflection* which allows dynamic adaptation of a system through its design information. Software architecture manipulation allows adaptation in-the-large of the system, i.e., one can add and remove components but cannot add new functionality to a component. The strategic reflection has presented, an aspect of architectural reflection, which is an extension of classic reflection to the software architecture level. The basic application of

this extension is to allow for a systematic and conceptually clean approach to designing systems with self-management functionality (such as dynamic reconfiguration) which also supports such functionality to be added to an existing system without modifying the system itself. In the rest of this subsection, we introduce in details two approaches based on architecture reflection: an adaptive and reflective middleware (ARM) [41, 40] and an architectural reflection for software evolution [88, 89].

In [41, 40] present an adaptive and reflective middleware (ARM). The ARM is composed of two layers: the first layer defines the reflective knowledge. It reifies the system's components in terms of reflective objects and their related quality of services (QoS). The second layer introduces the view concept that is representing an organizational mechanism of the reflective knowledge. Views organize reflective objects based on their QoS, structure, location, and topology. Each view has associated strategies that implement the logic necessary to take decision. The mechanism that allows the management of the reflective knowledge is represented by views. The ARM achieves the adaptivity by *reflective objects*, *views*, and *strategies*. However, this architecture is limited to reconfiguration of views and reflected objects and such is not really able to create new solutions. The software systems with this approach are partially evolved on-site without going back to software factory and no new algorithms are really invented.

A framework presented in [88, 89] allow evolution at the architecture level. This framework uses a reflective layer to maintain and control meta-level information. Allowing visibility and manipulation of meta-level information gives the maintainer of software the ability to compose software at the architecture level. The framework fails to achieve all of the safety requirements. In this case, no new non-functional requirements are really added to evolve the system. Non-functional requirements can be connected and disconnected at run-time, but new non-functional cannot be included online, as soon as they are available.

### 7.2.3. Co-operative actions (CO Actions)

A reflective architecture by using a cooperative object-oriented style presented in [106]. Structural elements of this approach are classes as basic components, and (*CO Actions*) to represent interactions among objects characterizing the collaborative behavior [35] as basic connectors. This approach achieves adaptability by: (1) dynamically extending objects behavior using roles, and (2) selecting at run-time the objects and roles participating in a cooperation. Reflection has been used to clearly separate and intertwine the description of functionality, synchronization, interaction, and adaptation. By encapsulating each description into a different component of the proposed architecture it is possible to support reconfiguration, and evolution is enhanced. Although the separation of concerns increases the number of components, it helps reducing complexity since the various components can be understood and altered independently. An architecture pattern has presented to support run-time adaptability targeted for co-operative object-oriented architectures. Decisions on changing or adding objects are not automated, but

a human expert operates on the architecture by configuring the role of *Co Actions* so that objects are appropriately changed at run-time. Moreover, this architecture does not deal with introducing into an application new class versions at run-time, but only with re-configuring loaded application classes.

### 7.2.4. DART

DART (Distributed Adaptive Runtime) [90] is a runtime reflective framework for distributed adaptation, developed by Sony. A framework for reflective objects is provided to support functional behaviour adaptation of the application, which operates by allowing alternative method implementations (adaptive methods) to be selected via *selectors*, in a manner similar to the Strategy design pattern [44]. Also included is a method interception system (reflective methods) for non-functional behaviour adaptation in response to environmental changes. Using this approach, intercepted method calls are redirected to a set of meta objects before and after invocation using a *reflector*, which manages these meta-objects. A runtime manager is instantiated for each application as it starts up. Adaptation policy functions, written in C, register for adaptation events and can introspect on both the base-level and meta-level code.

DART does not support the dynamic specification of new adaptations, and many aspects of the adaptation must be anticipated at or before runtime. DART is of interest to the RAMSES project for its support for named object specifications, event driven dynamic adaptation of functional and non-functional behaviours, adaptation using behavioural reflection, and a configuration mechanism that can be adapted for individualised adaptation policies.

### 7.2.5. Some comments on the comparison

After sketching the feature of the each of five reflective approaches for evolution, table 2.1 summarizes the result of each approach with respect the afore-described evolution criteria categories. The RAMSES features are assumed to be largely understood from the previous chapters, and therefore their results are directly reported in this table. The used legends are: '$\surd$' standing for Yes; '$\times$' standing for No and '$+/-$' for a non satisfactory fulfillment of the criterion.

In this table we notice that the RAMSES does not support the criteria item organisational stability and there are two criteria items not yet satisfied such as: detecting the runtime events and reflect changes to it base objects. Whereas K-components does not support, for instance, detecting the runtime changes, robustness, couple connection between specification and concrete, and organisational stability. The criteria items not yet satisfactory by K-component for instance: vertical E&V, trap specific models, global view, and feedback. The drawbacks for *Architecture reflection* present in the

| | RAMSES | K-Component | Architectural Reflection | CO actions | DART |
|---|---|---|---|---|---|
| Meta-based criteria | | | | | |
| monitoring base-system | √ | √ | √ | √ | √ |
| detecting the runtime events | +/− | × | × | √ | √ |
| interpreting the changes | √ | √ | +/− | +/− | √ |
| consistency management | +/− | √ | × | × | × |
| Design information-based criteria | | | | | |
| horizontal E&V | √ | +/− | √ | × | √ |
| vertical E&V | √ | √ | × | × | × |
| Reflection features | | | | | |
| reifying explicit data | √ | √ | √ | √ | √ |
| reflect changes to its base objects | √ | √ | √ | +/− | √ |
| trap specific models | √ | +/− | +/− | × | √ |
| Dynamic analysis and evolution features | | | | | |
| specification and concrete are connected | √ | × | × | × | × |
| global view | √ | +/− | √ | +/− | +/− |
| dynamic adaptation | √ | √ | √ | √ | √ |
| Advanced features | | | | | |
| feedback system | √ | +/− | × | +/− | +/− |
| continuing change | √ | √ | √ | √ | √ |
| failure avoidance (Robustness) | +/− | × | × | × | × |
| organisational stability | × | × | × | × | √ |

