Pós-Graduação em Ciência da Computação

João Henrique Correia Pimentel

SYSTEMATIC DESIGN OF ADAPTIVE SYSTEMS — CONTROL-BASED FRAMEWORK

Ph.D. Thesis

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2015

João Henrique Correia Pimentel

SYSTEMATIC DESIGN OF ADAPTIVE SYSTEMS — CONTROL-BASED FRAMEWORK

A Ph.D. Thesis presented to the Centro de Informática of Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Ciência da Computação.

Advisor: *Jaelson Freire Brelaz de Castro*

RECIFE
2015

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB4-1758

P644s Pimentel, João Henrique Correia.
Systematic design of adaptive systems: control-based framework. / João Henrique Correia Pimentel. – Recife: O Autor, 2015.
242 f.: fig., tab.

Orientador: Jaelson Freire Brelaz de Castro.
Tese (Doutorado) – Universidade Federal de Pernambuco. CIN, Ciência da Computação, 2015.
Inclui referências, glossário e apêndices.

1. Engenharia de software. 2. Sistemas de controle por realimentação. 3. Desenvolvimento de software. I. Castro, Jaelson Freire Brelaz de. (Orientador). II. Titulo.

005.1 CDD (22. ed.) UFPE-MEI 2015-123

Tese de doutorado apresentada por **João Henrique Correia Pimentel** ao programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **Systematic Design of Adaptive Systems — Control-Based Framework**, orientada pelo **Prof. Jaelson Freire Brelaz de Castro** e aprovada pela banca examinadora formada pelos professores:

Prof. Ricardo Massa Ferreira Lima
Centro de Informática/UFPE

Prof. Carla Taciana Lima Lourenço Silva Schuenemann
Centro de Informática/UFPE

Prof. Fernanda Maria Ribeiro Alencar
Departamento de Eletrônica e Sistemas/UFPE

Prof. Itana Maria de Souza Gimenes
Departamento de Informática/UEM

Prof. Julio Cesar Sampaio do Prado Leite
Departamento de Informática/PUC-Rio

Visto e permitida a impressão.
Recife, 27 de fevereiro de 2015.

Profa. Edna Natividade da Silva Barros
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco

To the generous people around the world.

Acknowledgements

Thanks to my family and friends.

Thanks to the new friends on the requirements engineering group, to the "people from the room", to the chicos from Barcelona, and to the ragazzi from Trento.

Special thanks to those that directly helped with this thesis: Emanuel Santos, the prototypical research mate; Marc and Marc, the web services guys; Vitor Souza and Konstantinos, the "hey, what about architecture?" team; Daniel, Hans, and Orivaldo, the robot guys; and Jéssyka, the statecharts partner.

Thanks to Carla Silva, Fernanda Alencar, Itana Gimenez, João Araújo, Júlio Leite, Nelson Rosa, and Ricardo Massa, members of the proposal and thesis defense committee.

Thanks to Jaelson Castro, Xavier Franch, and John Mylopoulos, for guiding and inspiring me to become a better researcher; for going beyond the call of duty. And for the free food.

Many more people helped me throughout this PhD research, both known and unknown to me, both knowingly and unknowingly. To you all, my thanks.

Resumo

Um grande número de abordagens foram propostas para elicitar, modelar e analisar requisitos para sistemas adaptativos. No entanto, ainda existe uma grande distância entre a especificação de requisitos e a implementação de um sistema adaptativo. Nesta tese foi investigada a inter-relação entre requisitos e arquitetura para o desenvolvimento de sistemas adaptativos. Mais especificamente, nós propomos o *framework* Adaptação Multi-Nível para Sistemas de *Software* (MULAS, do inglês *Multi-Level Adaptation for Software Systems*). Este framework é focado no refinamento iterativo e incremental de um modelo de objetivos, em direção à criação de um Modelo de Objetivos de Design (DGM, do inglês *Design Goal Model*). Este modelo pode então ser utilizado em tempo de execução para se gerenciar a adaptação em um sistema devidamente instrumentado. Ademais, o framework inclui um processo para gerar diagramas de estados a partir do Modelo de Objetivos de Design. Uma ferramenta desenvolvida especificamente para apoiar este framework (GATO, do inglês, *Goal TO Architecture*) permite criar os diferentes artefatos do processo, incluindo a geração automática de diagrama de estados base. A adequação desta abordagem ao desenvolvimento de sistemas adaptativos é ilustrada através de estudos de caso. Resultados empíricos mostram que as técnicas desenvolvidas para criar diagramas de estados a partir do modelo de objetivos com elementos de *design* apresentam boa escalabilidade, i.e. possui bom desempenho mesmo no caso de modelos extensos. Adicionalmente, um experimento com estudantes de engenharia de software indica que a adoção do framework por não-especialistas não é apenas possível como também é benéfica.

Palavras-chave: Adaptação de Software. Engenharia de Requisitos. Projeto Arquitetural. Engenharia Dirigida a Modelos. Sistemas Adaptativos. Transformação de Modelos. Ferramenta de modelagem.

Abstract

A number of approaches have been proposed for eliciting, modeling and analyzing requirements for adaptive systems. However, there is still a large gap between such requirements specifications and the actual implementation of adaptive systems. In this thesis we investigate the interplay between requirements and architecture for the development of adaptive systems. Furthermore, we propose the Multi-Level Adaptation for Software Systems (MULAS) framework. This framework is centered on the iterative and incremental refinement of a goal model, towards the creation of a Design Goal Model. This model can then be used at runtime to drive adaptation on a system that is properly instrumented. Moreover, the framework includes a tool-supported process for generating statechart behavioral models from a Design Goal Model. The GATO tool (Goal TO Architecture) allows the creation of the different artifacts of the process, including the automatic generation of base statecharts. The suitability of this approach for developing adaptive systems is illustrated by means of case studies. Empirical results show that the techniques developed to translate enriched goal models onto statecharts are scalable, i.e. they present a good performance even with large models. Furthermore, an experiment with software engineering students indicates that the adoption of this framework by non-experts is feasible and beneficial.

Keywords: Software Adaptation. Requirements Engineering. Architectural Design. Model-Driven Engineering. Adaptive Systems. Model Transformation. Modeling Tool.

List of Figures

List of Tables

Listings

List of Acronyms

ADL	Architecture Description Language
ATM	Automatic Teller Machine
BPMN	Business Process Model and Notation
BREAD	Browse, Read, Edit, Add, Delete
DGM	Design Goal Model
EM	Environment Monitoring
EMF	Eclipse Modeling Framework
GATO	Goal TO Architecture (tool)
GWT	Google Web Toolkit
IDE	Integrated Development Environment
JSF	JavaServer Faces
JSP	JavaServer Pages
MDE	Model-Driven Engineering
MULAS	Multi-Level Adaptation for software Systems
PIN	Personal Identification Number
R&D	Research and Development
RE	Requirements Engineering
RUP	Rational Unified Process
SVG	Scalable Vector Graphics
XML	eXtensible Markup Language
YST	Yakindu Statechart Tools

Contents

1 Introduction

In this chapter we present the context and motivation for this work, as well as its goals, the methodology used to achieve these goals, and the contributions of the thesis.

1.1 Context and Motivation

Software adaptation, which enables the creation of more flexible, resilient, robust, recoverable and energy-efficient systems (CHENG et al., 2009), can be defined as the software capability of accepting environmental changes. Software systems have been adaptive for some decades now — from operating systems that identify when a new device has been plugged in, to games where enemies learn the players' behavior and change their strategies accordingly; from traction control systems on vehicles, to websites that change their user interface to fit the screen they are being displayed on.

The relevance of adaptive systems has been growing with the increase of software complexity, of platform heterogeneity, of user bases, of reliance on software systems and of maintenance costs. Focusing on the latter, the Autonomic Computing vision was laid out in the beginning of this millennium (HORN, 2001; KEPHART; CHESS, 2003), proposing four main characteristics for systems that are able to manage themselves:

- **Self-configuration** - a system with this capability is able to automatically configure its components and sub-systems according to high-level preferences.
- **Self-optimization** - a system with this capability continually monitor its performance, with regard to different non-functional requirements (such as dependability, response time and cost), and automatically tune its parameters towards optimal performance levels.
- **Self-healing** - a system with this capability tries to prevent downtime, by identifying failures as they happen, diagnosing their causes, and enacting corrective operations.
- **Self-protection** - a system with this capability aims to improve its security, by preventing problems related with malicious attacks.

By presenting these characteristics, self-adaptive systems can bring benefits regarding both their use and their maintenance:

- **Performance** - by automatically tuning its parameters, the system may reach optimal levels of performance in its different dimensions, such as speed (response time, latency), size (memory usage, disk usage), energy consumption, scalability, and so on (FRANCH et al., 2011; CHENG; GARLAN; SCHMERL, 2009).

- **Availability** - the self-healing and self-protection characteristics can help to reduce systems' downtime, thus improving their availability (FILIERI et al., 2012).

- **Security** - by monitoring the system and its environment for malicious attack, the system may prevent some kinds of security breaches (SAVOLA; HEINONEN, 2010).

- **Maintainability** - by automating several tasks that would otherwise be performed by system administrators, self-adaptive systems can largely reduce the effort and complexity of software maintenance (HEINIS; PAUTASSO, 2008).

As a result of the increasing interest for software adaptation, there is a need for approaches that allow to systematically define, analyze and develop adaptive capabilities. Thus, the state-of-the-art evolved from ad-hoc, case-by-case development, to systematic approaches and frameworks.

Different approaches to support the development of self-adaptive systems have been proposed in the academic literature, encompassing the categories mentioned earlier. However, those are often restricted to a single aspect of software development. For instance, the Zanshin framework (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2011) provides support for handling adaptation at the requirements level, enacting a monitoring-diagnosis-compensation cycle. According to Zanshin, adaptation is specified in terms of stakeholders' goals, tasks, quality constraints, and so on. On the other hand, Rainbow (GARLAN et al., 2004) provides similar capabilities, but addressing architectural models. Thus, it is concerned with properties of systems' components and connectors, e.g., response time, number of servers and load balancing.

Requirements engineering and architectural design, while addressing the system specification at different abstraction levels, comprise intertwined activities (CASTRO et al., 2012). The former focuses on the problem at hand, whereas the latter provides solutions for that problem (NUSEIBEH, 2001). Approaches that only support requirements-based or architecture-based adaptation thus lack relevant elements of the adaptation space. For instance, architecture-based approaches might ignore stakeholders' goals and preferences, while requirements-based ones may not address concerns related to the system implementation, such as algorithms and components. Hence, the investigation on how to support seamless adaptation mechanisms across the different phases of software development is a promising venue to improve the development of self-adaptive software systems, especially in the context of Model-Driven Engineering (MDE).

Model-Driven Engineering is a development approach that focuses on models and their transformations, rather than on source-code (PASTOR; MOLINA, 2007). Starting from abstract models, developers include additional information and generate more concrete models. These concrete models can then be used to generate source-code that will be compiled and run on different platforms. Thus, instead of writing and changing source-code directly, developers manipulate the different models of the system. A key element of MDE are model transformations, defined as "the automatic generation of a target model from a source model, according to a transformation definition" (KLEPPE; WARMER; BAST, 2003).

Among the benefits of MDE approaches that can be obtained with proper tool support and training, we highlight (KLEPPE; WARMER; BAST, 2003):

- **Productivity** - the use of models to generate source-code enables developers to work in a more abstract level, thus reducing the amount of details and boilerplate to be handled. This is similar to the benefit that came with high-level programming languages (in contrast with low-level ones), where programmers can abstract from machine details and program with a more abstract language.
- **Portability** - the same models can be used to generate source-code for different target platforms, which includes not only different programming languages but also different devices.
- **Maintenance** - besides the general productivity and portability benefits that apply to maintenance, there is the additional benefit that comes from proper documentation, where with MDE the problem of low quality and obsolete documentation is absent — since the code comes from the models, the models are always up-to-date.

In this work we follow the MDE approach, focusing on models as concrete representations of a system. Moreover, we make use of techniques such as metamodeling and model transformation in order to provide an integrated framework.

1.2 Research Goal

Considering the intertwined nature of requirements and architectural design, and the relevance of each for the development of self-adaptive systems, the objective of this thesis is

to support the development of requirements-based and design-based adaptation on software-intensive systems, generating behavioral models (statecharts) from goal models, by means of a modeling language, a design process, integration mechanisms, and tool support.

Based on this goal, the following research questions were identified.

- **RQ1 - Which requirements-related adaptations are supported by current approaches for software adaptation?**
- **RQ2 - Which architecture-related adaptations are supported by current approaches for software adaptation?**
- **RQ3 - How the adaptation in these different abstraction levels can be integrated?**
- **RQ4 - How can we support software developers when moving from requirements to system behavior, for the case of adaptive systems?**

In order to answer **RQ1** and **RQ2**, we pursued the following specific goals: *investigate existing requirements-adaptation approaches; investigate existing architecture-adaptation approaches; investigate approaches that support adaptation at both levels (if any);* and *examine adaptation scenarios and examples in the software engineering literature*.

Regarding **RQ3**, we observed that the stated integration can be achieved in different ways: (i) by adopting different approaches for the different levels, and integrating them; (ii) by extending an existing approach in order to support the abstraction level that is not yet supported; (iii) by developing a new approach from scratch. Considering that the use of a single approach streamlines the development process, and the suitability of existing approaches, the second option showed to be the more appropriate. Thus, the following specific goals were defined and pursued: *select an existing software adaptation approach to be used as baseline* and *propose an integrated model that includes both requirements and architectural concerns*.

Considering the specific goal of proposing an integrated model, we considered that a good way to address **RQ4** was to pursue the following specific goal: *propose an architectural design process to guide the design of adaptive system starting from its requirements*. In order to narrow the scope of this research effort, we have decided to focus on the behavioral aspects of architectural design, instead of contemplating architectural design as a whole. Furthermore, considering the effort required to manipulate the proposed integrated model, an additional specific goal was devised: *provide tool support for the creation and edition of the proposed model*.

1.3 Methods

This research started with an exploratory study, surveying and analyzing the software engineering bibliography on the topic of adaptive systems. Through this analysis it was observed that the existing proposals for tackling the development of adaptive systems focus on particular disciplines, such as requirements engineering, architectural design, and testing. Considering the relationships and the intertwining nature of these different disciplines, the following hypothesis was formulated: *the development of adaptive systems would not benefit from the integration of adaptation at the system's different abstraction levels*.

The remainder of this research was carried on with the hypothetical deductive method. By studying requirements-based and architecture-based approaches and their examples, we identified different scenarios where a system may require both requirements-based adaptation, related to the stakeholder needs and domains entities, as well as architecture-based adaptation, related to the implemented system. Moreover, on some scenarios an adaptation triggered by a requirements failure can be solved through an architectural adaptation, and vice-versa. Thus, there is evidence that this hypothesis may be rejected.