**Table 2.1:** Comparison of RAMSES with reflective architectures

following criterions: dealing with runtime events, consistency management, interpreting the changes,trap specific models, and specification and concrete connected. Also all the advanced features criteria except the continuing change criterions. Concerning the *Co-action* does not satisfy the design information-based criteria, however, the following criterions: interpreting changes, consistency management, specification and concrete connection, robustness and stability. Finally, the DART does not satisfy the following criterions: consistency management, specification and concrete connected, global view, feedback and robustness.

## 7.3. Analysis&Evolution Approaches and RAMSES

In this section, we divided the analysis and evolution approaches into three categories as described in chapter 2.4. Here, we focus on the tools that are used at each category to support analysis or evolution.

**re-engineering tools:** we selected the re-engineering tools that automatic recovery and drive partially or totally evolution;

**impact analysis tools:** that use to static or dynamically analysis of the software system;

**refactoring tools:** that automatically restructuring the software system.

In the rest of this section, we enumerate the related tools for each category.

### 7.3.1. Re-engineering tools

Re-engineering approaches all seek to represent the software at a higher level than that of the different information which is directly extracted from the code. They differ, however, in their solutions to the following main issues: the data model on which the tool operates, the strategy for creating a high-level model and the kinds of view offered.

The MANSART tool [50], requires information obtainable from an abstract syntax tree (AST) of the program, and uses *recognizers* to detect language-specific cliches associated with specific architecture styles. Each style can then be viewed as a simple graph.

*Rigi* [107] and the reflexion model tool [78] both use any set of relations extracted from the code. The *Rigi* tool incorporates automatic clustering, but also allows user defined grouping of the source model. It allows for hierarchically embedded views of different relations and presents a sophisticated user interface for manipulating these. The Reflexion model approach expects an engineer to define a high-level model and a declarative mapping from the source relations to this model. Its view then reports how close the high-level model comes to describing the source code. Dali [57] is a workbench

which integrates several extraction tools and allows for the combination of the views obtained from these different sources.

In terms of the data model views required, our middleware is using reification library to explicit the higher level views. These views describe both structural and behavioral of the system. We see the strength of our approach is the flexibility of separation of concern the evolution of these higher views from the applications.

### 7.3.2. Impact analysis tools

Program traces have been used in software maintenance to locate code implementing a particular program feature. For understanding of object-oriented software, much of the work on using dynamic information has focused on techniques for visualizing the large amount of information. *ISViS* [56] is a visualization tool which displays interaction diagrams using a mural technique and also offers pattern matching capabilities to aid in identifying recurring patterns of events. In [107] use program animation techniques to display the number of objects involved in the execution, and the interaction between them through user-defined high-level models.

*Chianti*: a change impact analysis tool for java that is implemented in the context of *Eclipse* environment [91]. *Chianti* has been implemented in the context of the Java editor of Eclipse, a widely used extensible open-source development environment for java. This tool is responsible for driving a set of atomic changes from two versions of an Eclipse project (i.e., java programs), which is archived via a pairwise comparison of the abstract syntax trees of the classes in the two project versions.

Most of impact analysis tools explicit static and dynamic information from the concrete-level and drive the evolution through the elements specified in the impact change categories. Our middleware explicit static and dynamic information in form of UML models and its computerized form XMI, the both forms are familiar for the designer and developer. The impact analysis tools drive atomic changes identified with concrete-level (e.g., add a class, delete a method, add a field). In contrast, our approach apply all the possible changes in the requirement by using script engines, that directly interacts with the explicit views. Finally, The impact analysis techniques are aimed at deployed concrete-level, in that they are interested in obtaining user patterns of concrete-level execution. In contrast, our approach are intended for use during the earlier phases of software development, to give developers immediate feedback on changes they make.

### 7.3.3. Refactoring tools

The nature of applying refactorings is very much language specific. The early phases of developing refactoring (restructuring) concentrated on block-structures an object-oriented languages. A number of more recent tools also support refactoring: the *Smalltalk*

*Refactoring Browser* [93], which automatically performs a set of refactorings taken primarily from Opdyyke's original work; the *IntelliJ Renamer tool* (www.intellij.com), which supports renaming of packages, variables, etc. and moving packages and classes for java; and the *Xref-Speller* (www.xref-teck.com/speller/), which extends the Emacs editor to support a set of refactorings for C and for Java.

Bowdidge's *Star Diagram* [12] is a visualization technique that represents a high level abstraction of a programs structure. The tool can be used to understand a C++ program through a graphical tree structure which can be interactively changed to restructure the program. However the *Star Diagram* has limitations in relation to the graphical representation of a large problem becomes unmanageable coherent. The restructurings which can be applied are also limited since the visualization is too high level to consider what is happening within a method. The tool does not recommend specific refactorings, and the tool user must identify the variables or data structures that are candidates for refactorings.