Assuming that such integration may indeed be beneficial, we developed a framework for supporting the integration of adaptation at the requirements and architectural levels. This framework was developed incrementally: first, the incorporation of design concerns such as design tasks and design constraints; then, derivation of behavioral models (statecharts) from goal models; lastly, the integration of adaptation with both requirements and architectural concerns. Each of these partial solutions was evaluated with case studies and simulations, which were also used to further refine the proposal.

Given that the manual derivation of architectural models showed to be time-consuming, we developed a prototype tool to automate the model transformations. The performance and the scalability of these automatic transformations were evaluated through a controlled experiment. Furthermore, a qualitative experiment with a group of computer science students was performed in order to gather early evidence regarding the suitability and the quality of the proposed architectural design process.

1.4 Overview of the Proposed Solution

This thesis proposes a framework, named Multi-Level Adaptation for Software Systems (MULAS), to support the development of self-adaptive software systems. It comprises a modeling language, an architectural design process, a supporting tool and mechanisms to integrate with an existing adaptation framework.

The baseline for this work is a goal model extension (LAPOUCHNIAN, 2011; SOUZA et al., 2013) that includes concepts essential for feedback loops (ASTRöM; MURRAY, 2012), which in turn is essential for the development of adaptive systems (BRUN et al., 2009; CHENG et al., 2009; WEYNS; Usman Iftikhar; SODERLUND, 2013). These goal models, besides including traditional goal modeling elements, such as goals, tasks and quality constraints, also contain information that is essential for system adaptation: awareness requirements and parameters. *Awareness requirements* define what needs to be monitored during system execution — e.g., the desired success rate for the execution of a task, a limiting threshold for a quality constraint, and so on. On the other hand, *parameters* define what can be changed in the system, from the point of view of high-level stakeholders (e.g., clients and users). Additionally, the relations between awareness requirements and parameters are defined, expressing the impact of parameter changes onto the satisfaction of awareness requirements. These three kinds of information are used at runtime in a feedback loop that assess how well the system is performing, identify whether some change is required, select which parameter(s) to change, and effect the change(s). Currently, this goal model extension is part of the Zanshin framework (SOUZA, 2012; SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012a), which includes reasoning mechanisms that support the execution of the feedback loop.

We take the concepts of awareness requirements and parameters and apply them to architectural concerns, such as design constraints and execution flows, proposing a new kind of model called *Design Goal Model* (DGM).

The Design Goal Model is based on a requirements goal model, extended with architectural design elements, as follows:

- **Design Tasks, Design Constraints, and Design Assumptions —** These elements are similar to their requirements counterpart, but differ in that they are results of design decisions (thus in the solution space), rather than mandated requirements (from the problem domain).

- **Assignments —** These elements define responsibilities for the execution of tasks .

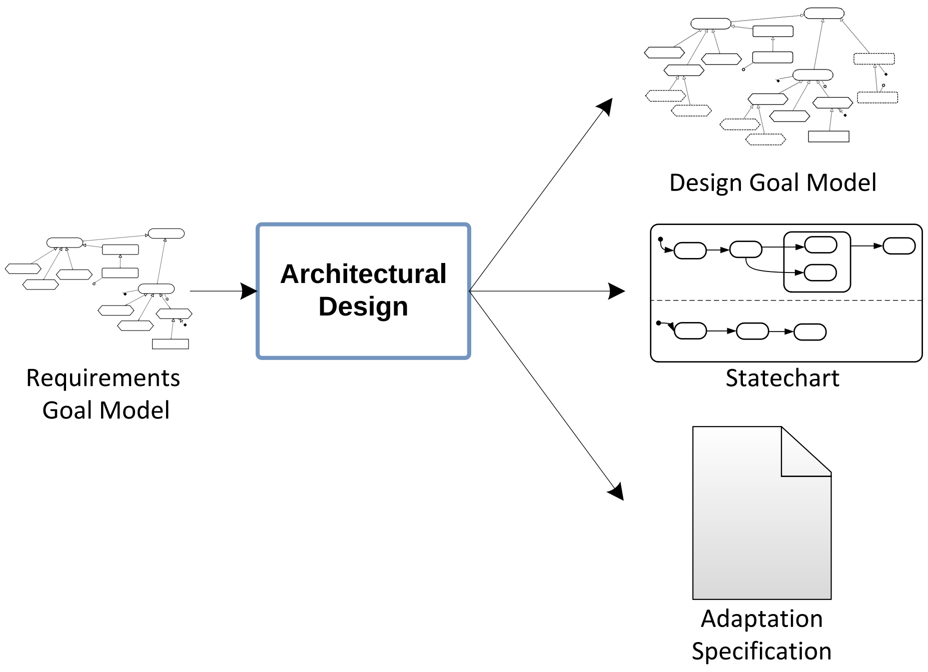
- **Behavioral annotations —** Define possible execution flows for the different elements of the model.

The DGM has two roles. The first role is as an adaptation artifact — by using an extended goal model, we are able to *reuse the reasoning mechanisms* from the Zanshin framework (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012a). As a result, the users of the framework do not need to implement or handle additional adaptation components, other than those already provided by Zanshin. The second role is as an *input for model transformations* — rather than creating the different requirements and behavioral models, and then creating the Design Goal Model, we propose a set of transformation rules where behavioral models can be automatically derived from design goal models. Different models can be adopted to represent the behavioral view of a system architecture, such as sequence diagrams, use-case maps, and statecharts (BACHMANN et al., 2002) — due to the reasons described in Section 2.3, we adopted the statecharts notation

Aiming to provide methodological guidance for software architects, we have proposed an architectural design process. Along with automatic transformation rules and a prototype tool, it can reduce the effort required to go from a requirements goal model to a complete, implemented solution, by supporting the creation of the following artifacts: design goal model, statechart, and adaptation specification (Fig. 1.1). The adaptation specification contains information about which adaptations must be performed by the system, about the possible values assumed by parameters, as well as the relation between awareness requirements and parameters.

The transformation rules defined in this research project enable the automatic generation of statecharts from design goal models. Thus, the proposed architectural design process fits into

Figure 1.1: Overview of the architectural design process proposed in this thesis



the overall software development process, handling the creation of the specification of the system behavior (statecharts). Moreover, the statechart by itself can be used to execute simulations that provide early validation of the system-to-be (EGYED; WILE, 2001a; LIAN; HU; SHATZ, 2008), as well as to generate source-code (NIAZ; TANAKA, 2003; TIELLA; VILLAFIORITA; TOMASI, 2007) and to perform architectural analysis (DIAS; VIEIRA, 2000).

A prototype tool (GATO - Goal to Arch) provides a friendly interface to create DGM diagrams and to execute the model transformations. Furthermore, by implementing different views for the different facets of the DGM, such as requirements and design, it prevents the complexity that would arise if handling all these facets at the same time, in a single view.

In the following section we present the research methods that guided this research project.

1.5 Published work

In this section we list papers related to this thesis that were published in international peer-reviewed venues.

- **Pimentel, J. , Castro, J., and Franch, X. (2011). Specification of Failure-Handling Requirements as Policy Rules on Self-Adaptive Systems. In 14th Workshop on Requirements Engineering —** This work describes a policy language that allows to specify failure tolerance rules in terms of goal models, similar to awareness requirements.

- **Pimentel, J., Castro, J., Perrelli, H., Santos, E., and Franch, X. (2011). Towards anticipating requirements changes through studies of the future. In 5th International Conference on Research Challenges in Information Science —** In this paper we investigate the elicitation of adaptation, considering future events that may impact the requirements of a software system (selected as one of the best papers).