Moore's *Guru* tool [77] automates two specific and somewhat more global refactorings. It employes a graph-based inheritance hierarchy inference algorithm that can automatically restructuring an inheritance hierarchy and refactors methods written in self programs. Restructuring the hierarchies are based upon "maximising sharing and minimising duplication of the features (mostly methods) of objects and concrete classes". After restructuring a hierarchy, *Guru* produces a totally new structure as this produces better results. However this automation of a new hierarchy will effect a developers mental model of the system. Using a similar algorithm *Guru* can also automatically extract shared expressions from methods.

In [92] presented the case of refactoring scenario (e.g., *Extract Method*) that leading to evolution information loss. From our point of view, refactorings is the best way to evolve and validate each component of the concrete-level. For evolving and validate the specific behavior that gathering from different concrete-level components, our middleware presents dynamic views at the specification-level.

### 7.3.4. Assessment against the evolution criteria

In table 3.1 we reported the result of the features of each approach, the RAMSES included, against the selected evolution criteria. As shown in this table, the RAMSES architecture does not support the criterion stability and not yet satisfy the criterions: detecting the runtime events, reflect to base object, and robustness. Whereas *impact analysis tools* do not support, for instance, design information-based criteria and the following features: specification and concrete connection, global view, feedback system, and robustness. The *re-engineering tools* do not support the following criterions: detecting runtime events, vertical E&V, dynamic adaptation, robustness, and stability. Finally, *refactoring tools* do not support the following features: interpreting the changes, con-

| | RAMSES | reengineering tools | Imapct analysis tools | Refactoring tools |
|---|---|---|---|---|
| Meta-based criteria | | | | |
| monitoring base-system | √ | √ | √ | √ |
| detecting the runtime events | +/− | × | × | √ |
| interpreting the changes | √ | √ | √ | +/− |
| consistency management | √ | √ | × | × |
| Design information-based criteria | | | | |
| horizontal E&V | √ | √ | × | × |
| vertical E&V | √ | +/− | × | √ |
| Dynamic analysis and evolution features | | | | |
| specification and concrete are connected | √ | √ | +/− | × |
| global view | √ | √ | +/− | +/− |
| dynamic adaptation | *p* | × | √ | √ |
| Advanced features | | | | |
| feedback system | √ | √ | × | +/− |
| continuing change | √ | √ | √ | √ |
| failure avoidance (Robustness) | +/− | × | × | √ |
| organisational stability | × | × | √ | × |

* p= Partially applied only into the design information.

**Table 3.1:** Comparison of RAMSES with analysis and evolution tools

sistency management, specification and concrete connection, global view, feedback, and stability.

## 7.4. Summary

This chapter has presented the results of comparing the planning phase of RAMSES middleware with the related reflective architectures and analysis&evolution tools. We have defined set of evaluation criteria. The RAMSES middleware and the related approaches have been evaluated against this set of evaluation criteria.

The next chapter concludes this thesis with a summary of the contributions of the RAMSES project, along with a discussion of open research questions and suggestions for further work in this area.

# 8 Concluding Remarks

The purpose of this closing chapter is to recapitulate the achieved research results and main contributions of this thesis. Furthermore, we point at some future research directions around the RAMSES proposal, that we think are very worthwhile to pursue.

## 8.1. Recapitulation

Most of today's research on software evolution is concerned with processes supporting adaptation and maintenance. Based on its results, predictions about future directions and expected modifications are made that can serve as aids for future self-adapting activities. With increasing deployment and use of continuously running systems, additional effort seems best to be made on allowing such software systems to be evolved without taking them offline or even shutting them down for maintenance, upgrade, or other related activities, are become more than ever a challenging task. Particulary, for crucial phase of software evolution, it is widely acknowledged that any suitable self-adaptation approach, able to absorb this ever-increasing complexity, has to fulfill at least the following requirements:

- A clear separation of concerns between the application functionality and the adaptation processes. All the code necessary to make the application aware of the execution environment as well as the code that defines the adaptation actions are encapsulated inside the evolutionary objects;
- A software system should be able to modify itself to improve system response time, recover from a subsystem failure, or incorporate additional behavior during runtime;
- A software system should be opened to adaptive if new application behaviors realized and adaptation plans can be introduced during runtime;
- On-the-fly reduction of the gap between software design and software evolution, to get a good evolution we have to pass through the evolution software design;
- Dependency-preserving evolution. By maintaining an explicit representation of the prerequisites and dynamic dependencies in the software system, the evolutionary objects acquires the necessary knowledge to adapt the system in a safe and consistent way;

- A set of policies should be used to dynamically adapt to changing system circumstances in order to continue to meet system requirements;
- Last but not least an evolution framework has also support computational reflection properties.

The research we carried out in this dissertation stems from observation that in spite of the effort undertaken in recent years, we are far from a widely accepted evolution approach fulfilling all the mentioned (minimal) evolution requirements. In particular the challenging issues transcending existing approaches concern dynamically evolving the software systems.

In a contribution towards such a suitable evolutionary approach, we proposed in this thesis a new proposal based on these requirements. The middleware, referred to as RAMSES, is based on a reflective architecture. This middleware uses design information for driving evolution and reducing the gap between software design and software evolution. In the meta-level of this middleware, we defined two evolutionary meta-objects that use the script engines for satisfying the runtime evolution and consistency. The middleware has been compared with most of existing similar approaches, and a non-trivial case study has been drawn up enhancing it practicality.