- **Pimentel, J., Santos, E., Castro, J., and Franch, X. (2012). Anticipating Requirements Changes-Using Futurology in Requirements Elicitation. International Journal of Information System Modeling and Design —** Expanding on the previous work, this paper proposes a process for adapting goal models based on future events.

- **Pimentel, J. , Franch, X., and Castro, J. (2011). Measuring Architectural Adaptability in i * Models. In XIV Ibero-American Conference on Software Engineering —** While in previous work we focused on requirements, in this paper we investigate software adaptation related to architectural goal models (selected as one of the 3 best papers).

- **Franch, X., Grunbacher, P., Oriol, M., Burgstaller, B., Dhungana, D., Lopez, L., Marco, J., and Pimentel, J. (2011). Goal-Driven Adaptation of Service-Based Systems from Runtime Monitoring Data. In 35th Annual International Computer Software and Applications Conference Workshops —** This work presents a framework for service-based adaptation, based on requirements goal models.

- **Pimentel, J., Lucena, M., Castro, J., Silva, C., Santos, E., and Alencar, F. (2012). Deriving software architectural models from requirements models for adaptive systems: the STREAM-A approach. In Requirements Engineering Journal —** This is an initial version of the architectural design process described in this thesis, adopting different baselines.

- **Castro, J., Pimentel, J., Lucena, M., Santos, E., and Dermeval, D. (2011). F-STREAM: A Flexible Process for Deriving Architectures from Requirements Models. In Advanced Information Systems Engineering Workshops —** Building on the previous work, this paper proposes a generic architectural design process that can be instantiated not only for handling adaptation but also for other non-functional requirements.

- **Pimentel, J., Santos, E., Dermeval, D., Castro, J., and Finkelstein, A. (2012). Towards Architectural Evolution through Model Transformations. In 24th International Conference on Software Engineering and Knowledge Engineering —** The subject of this paper is the use of model transformations for the automatic adaptation of architectural models.

- **Pimentel, J., Castro, J., Santos, E., and Finkelstein, A. (2012). Towards Requirements and Architecture Co-evolution. In Advanced Information Systems Engineering Workshops —** In this paper we investigate the problem of maintaining two different models in sync during a software's life cycle: requirements models and architectural models.

- **Angelopoulos, K., Souza, V. E. S., and Pimentel, J. (2013). Requirements and Architectural Approaches to Adaptive Software Systems: A Comparative Study. In 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems —** This paper analyzes requirements-based and architectural-based adaptation approaches, arguing in favor of an integrated approach.

- **Pimentel, J., Angelopoulos, K., Souza, V. E. S., Mylopoulos, J., Castro, J. (2013). From Requirements to Architectures for Better Adaptive Software Systems. In 6th International i* Workshop 2013 —** This paper presents an overview of the MULAS framework.

- **Horkoff, J., Li, T., Li, F.-l., Pimentel, J., Salnitri, M., Cardoso, E., Giorgini, P., and Mylopoulos, J. (2014). Taking Goal Models Downstream: A Systematic Roadmap. In 8th International Conference on Research Challenges in Information Science —** A systematic mapping on the topic of deriving different models based on goal models is presented in this paper (best paper award).

- **Pimentel, J., Castro, J., Mylopoulos, J., Angelopoulos, K., and Souza, V. E. S. (2014). From Requirements to Statecharts via Design Refinement. In 29th Annual ACM Symposium on Applied Computing —** This work describes the derivation of statecharts from goal models.

- **Vilela, J., Castro, J., Pimentel, J., Soares, M., and Cavalcanti, P. (2015). Deriving the behavior of context-sensitive systems from contextual goal models. In 30th Annual ACM Symposium on Applied Computing —** In this work the inclusion of contextual elements in the statechart derivation is discussed.

- **Vilela, J., Castro, J., Pimentel, J., and Lima, P. (2015). On the behaviour of context-sensitive systems. In 18th Ibero-American Conference on Software Engineering —** This work describes a process to derive statecharts from contextual goal models.

Additionally, eleven other papers were authored and co-authored during this Ph.D. They were published in peer-reviewed international venues but are not strongly related to the conception of the MULAS framework:

- **Xavier, L., Alencar, F., Castro, J., Pimentel, J. (2010). Integração de Requisitos Não-Funcionais a Processos de Negócio: Integrando BPMN e NFR. In 13th Workshop on Requirements Engineering.**

- **Santos, E., Pimentel, J., Castro, J., Sánchez, J., Pastor, O. (2010). Configuring the Variability of Business Process Models Using Non-Functional Requirements. In Enterprise, Business-Process and Information Systems Modeling, Lecture Notes in Business Information Processing (LNBIP).**

- **Pimentel, J., Santos, E., Santos, B., Borba, C., Paes, J., Lima, C., Bezerra, A., Castro, J., Alencar, F., Silva, C., Ramos, R. A., Lucena, M. (2010). Using i* and Tropos in a Software Engineering Contest: Lessons Learnt and Some Key Challenges. In 4th International i* Workshop.**

- **Dermeval, D., Soares, M., Alencar, F., Santos, E., Pimentel, J., Castro, J., Lucena, M, Silva, C., Souza, C. (2011). Towards an i*-based Architecture Derivation Approach. In 5th International i* Workshop.**

- **Santos, E., Pimentel, J., Dermeval, D., Castro, J., Pastor, O. (2011). Using NFR and Context to Deal with Adaptability in Business Process Models. In 2nd International Workshop on Requirements@Run.Time.**

- **Castro, J., Lucena, M., Silva, C., Alencar, F., Santos, E., Pimentel, J. (2011). Changing Attitudes Towards the Generation of Architectural Models. In Journal of Systems and Software.**

- **Santos, E., Pimentel, J., Castro, J., Pastor, O. (2012). On the Dynamic Configuration of Business Process Models. In Enterprise, Business-Process and Information Systems Modeling, Lecture Notes in Business Information Processing (LNBIP).**

- **Dermeval, D., Pimentel, J., Silva, C., Castro, J., Santos, E., Guedes, G., Lucena, M, Filkenstein, A. (2012). STREAM-ADD – Supporting the Documentation of Architectural Design Decisions in an Architecture Derivation Process. In 36th Annual International Computer Software and Applications Conference.**

- **Soares, M., Pimentel, J., Castro, J., Silva, C., Talitha, C., Guedes, G., Dermeval, D. (2012). Automatic Generation of Architectural Models From Goals Models. In 24th International Conference on Software Engineering and Knowledge Engineering.**

- **Dermeval, D. Castro, J., Silva, C., Pimentel, J., Bittencourt, I., Brito, P., Elias, E., Tenório, T., Pedro, A. (2013). On the Use of Metamodeling for Relating**

Requirements and Architectural Design Decisions. In 28th Annual ACM Symposium on Applied Computing.

- **Oliveira, K., Pimentel, J., Santos, E., Dermeval, D., Guedes, G., Souza, C., Soares, M., Castro, J., Alencar, F., Silva, C. (2013). 25 years of Requirements Engineering in Brazil: a systematic mapping. In 16th Requirements Engineering Workshop.**

1.6 Thesis Outline

The remainder of this thesis is organized as follows.

- **Chapter 2 -** Introduces the modeling languages and main concepts used throughout this work;
- **Chapter 3 -** This chapter characterizes the adaptation concerns supported by this approach, both at requirements and the architectural level, and then describes the Design Goal Model (DGM);
- **Chapter 4 -** The MULAS Architectural Design process to move from requirements to architecture, supporting multi level adaptation, is presented in this chapter;
- **Chapter 5 -** This chapter presents the prototype tool developed to support the use of this approach, as well as algorithms for automatic derivation of statecharts;
- **Chapter 6 -** Two case studies described in this chapter not only illustrate the use of the MULAS framework on different domains (banking and robotics), but also provides early validation of its feasibility and suitability;
- **Chapter 7 -** The evaluation of this approach through experiments is described in this chapter;
- **Chapter 8 -** Presents final remarks and the next steps of this research.

2 Baseline

In this chapter we provide an overview of the baseline of this work, focusing on the modeling languages used throughout this thesis: Goal Models with Zanshin, which is a goal modeling language that includes adaptation elements; and Statechart, a language often used to document the behavioral view of systems. Furthermore, the main concepts of requirements engineering, goal modeling, flow expressions and control theory are also presented in this chapter.

2.1 Requirements Engineering

Different methodologies available in the requirements engineering literature can be used to assist the creation of requirements models for software systems: for instance, PRiM provides a series of steps to create an *i** model (GRAU; FRANCH; MAIDEN, 2008) based on existing business processes. The approach presented in (NETO; GOMES; CASTRO, 2005) proposes a set of guidelines for the creation of *i** models using concepts from Activity Theory. In fact, through a systematic mapping on goal models derivation, we identified 24 works that map different kinds of models onto goal models (HORKOFF et al., 2014).

Additionally, more traditional requirements engineering processes could be observed for the creation and the refinement of requirements goal models. For instance, the process from (SOMMERVILLE, 2004) contains 4 activities, besides requirements management: requirements elicitation, requirements analysis and negotiation, requirements documentation, and requirements validation. A different, albeit similar set of activities, is proposed in (NUSEIBEH; EASTERBROOK, 2000): requirements elicitation, requirements modeling and analysis, requirements communication, requirements agreement and requirements evolution.

Requirements elicitation is an information gathering activity, where requirements engineers try to understand what the stakeholders expect from the system to be. This kind of information is subjective and often hard to obtain, possibly requiring the use of a mix of elicitation techniques, such as the different modalities of interviews, group workshops, observation, protocol analysis, scenario-based and prototype-based analysis (SUTCLIFFE; SAWYER, 2013).

It is then necessary to analyze the requirements identified through requirements elicitation in order to prevent problems related to conflicts, consistency, completeness, and others. Some questions that can guide the execution of this activity are: "Is this requirement really needed?", "Is there any conflict between different requirements?", "Is there anything missing?", "Is this requirement feasible for this project?", "Are these requirements actually the same thing?", and so on. Negotiation may then follow to solve the doubts and issues raised throughout the requirements analysis.

Sometimes it is important to document the system requirements in some form of requirements artifact, specially in the case of projects that span for a long time period, as well as systems developed by a large team. Requirements are usually documented using natural language, diagrams, formal languages or a mix thereof. When creating this document, it is relevant to consider its different users: clients may use it to check whether it represents what they expect from the system to be; architects may use it to guide the architectural design; developers may use it to understand how a functionality is supposed to behave; testers may use it to define how to test the developed system against those requirements; and so on.

Requirements validation is concerned with deciding if the documented requirements correctly represented the actual requirements, checking the document with respect to validity, consistency, completeness, feasibility, conflicts, technical errors, compliance to standards, ambiguity and verifiability. Some techniques that can be used to this end are requirements reviews, prototyping, test-case generation and model paraphrasing (KOTONYA; SOMMERVILLE, 1998).

These activities can be used both for the creation of a requirements document from scratch and for the modification of an existing document, and also apply to the creation of goal models.

In this thesis we are assuming that requirements are those concerns explicitly expressed by stakeholders of the system (such as costumers and users), which they expect to be addressed by a system. As discussed in (JURETA; MYLOPOULOS; FAULKNER, 2008):

> "Utterances that stakeholders make in communicating with the engineer are actions intended to advance stakeholders' personal desires, intentions, beliefs, and attitudes, in the aim of ensuring that the engineer can produce a specification that then leads to the system responsive to the communicated concerns."

In the following subsection we describe a specific kind of requirements notation, which captures the concerns expressed by stakeholders: goal models

2.2 Goal Models

Different approaches can be used for requirements documentation, such as natural language text (ROBERTSON; ROBERTSON, 2012), use cases (JACOBSON; SPENCE; BITTNER, 2011), goal models (LAMSWEERDE, 2001) and formal methods (GREENSPAN; MY-

LOPOULOS; BORGIDA, 1994). Most of these approaches focuses on *what* and *how* needs to be done (YU; MYLOPOULOS, 1994). Goal modeling goes a step further and include the *why* dimension, explicitly representing the space of alternatives for fulfilling stakeholders' needs. As stated in (YU; MYLOPOULOS, 1998), *"Goals lead to the exploration and consideration of alternatives, decision spaces, tradeoffs, and decisions. Very importantly, it allows the expression of freedom within such spaces."* These characteristics are beneficial for the design of adaptive systems (LAPOUCHNIAN et al., 2006). In fact, different goal-based adaptation approaches have been proposed in the software engineering community (MORANDINI; PENSERINI; PERINI, 2008; DALPIAZ; GIORGINI; MYLOPOULOS, 2009; QURESHI et al., 2010; SOUZA et al., 2013).

In goal models, requirements are represented as goals, tasks, domain assumptions (DAs), and quality constraints (QCs). These elements are supported by many requirements modeling approaches (LAMSWEERDE, 2001). Goals represent the needs and desires of stakeholders — often, customers and users. The space of alternatives for goal satisfaction is represented by Boolean AND/OR refinements. If elements e_1, e_2 are AND-refinements of a goal g, then both e_1 and e_2 are required. If, instead, e_1, e_2 are OR-refinements of a goal g, then the implementation of either e_1 or e_2 is sufficient.

Tasks, on the other hand, are directly mapped to functionality in the running system and are satisfied if executed successfully. Notwithstanding, quality constraints (QCs) can define additional restrictions for the satisfaction of goals as well as for the execution of tasks. Another important goal modeling concept is that of softgoals, which express goals for which there is no clear-cut achievement criteria. However, we are assuming here that the softgoals that arise in early requirements were already refined onto goals or QCs. Finally, Domain Assumptions (DAs) are satisfied if they hold (i.e., if they are true) while the system is pursuing their parent goal.

Formally, a goal model GM can be defined as a tuple $\langle E, R \rangle$, where E is a set of elements (goals, tasks, domain assumptions and quality constraints) and R is a set of relationships between these elements. Given relationships (e_1, e), ... , (e_n, e) in R, the elements $e_1 ... e_n$ are called children, descendants, or even sub-elements of e, whereas e is called the parent of $e_1 ... e_n$.

The definitions for these goal modeling concepts can be found in the requirements engineering ontology presented in JURETA; MYLOPOULOS; FAULKNER (2008), and are summarized next:

- **Domain Assumption —** *Believed content communicated by way of assertive, declarative, or representative declarative speech acts.*
- **Quality Constraint —** *Desired content communicated by way of a directive speech act is a quality constraint if and only if the content describes qualities and constrains quality values. Described qualities must have quality space with a well-defined and shared structure.*

- **Goal —** *Desired content communicated by way of a directive speech act is a goal if and only if the content neither describes qualities nor constrains quality values.*

- **Task —** *Intended content communicated by way of a commissive speech act is a task*[1].