In some detailed, after introducing in chapter 2, necessary motivation and preliminary concepts, in chapter 3, we presented the *design information* as a global view impacting of the whole system, then, we analysed the explicit and implicit view through example, finally, the first contribution, we defined the design information taxonomy and presented the lightweight formalisation of this taxonomy representative the basic features of UML.

The fourth chapter has been devoted to illustrate how to build an evolution and validation planning to evolve and validating the design information system for changes. We formally defined in general the evolution and validation strategy, then we completely analysed the processes of evolution and validation through algorithms. Finally, we defined the rule-based script that automatically evolving the design information based on the evolution and validation plan.

The middleware was the subject of the fifth chapter. In context, we have introduced a reflective architecture for supporting runtime evolution. More precisely, RAMSES helps in significantly evolving and validating the information systems by separating the adaptation concerns of an information system from its internal concerns. During this chapter, firstly, we described and defined the whole RAMSES infrastructure. secondly, we presented the RAMSES reification library, that describes the reification and reflection of the design information through a set of examples. Finally, we defined the role of the RAMSES meta-level for driving the evolution and validation. By describing functionality of the evolutionary and consistency checker meta-objects.

After putting the structure for the RAMSES middleware. In chapter 6, we carried on UTCS case study. To show how our RAMSES middleware works, we built three cases of the UTCS case study. In particular, we analysed each case with a specific runtime

event, then built the required evolution and validation strategy to adapt the system in according to the realized event. On the other hand, we showed how to built the scripting engines by providing a set of Ruby rules. Finally, we presented the evolution and validation prototype to automatically evolved and validated the design information. We have tested our prototype by using the defined UTCS cases.

In chapter 7, we situate the RAMSES proposal with respect to most of existing similar approaches, including reflective architectures and software development approaches.

Some of the results prior to this dissertation have been published in [22, 20, 19, 21, 23, 46, 24, 26]. Finally, our approach to software evolution has the following benefits:

- evolution is not tailored on a specific software system but depends on its design information;
- evolution is managed as a nonfunctional features, therefore, can be added to every kind of software system without modifying it; and
- evolution strategy is not hardcoded in the system but it can dynamically change by substituting the evolutionary and validation rules.

## 8.2. Further future work

After this crucial first step towards evolving and validation the software systems for the changes in their environment based on design information using the proposed RAMSES middleware. We are conscious that much work remains ahead to software evolution in result in a largely acceptable self-adapting approach. In this section, we identify and summary at least two relevant phases for extending the RAMSES middleware as presented in this thesis; where, for each foreseen extension we shed some lights, even very superficial it is, on our thinking about possible solutions.

### 8.2.1. Reflecting the planned evolution by the AOP on the base-system code

This phase deals with reflecting the modified design information from the meta-level to the base-level. It completes the overall adaptation process. One key question of this phase is how design information can be reflected to code level. State charts and class diagrams are abstract views of the system behavior and structure. It is not trivial to map changes of these information to changes at code level. A powerful mapping mechanism is the goal of this package. In [25] first results show the way how to accomplish that. We argue that extended design information which includes mapping information can help to implement reflection also for domain-specific design information. A second challenge is the modification of code at run-time. Regarding this question we want to use current

techniques of the AOP community. We want to investigate how to use run-time aspect weaving for our reflection process. An analysis of current approaches (load-time vs. run-time weaving, proxy-based vs. instrumented code approaches) will reveal an appropriate solution.

Concerning the step of the reflection of the modified meta data back to the basis level, this phase will analyze different techniques for sound time code creation/modification. The goal to be achieved is not just new code load or weaving techniques but also on their integrableness. As stated before all the reflective and evolutionary activity is performed on a reification of the design information (reflection should change its application domain) whereas the actuation of the evolution directly involves the system code.

In this phase we will explore the application of the aspect-oriented techniques to this job. They are the perfect tools for instrumenting the code when driven from other code but in this work the instrumentation should be driven by the design. So the expected result of this phase is a novel approach to the aspect-oriented software development that use the design information to weave and instrument the code.

### 8.2.2. Formal underpinning

We consider this phase as crucial as it allows us to recapitulate on the experience gained in chapter 5, on guiding us towards a disciplined way of self-adapting and dynamically reconfiguring information systems. For this purpose, we endeavor to capitalize on first result [16], that shows a petri nets based reflective framework that leads a system able to evolve, keeping separated functional aspects from evolutionary ones and applying evolution to the model if necessary. More precisely, as UML is becoming the standard defacto for software development. That is, we plan to exploit the class roles with *OCL2* for endowing interfaces with pre- and post-condition constraints to be compliant with the corresponding components. Finally, the corresponding domain- and platform- dependent aspects will be investigated so that all what can happen at run-time will be validated at design time in a formal way.

### 8.2.3. Dynamic adaptation with reflective graph-transformations

For this phase we propose to recapitulate on the work on K-components [38] and also reflective-based process based level [33]. We have to adapt these work to our. More precisely in this phase, we highlight the following points: firstly, abstraction of components and their connectors to graph transformation. In this task the aim to capture the coarse-grained architecture of the conceived and validated conceptual model in term of nodes and links and rules reflecting its behavior. Secondly, proposition of evolving transformation rules. These rules should reflect the reconfiguration laws that the system

has to bind itself to, according to the nature of the domain application, the events to be intercepted and the context allowed by the application. The work proposed in [110] and related investigations will be our main inspiration for this task. That is, we will capitalize on the expertise and previous experiences in graph-based techniques to adopt a logical view of configurations as diagrams (labeled graphs) and reconfiguration as a rewrite process defined over graph-transformations. Finally, abstraction of the middleware evolutionary scripts and consistency checking rules to graph transformation rules. This task should be regarded as a complementary and a formalization to the second phase, where consistency checking rules for evolution and self-adaptation are proposed. That is, to allow formally reasoning about these implementation-driven rules, we propose to abstract them at a higher level where we can validate and reason about them using the formal frameworks we propose.