Fig. 2.1 shows a goal model with partial requirements for the running example that will be used throughout this thesis: a Meeting Scheduler system (PIMENTEL et al., 2014). This kind of system has been well documented by the Requirements Engineering community (LAMSWEERDE; DARIMONT; MASSONET, 1995; FEATHER et al., 1997; YU, 1997; SANTANDER; CASTRO, 2002), with some works particularly focusing on analyzing adaptation scenarios based on its requirements (FEATHER et al., 1998; SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2011).

The main goal of this Meeting Scheduler system, as illustrated in Fig. 2.1, is to *Schedule Meeting*. To be successful, the system is supposed to satisfy the *Characterize Meeting*, *Collect Timetables*, *Define Schedule* and *Manage Meeting* sub-goals, while also satisfying the *Portability* constraint of being accessible through PC and smartphones (AND-refinement). With *Characterize Meeting*, a user can provide basic information about the meeting to be scheduled, through the following tasks: *Define Topics* to be discussed, *Define Date Range*, and *Define Participants*. Furthermore, this characterization must be doable within a time limit of 5 minutes, due to usability concerns. The gathering of timetables from meeting participants can be achieved by three alternative means: *Collect by Phone*, *Collect by Email*, and *Collect Automatically* — the latter assumes that the calendars of the meeting participants are updated. The actual scheduling can be achieved manually (*Schedule Manually*), by some user of the system, or automatically (*Schedule Automatically*) by the system itself. Both options must be supported by the system, thus the AND-refinement. Lastly, *Manage Meeting* concerns both *Cancel Meeting* and *Confirm Occurrence*.

A key characteristic of this kind of goal model is that it captures a space of alternatives, by means of OR refinements. Thus, instead of documenting a single way of achieving a goal, the modeler can include different alternatives identified during requirements elicitation. The most suitable(s) alternative(s) can be later selected according to, for instance, how well they perform with respect to quality constraints.

Albeit rich, goal models are not able to express relevant information such as pre-conditions, triggers, and order of execution. This information is better represented through behavioral models, as described in the following section.

[1]The original ontology refers to *plans* instead of *tasks*. We adopt the name of *task*, as other approaches, to differentiate from the concept of plans used in Artificial Intelligence.

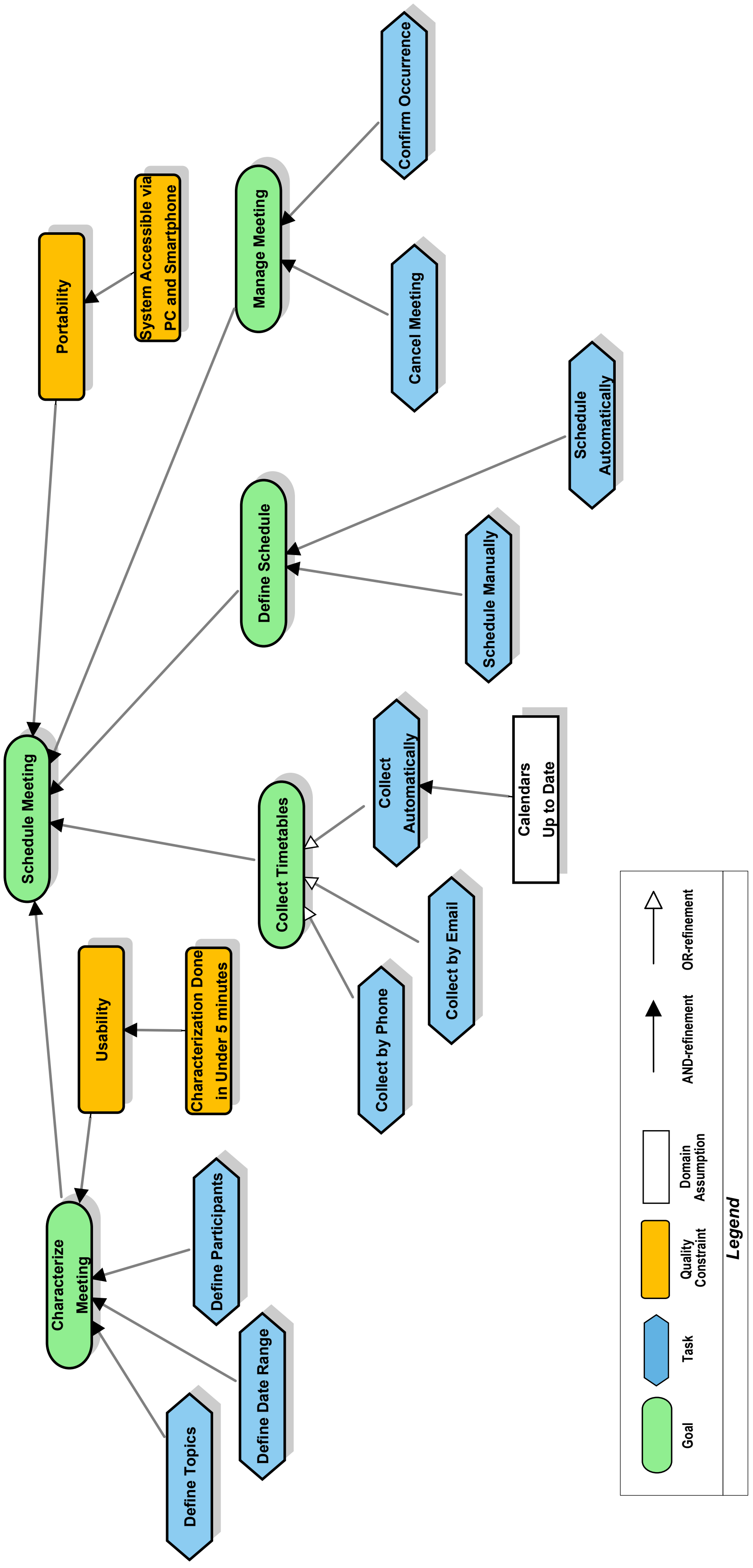
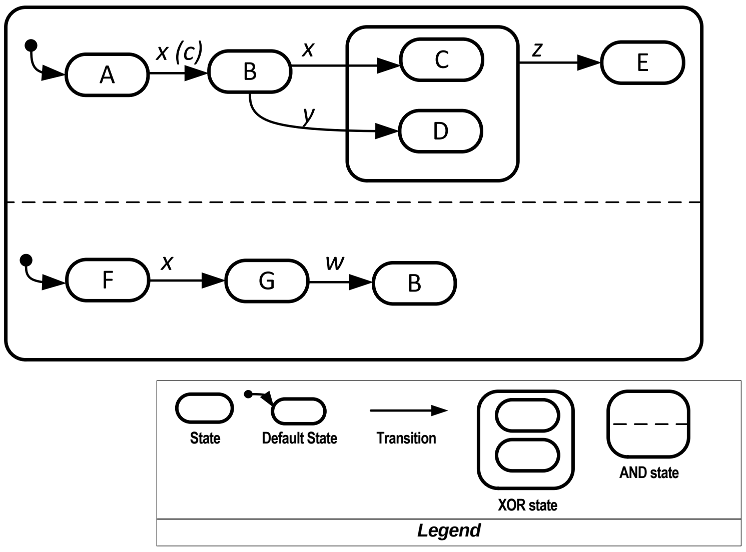


Figure 2.1: Partial Requirements Model for a Meeting Scheduler system

Figure 2.2: Statechart example



2.3 Statecharts

Statecharts (HAREL, 1987) is a visual formalism that can be used to specify system behavior in the context of architectural design (BACHMANN et al., 2002). The main elements of this modeling language are states that our system-to-be can be in, and transitions that represent possible state changes. A transition has an associated event that triggers the transition, and a condition that must be true for the transition to occur. Thus, by specifying a graph of states connected by transitions it is possible to specify how the system-to-be reacts to different events, depending on its current state.