# Bibliography


[1] Mehmet Akşit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In *Proceedings of Object-Based Distributed Programming (ECOOP'94 Workshop)*, Lecture Notes in Computer Science 791, pages 152–184. Springer-Verlag, July 1994.

[2] Robert Arnold and Shawn Bohner. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, June 1996.

[3] Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. pages 292–301. IEEE Computer Society, 1993.

[4] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(1):225–252, 1976.

[5] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM Press.

[6] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, 1989.

[7] Daniel G. Bobrow, Richard G. Gabriel, and Jon L. White. CLOS in Context - The Shape of the Design Space. In Andreas Pæpcke, editor, *Object Oriented Programming: The CLOS Perspective*, pages 29–61. MIT Press, 1993.

[8] B. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy. Using the Win-Win Spiral Model: A Case Study. In *IEEE Computer*, pages 33–44, 1998.

[9] B.W. Boehm. A Spiral Model for Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[10] Grady Booch. *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., second edition, 1994.

[11] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.

[12] Robert W. Bowdidge and William G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Trans. Softw. Eng. Methodol.*, 7(2):109–157, 1998.

[13] Lional C. Briand, Yvan Labiche, and Tao Yue. Vertical impact analysis of uml models. Technical Report Technical Report SCE-06-06, Carleton University, April 2006.

[14] Lionel C. Briand, Yvan Labiche, and L. O'Sullivan. Automated impact analysis of uml models. *Journal of Systems and Software*, 79(3):339–352, 2006.

[15] Jean-Pierre Briot and Pierre Cointe. Programming with Explicit Metaclasses in SmallTalk-80. In USA Portland, Oregon, editor, *Proceedings of OOPSLA'89*, volume 24(10) of *Sigplan Notices*, pages 419–431. ACM, October 1989.

[16] Lorenzo Capra and Walter Cazzola. A Petri-Net Based Reflective Framework. In *Proceedings of the IPM International Workshop on Foundations of Software Engineering (FSEN'05)*, Tehran, Iran, on 1st-3rd of October 2005.

[17] Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, on 20th-24th July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.

[18] Walter Cazzola. *Communication-Oriented Reflection: a Way to Open Up the RMI Mechanism*. PhD thesis, Università degli Studi di Milano, Milano, Italy, February 2001.

[19] Walter Cazzola, James O. Coplien, Ahmed Ghoneim, and Gunter Saake. Framework Patterns for the Evolution of Nonstoppable Software Systems. In Pavel Hruby and Kristian Elof Søresen, editors, *Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02)*, pages 35–54, Højstrupgård, Helsingør, Denmark, on 20th-22nd of September 2002. Microsoft Business Solutions.

[20] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Reflective Analysis and Design for Adapting Object Run-time Behavior. In Zohra Bellahsène, Dilip Patel, and Colette Rolland, editors, *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'02)*, Lecture Notes in Computer Science 2425, pages 242–254, Montpellier, France, on 2nd-5th of September 2002. Springer-Verlag. ISBN: 3-540-44087-9.

[21] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. RAMSES: a Reflective Middleware for Software Evolution. In *Proceedings of the 1st ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04)*, in 18th

European Conference on Object-Oriented Programming (ECOOP'04), pages 21–26, Oslo, Norway, on 15th June 2004.

[22] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In Hans-Dieter Ehrich, John-Jules Meyer, and Mark D. Ryan, editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, February 2004.

[23] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. System Evolution through Design Information Evolution: a Case Study. In Walter Dosch and Narayan Debnath, editors, *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE 2004)*, pages 145–150, Nice, France, on 1st-3rd of July 2004. ISCA.

[24] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. ViewPoint for Maintaining UML Models Against Application Changes. In Joaquim Filipe, Markus Helfert, and Boris Shishkov, editors, *Proceedings of the 1st International Conference on Software and Data Technologies (ICSOFT'06)*, pages 263–268, Setúbal, Portugal, on 11th-14th of September 2006. INSTICC. ISBN: 978-8865-69-4.

[25] Walter Cazzola, Sonia Pini, and Massimo Ancona. Aop for software evolution: a design oriented approach. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, pages 1346–1350. ACM, 2005.

[26] Walter Cazzola, Sonia Pini, Ahmed Ghoneim, and Gunter Saake. Coevolving application code and design models by exploiting metadata. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11 - 15, 2007, toappear*. ACM, 2007.

[27] Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Rule-Based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level. In *Proceedings of 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, pages 263–266, Cocoa Beach, Florida, USA, on 12th-15th October 1999.

[28] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishing, 2000.

[29] Pierre Cointe. MetaClasses are first class objects: the ObjVLisp model. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22(10) of *Sigplan Notices*, Orlando, Florida, USA, October 1987. ACM.

[30] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[31] Fábio M. Costa, Hector A. Duran, Nikos Parlavantzas, Katia B. Saikoski, Gordon Blair, and Geoff Coulson. The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 79–99. Springer-Verlag, Heidelberg, Germany, June 2000.