Unlike their finite state machine cousins (FERRENTINO; MILLS, 1977), statecharts allow to structure states in a hierarchy of super- and sub-states. In an XOR state, if the state is active, so is one and only one of its sub-states; in an AND state, if a state is active so are all of its sub-states. The AND state (concurrent) is represented with a dashed line separating its sub-states.

Fig. 2.2 shows a statechart where the system starts in both states A and F, concurrently. If an event x occurs, the system transitions from F to G. Additionally, if condition c holds, there will also be a transition from A to B. If the event x happens when the system is in B, it will transition to C. Similarly, event y triggers a transition from B to D, while event w triggers a transition from G to B. Lastly, if the system is in C or D and event z occurs, the system transitions to E.

An excerpt of the statechart for the Meeting Scheduler system is presented in Fig. 2.3. In this excerpt, the system starts in an *Idle* state. If the user request to *Collect by Phone*, the system will transition to the *Input Participants Availability* state, on which the user will be able to input the information that was obtained through phone calls. Once this input is completed, the system will return to the *Idle* state. Alternatively, if the user requests to *Collect by Email*, the system will transition to the *Request Timetables by Email* state. Once completed, the system returns to *Idle*. Lastly, if the user requests to *Collect Automatically*, the system will *Collect from Google Calendar* and *Check Calendar Update Date*, before returning to *Idle*.

Idle states are very common in statechart models. They represent intermediate states, points where the system is just waiting for some input. E.g., waiting for a selection by the user, or waiting for some message to be received. These *idle* states appear on systems of different application domains, such as : cruise control (Pettit Iv; GOMAA, 2001), gas station (LIAN; HU; SHATZ, 2008), Automatic Teller Machine – ATM (BALSER et al., 2004), mobile phones (BACHMANN et al., 2002), streaming server (EGYED; WILE, 2001b), communication protocol (MAHONEY; ELRAD, 2005).

Statecharts can be formalized as triples $\langle S, T, R \rangle$, where S is a set of states, T is a set of transitions between states ($T \subseteq S \times S$), and R is a set of state refinements that defines the XOR and AND states ($R \subset S \times S$).

The modularization mechanisms provided by statecharts make it a suitable notation for iterative design, allowing the representation of system behavior at different granularities. Different formalization techniques can be used to perform automatic reasoning with its models (LEVI, 1997), and tool support provide simulation and model-driven capabilities. Lastly, Statecharts are well known within the software engineering community, specially since after its adoption as part of UML. Given these characteristics, we decided to adopt Statecharts for the representation of system behavior in the MULAS framework.

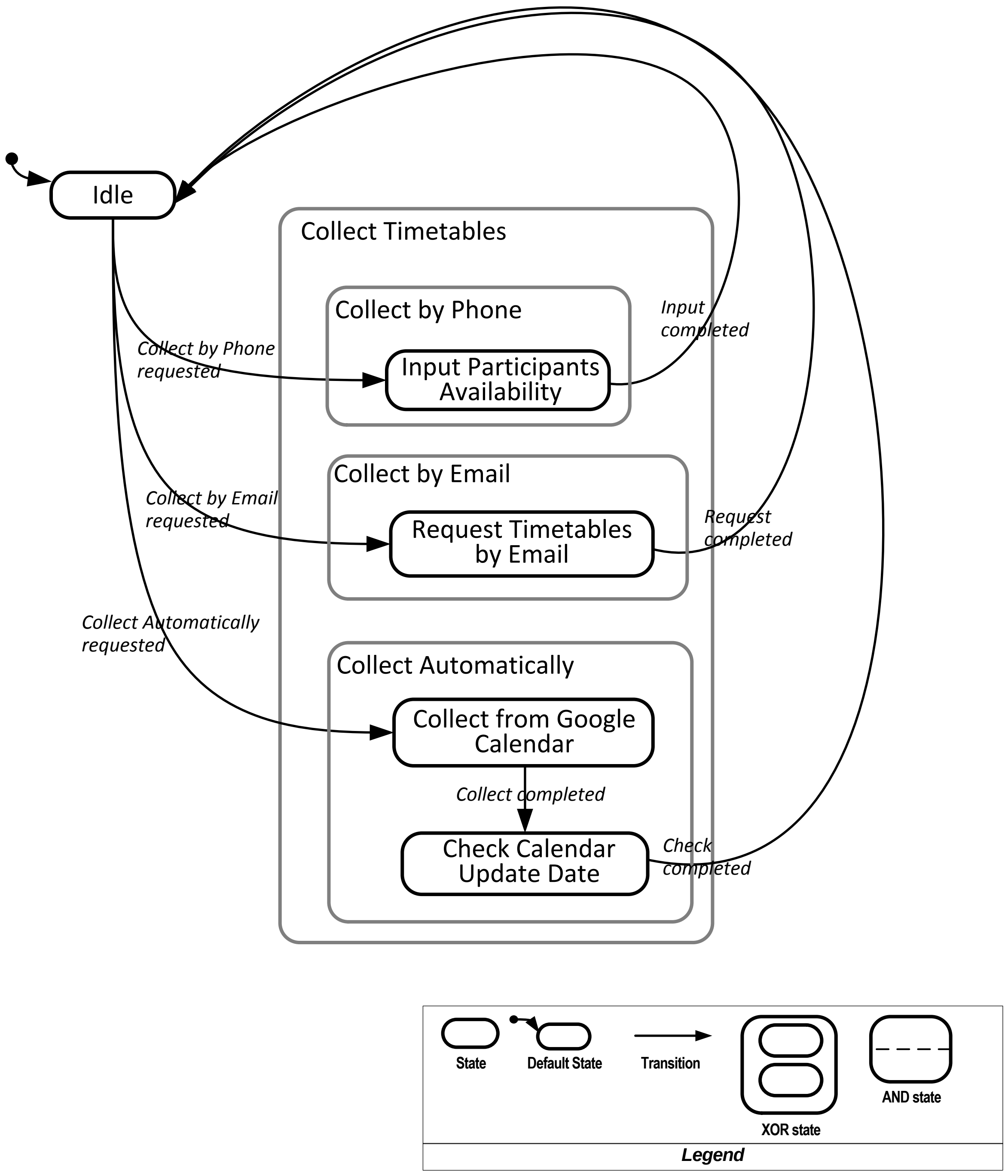
In the next section we present a textual notation that can also be used to express system behavior, which is adopted in MULAS to annotate goal models with behavioral information.

2.4 Flow Expressions

When investigating how to design system behavior having goal models as the starting point we have tried different approaches, such as complementary tables and a modified visual syntax. The option we identified as being the best, with respect to expressiveness and usability, is the use of a notation akin to regular expressions included in the goal model. After surveying the software and systems engineering literature, we identified the proposal of flow expressions (SHAW, 1978), which we adopted in the MULAS framework.