[32] Todd Cotton. Evolutionary Fusion: A customer Oriented Incremental Life cycle for Fusion. In *Hewlett-Packard Journal*, pages 1–5, 1996.

[33] C.E. Cuestaa, P. Fuentea, and E. Barrio-Solorzanoa, M.and Beatob. An Şabstract process$\check{\text{T}}$ approach to algebraic dynamic architecture description. *The Journal of Logic and Algebraic Programming xx (2004)*, 2004. to appear.

[34] W. Curtis, Herb Krasner, Vincent Y. Shen, and Neil Iscoe. On Building Software Process Models Under the Lamppost. In *Proceedings of the 9th International Conference on Software Engineering*, pages 96–105, Monterey, California, USA, 1987. ACM Press.

[35] Roger de Lemos and Alexander Romanovsky. Coordinated Atomic Actions in Modelling Object Cooperation. In *Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, volume 30 of *Sigplan Notices*, pages 152–161, Kyoto, Japan, April 1995.

[36] François-Nicola Demers and Jacques Malenfant. Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, Montréal, Canada, August 1995.

[37] Ralph Depke, Gregor Engels, Sebastian Thöne, M. Langham, and B. Lütkemeier. Process-oriented, consistent integration of software components. In *COMPSAC*, pages 13–18. IEEE Computer Society, 2002.

[38] Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, LNCS 2192, pages 81–88, Kyoto, Japan, September 2001. Springer-Verlag.

[39] Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of 4th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.

[40] Francesca Arcelli Fontana, Claudia Raibulet, and Francesco Tisato. Exploiting reflection for software architectures. In Jaelson Castro and Ernest Teniente, editors, *Advanced Information Systems Engineering, 17th International Conference,*

*CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings of the CAiSE'05 Workshops, Vol. 2*, pages 109–123. FEUP Edições, Porto, 2005.

[41] Francesca Arcelli Fontana, Claudia Raibulet, and Francesco Tisato. Modeling qos through architectural reflection. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the International Conference on Software Engineering Research and Practice, Las Vegas, Nevada, USA, June 27-29, 2005, Volume 1*, pages 347–363, 2005.

[42] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition* . Addison-Wesley, Reading, Massachusetts, 2003.

[43] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.

[45] Carlo Ghezzi, Mehdi Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, 1991.

[46] Ahmed Ghoneim, Sven Apel, and Gunter Saake. Evolutionary software life cycle for self-adapting software systems. In Chin-Sheng Chen, Joaquim Filipe, Isabel Seruca, and José Cordeiro, editors, *ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems, Miami, USA, May 25-28*, pages 211–216, 2005.

[47] Tom Gilb. *Principles of Software Engineering Management.* Addison-wesley, 1988.

[48] Brendan Gowing and Vinny Cahill. Making Meta-Object Protocols Pratical for Operating Systems. In *Proceedings of 4th International Workshop on Object Oriented in Operating Systems*, pages 52–55, April 1995.

[49] Timothy J. Grose, Gary C. Doney, and Brodsky Stephan A. *Mastering XMI: Java Programming with XMI, XML, and UML.* John Willy & Sons, Inc., April 2002.

[50] David R. Harris, Alexander S. Yeh, and Howard B. Reubenstein. Extracting architectural features from source code. *Automated Software Engineering*, 3(1/2):109–138, 1996.

[51] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM Press.

[52] ISPSE2000. International Workshop on the Principles of Software Evolution. Kanazawa, Japan, November 2000.

[53] ISPSE98. International Workshop on the Principles of Software Evolution. Kyoto, Japan, 20 1998.

[54] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 2–10, New York, NY, USA, 1994. ACM Press.

[55] Ivar Jacobson, Magnus Christerson, Patrick Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A use Case Driven Approach.* Addison wesely, 1992.

[56] Dean Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 56, Washington, DC, USA, 1997. IEEE Computer Society.

[57] R. Kazman and S. J. Carrière. View extraction and view fusion in architectural understanding. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 290, Washington, DC, USA, 1998. IEEE Computer Society.

[58] Chris F. Kemerer and Sandra Slaughter. An empirical approach to studying software evolution. *IEEE Trans. Softw. Eng.*, 25(4):493–509, 1999.

[59] Marc-Olivier Killijian, Jean-Charles Fabre, Juan-Carlos Ruiz-Garcia, and Shigeru Chiba. A Metaobject Protocol for Fault-Tolerant CORBA Applications. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS'98)*, pages 127–134, 1998.

[60] Fabio Kon, Roy Campbell, and Manuel Román. Design and Implementation of Runtime Reflection in Communication Middleware: the DynamicTAO Case. In *Proceedings of ICDCS'99 Workshop on Middleware*, 1999.

[61] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[62] David Chenho Kung, Jerry Gao, Pei Hsia, F. Wen, Yasufumi Toyoshima, and Cris Chen. Change impact identification in object oriented software maintenance. In *ICSM '94: Proceedings of the International Conference on Software Maintenance*, pages 202–211, Washington, DC, USA, 1994. IEEE Computer Society.

[63] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr, and Erik Ruf. An Architecture for an Open Compiler. In Akinori Yonezawa and Brian C. Smith, editors, *Proceedings of the Int'l Workshop on Reflection and Meta-Level Architecture*, pages 95–106, 1992.

[64] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.

[65] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.

[66] Thomas Ledoux. OpenCorba: A Reflective Open Broker. In Pierre Cointe, editor, *Proceedings of the 2nd International Conference on Reflection'99*, LNCS 1616, pages 197–214, Saint-Malo, France, July 1999. Springer-Verlag.

[67] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[68] Meir M. Lehman and Juan F. Ramil. Software evolution: background, theory, practice. *Inf. Process. Lett.*, 88(1-2):33–44, 2003.

[69] Meir M. Lehman, Juan F. Ramil, and G Kahen. Evolution as a Noun and Evolution as a Verb. In *Proceedings of the Workshop on Software and Organisation Coevolution (SOCE 2000)*, Imperial College, London, July 2000.

[70] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[71] Orlando Loques, Alexandre Sztajnberg, Julius Leite, and Marcelo Lobosco. On the Integration of Configuration and Meta-Level Programming Approaches. In Walter Cazzola, Robert J. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, Lecture Notes in Computer Science 1826, pages 191–210. Springer-Verlag, Heidelberg, Germany, June 2000.

[72] Chris Lüer, David S. Rosenblum, and André van der Hoek. The evolution of software evolvability. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 134–137, New York, NY, USA, 2001. ACM Press.

[73] Pattie Maes. *Computational Reflection*. PhD thesis, Artificial Intelligence Laboratory, Vrije Universiteit, Brussel, Belgium, 1987.

[74] Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA, October 1987. ACM.

[75] Yukihiro Matsumoto. *Ruby In A Nutshell: A desktop Quick Reference*. O'Reilly & Associates, Inc., 2002.

[76] Michael Golm and Jügen Kleinöder. metaXa and the future of reflection. Technical report, In Proceedings of the Workshop on Reflective Programming in C++ and Java, 1998. Available on the World-Wide Web at http://www4.informatik.uni-erlangen.de/TR/pdf/TR-I4-98-09.pdf.

[77] Ivan Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *OOPSLA*, pages 235–250, 1996.

[78] Gail C. Murphy and David Notkin. Reengineering with reflexion models: A case study. *Computer*, 30(8):29–36, 1997.

[79] Gustaf Neumann, Mark Strembeck, and Uwe Zdun. Using runtime introspectible metadata to integrate requirement traces and design traces in software components. In *Proceedings of ECOOP Workshop on Unanticipated Software Evolution (USE 2002)*, pages 1–9, Malaga, Spain, 2002.

[80] OMG. Unified Modeling Language (UML) Specification version 1.4 (Draft). OMG Document ad/01-02-13, 2001.

[81] OMG. OMG-XML Metadata Interchange (XMI) Specification, v1.2. OMG Modeling and Metadata Specifications available at `http://www.omg.org`, January 2002.

[82] OMG. Unified Modeling Language 2.0 OCL Final Adopted Specification). OMG Document ptc/03-10-14, 2004.

[83] OMG. Unified Modeling Language: Diagram Interchange version 2.0). OMG Document ptc/05-06-04, June 2005.

[84] William F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, Champaign, IL, USA, 1992.

[85] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *CSC '93: Proceedings of the 1993 ACM conference on Computer science*, pages 66–73, New York, NY, USA, 1993. ACM Press.

[86] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM Press.

[87] Václav T. Rajlich and Keith H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 2000.

[88] Stephen Rank. Architectural reflection for software evolution. In Walter Cazzola, Shigeru Chiba, and Gunter Saake, editors, *RAM-SE'05, 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution Glasgow, Scotland, 25th of July 2005*, 2005.

[89] Stephen Rank, Keith Bennett, and Steven Glover. FLEXX: Designing Software for Change through Evolvable Architectures. In P. Henderson, editor, *in System Engineering for Business Process Change: Collected papers from the EPSRC research programme*, Lecture Notes in Computer Science, pages 38–50. Springer-Verlag, Heidelberg, Germany, 2000.

[90] Pierre-Guillaume Raverdy, Hubert Le Van Gong, and Rodger Lea. Dart: A reflective middleware for adaptive applications. In *in IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, 1998.

[91] Xiaoxia Ren, Barbara G. Ryder, Maximilian Störzer, and Frank Tip. Chianti: a change impact analysis tool for java programs. In *27th International Conference on Software Engineering (ICSE), 15-21 May 2005, St. Louis, Missouri, USA*, pages 664–665.

[92] Romain Robbes and Michele Lanza. Change-based software evolution. In Laurence Duchien, Maja D'Hondt, and Tom Mens, editors, *Proceedings of the International ERCIM Workshop on Software Evolution April*, pages 159–164, LIFL - INRIA, France, 2006.

[93] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, 1997.

[94] Donald Bradley Roberts. *Practical analysis for refactoring.* PhD thesis, 1999. Adviser-Ralph Johnson.

[95] Rosaldo Rossetti and Sergio Bampi. A Software Environment to Integrate Urban Traffic Simulation Tasks. In *Journal of Geographic Information and Decision Analysis*, volume 3, pages 56–63, 1999.

[96] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 169–184, New York, NY, USA, 1994. ACM Press.

[97] W. W. Royce. Managing the Development of large software systems : Concepts and techniques. In *Proceedings of WESCON*, pages 1–9, USA, 1970.

[98] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[99] Brian C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, MIT Laboratory of Computer Science, 1982.

[100] Perdita Stevens and Rob Pooley. *Using UML: Software Engineering With Objects and Components.* Pearson Education Limited, 2000.

[101] Ladan Tahvildari and Kostas Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformation. In *7th European Conference on Software Maintenance and Reengineering (CSMR 2003), 26-28 March 2003, Benevento, Italy, Proceedings*, pages 183–192. IEEE Computer Society, 2003.

[102] David Thomas and Andrew Hunt. *Programming Ruby: The Pragmatic Programmer's Guide.* Addison-Wesley, 2001.

[103] Francesco Tisato, Andrea Savigni, Walter Cazzola, and Andrea Sosio. Architectural Reflection: Realising Software Architectures via Reflective Activities. In Wolfang Emmerich and Stephan Tai, editors, *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000)*, Lecture Notes in Computer Science 1999, pages 102–115. Springer-Verlag, University of California, Davis, USA, on 2nd-3rd of November 2000.

[104] Lance Tokuda and Don S. Batory. Automating three modes of evolution for object-oriented software architectures. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies & Systems, May 3-7, The Town & Country Resort Hotel, San Diego, California, USA*, pages 189–202, 1999.

[105] Lance Tokuda and Don S. Batory. Evolving object-oriented designs with refactorings. *Autom. Softw. Eng.*, 8(1):89–120, 2001.

[106] Emiliano Tramontana. Reflective Architecture For Changing Objects. In *ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, 2000.

[107] Robert J. Walker, Gail C. Murphy, Bjørn N. Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *OOPSLA*, pages 271–283, 1998.

[108] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[109] Ian Welch and Robert J. Stroud. Kava - using byte code rewriting to add behavioural reflection to java. In *COOTS*, pages 119–130, 2001.

[110] M. Wermlinger and J. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44:133–155, 2002.

[111] Yasuhiko Yokote. The ApertOS Reflective Operating System: The Concept and Its Implementation. In Andreas Pæpcke, editor, *Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, volume 27(10) of *Sigplan Notices*, pages 414–434, Vancouver, British Columbia, Canada, October 1992. ACM.

# Curriculum Vitae

## Personal Data

**Name:** Ahmed Mohamed Ali Ghoneim

**Place and date of birth:** Born in Egypt on March 2, 1972

**Nationality:** Egyptian

**Military Service:** carried out

**Status:** married

## Educational History

**Bachelor of Science:** B. Sc. degree in Pure Mathematics and Computer Science, Menoufiya University, Egypt, May 1994.

**Master of Science:** M. Sc. degree in Computer Science, Menoufiya University, Egypt, December 1999.

**Doctor of Philosophy:** Ph.D. degree in Computer Science, Otto-von-Guericke- University Magdeburg, Germany, March 2007.

## Work and Teaching Experience

- Instructor for Computer Sciences in El-Menoufiya University - Egypt, 1996-1999.
- Assistant teacher in the department of Computers Science and Mathematics at University of El-Menoufiya, Egypt, 1999-2001.
- Research assistant in the department of Business and technical information systems at the Otto-von-Guericke- University Magdeburg, Germany. August 2001- March 2007.

## Co-operation with other scientists

I have also collaborated or I'm collaborating with:

- Prof. Dr. Walter Cazzola - University of Milano-Italy,
- Dr. Sven Apel - University of Magdeburg-Germany,
- prof. Dr. James O. Coplien - University of Manchester - England.
- Dr. Soni Pini - University of Genova-Italy.

**Publications**

- Walter Cazzola and Sonia Pini and Ahmed Ghoneim and Gunter Saake. Co-Evolving Application Code and Design Models by Exploiting Meta-Data. In Proceedings of the 12th Annual ACM Symposium on Applied Computing (SAC'07), Seoul, South Korea, on 11th-15th of March 2007.
- Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Viewpoint for Maintaining UML Models against Application Changes. In Proceedings of International Conference on Software and Data Technologies (ICSOFT 2006), Setúbal, Portugal, 11th-14th of September 2006.
- Ahmed Ghoneim and Sven Apel and Gunter Saake. Evolutionary Software Life Cycle for Self-Adapting Software Systems. In In Chin-Sheng Chen, Joaquim Filipe,Isabel Seruca, and José Cordeiro, editors, ICEIS 2005, Proceedings of the Seventh International Conference on Enterprise Information Systems,Miami, USA, May 25-28, pages 211-216, 2005.
- Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. RAMSES: a Reflective Middleware for Software Evolution. Proceedings of the 1st ecoop Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04) in 18th European Conference on Object-Oriented Programming (ecoop'04),Oslo, Norway.
- Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. System Evolution through Design Information Evolution: a Case Study. Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2004), July 1-3, Nice, France.
- Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In Hans-Dieter Ehrich, John-Jules Meyer, and Mark D. Ryan, editors, Objects, Agents and Features: Structuring Mechanisms For Contemporary Software, Lecture Notes in Computer Science, Heidelberg, Germany, 2003.
- Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Reflective Analysis and Design for Adapting Object Run-time Behavior. In Zohra Bellahsène, Dilip Patel, and Colette Rolland, editors, Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS'02), Lecture Notes in Computer Science 2425, pages 242-254, Montpellier, France, on 2nd-5th of September 2002. Springer-Verlag. ISBN: 3-540-44087-9.
- Walter Cazzola, James O. Coplien, Ahmed Ghoneim, and Gunter Saake. Framework Patterns for the Evolution of Nonstoppable Software Systems. In Pavel Hruby and Kristian Elof Søresen, editors, Proceedings of the 1st Nordic Conference on Pattern Languages of Programs (VikingPLoP'02), pages 35-54, Højstrupgård, Helsingør, Denmark, on 20th-22nd of September 2002. Microsoft Business Solutions.