Flow expressions (SHAW, 1978; DALPIAZ et al., 2013) describe the flow of system behavior in terms of extended regular expressions. In our use of such expressions we adopted different symbols in order to facilitate their writing (e.g., | in place of $\cup$ for expressing alternative behaviors). Each atomic symbol represents an element of the flow, which in our case is an element from a goal model. For example, if $g1$ is a goal, the atomic expression `g1` represents the state where the goal $g1$ has been fulfilled. Flow expressions can be composed in terms of

Figure 2.3: Partial statechart for the Meeting Scheduler system


Idle
Collect Timetables
Collect by Phone
Collect by Phone requested
Input Participants Availability
Input completed
Collect by Email
Collect by Email requested
Request Timetables by Email
Request completed
Collect Automatically
Collect Automatically requested
Collect from Google Calendar
Collect completed
Check Calendar Update Date
Check completed
State
Default State
Transition
XOR state
AND state
Legend

regular expression operators, such as concatenation (`g1 g2`), meaning "first satisfy $g1$ then $g2$" (sequential flow), or `g1*`, meaning that $g1$ is to be satisfied zero or more times.

Flow expressions separated with a vertical bar | symbol represent alternative flows. The question mark ? is used to represent the optionality of the flow to its left, i.e., that flow may be executed zero or one times. The star symbol * indicates that the flow to its left may be executed zero, one or more times, while the plus symbol + indicates that that flow may be executed one or more times. The shuffle operator (here expressed as the hyphen symbol –) indicates that two flows are to be carried out concurrently, in the sense that their states can be interleaved. Considering the letters from A to H as atomic flows, the flow expression

$$\texttt{(A B (C | D) E F* G) - (H*)}$$

indicates that state A is followed by state B. After B, the possible states are C or (exclusively) D, followed by E. After E, F may be reached any number of times, including zero. State G occurs after E or after F. Concurrently to all that, the state H may occur any number of times.

Fig. 2.4 illustrates the use of flow expressions in conjunction with goal models, using an excerpt of the Meeting Schedule system. Starting from the annotation on its root goal (*Schedule Meeting*), the system execution begins in an idle state, and then move to g2, which is *Collect Timetables*. Moreover, this flow may be repeated, as denoted by the star symbol (*). The execution of *Collect Timetables* is a selection between *Collect by Phone* (g3), *Collect by Email* (g4), or *Collect Automatically* (g5). The execution of *Collect by Phone* simply consists on the execution of *Input Participants Availability* (t6). Similarly, the execution of *Collect by Email* consists on the execution of *Request Timetables by Email* (t7). Lastly, the execution of *Collect Automatically* consists on executing *Collect from Google Calendar* (t8), and only then executing *Check Calendar Update Date* (t9). This execution flow corresponds to the behavior expressed on the statechart at Fig. 2.3

In this and in the previous sections we summarized some notations that are broadly used in system engineering and adopted in the MULAS framework: goal models, statecharts, and

Figure 2.4: Example of a goal model annotated with flow expressions

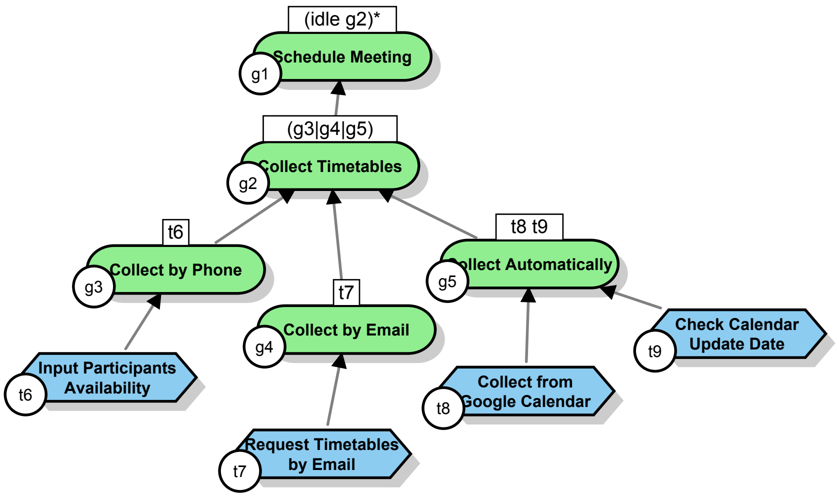
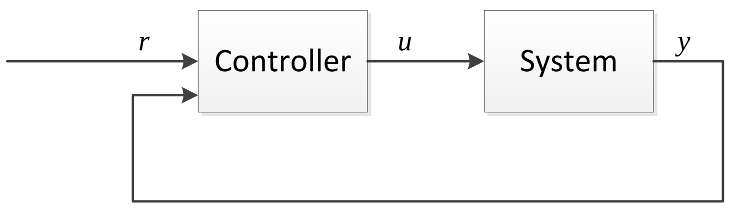


Figure 2.5: Block diagram of a simplified feedback loop, based on (HELLERSTEIN et al., 2004)



flow expressions. However, the development of adaptive systems presents some peculiarities which lead to the adoption of control theory, in general, and Zanshin, in particular. These topics are discussed in the following sections.

2.5 Control Theory

Control theory is a field of engineering and mathematics concerned with adjusting systems so that their output can be maintained as close as possible to the desired output, with applications ranging from robotics to economics. An example of a controlled system is that of a simple thermostat-controlled fridge: when the temperature inside the fridge rises above a given reference temperature, its compressor is turned on; when the temperature gets below the reference value, its compressor is switched off. This kind of mechanism is called an *on-off control* (ASTRöM; MURRAY, 2012). Despite being a simplistic mechanism, it illustrates the *feedback loop*, which is a central concept in control theory, comprised of sensing, computation and actuation. In the fridge example, temperature is monitored by means of a sensor, some computation is performed to check if the actual temperature meets the desired one (reference value) and, if necessary, actuators enact the required changes (turn the compressor on or off).

Fig. 2.5 shows a simplified data flow of the feedback loop. The reference value r, which represents the desired output of the System, is an input for the Controller. Based on that input and on the actual output y of the System that is fed back to the Controller, a command u (also called control input) is sent to the System.

In order to define what command to provide to the System, the Controller needs a model of the system, which is created in the activity of *system identification*. These models define what are the system's parameters and how they affect the system's output. By defining the relation between parameters and output, a controller could identify which parameters need to be changed in order to produce a desired output. The relation between parameters and outputs are usually defined through first-order differential equations or as difference equations (ASTRöM; MURRAY, 2012). The former is used for continuous domains and the latter for discrete domains. In both cases, the equation gives us the rate of change of the system according to the control input.

The use of feedback loops for managing adaptive systems has been advocated by a number of authors, both in the academy (KEPHART; CHESS, 2003; ABDELWAHED; KANDASAMY, 2007; HEINIS; PAUTASSO, 2008; BRUN et al., 2009; WEYNS; Usman Iftikhar; SODERLUND, 2013) and in the industry (HE et al., 2012; ZHANG et al., 2012; GHANBARI et al., 2012). In the following subsection we are going to present an approach for enacting feedback loops based on software system's requirements.

2.6 Goal Model with Adaptation Elements

LAPOUCHNIAN (2011) proposed the application of control theoretic principles for managing adaptive software systems based on goal-oriented requirements engineering.

In recent years, this goal model extension has evolved into an adaptation framework that comprises not only the modeling language, but also patterns for specifying adaptation strategies (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012b), reasoning mechanisms that diagnose failures in the target system (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012a), as well as a prototype implementation that supports the execution of the feedback loop (TALLABACI; SOUZA, 2013). That framework, named Zanshin (SOUZA, 2012), is explained in the next section, while in this sections we focus on the goal model extension.

The choice of this goal model extension, as well as the choice of Zanshin, was based on technical, social, and pragmatic factors, such as: it considers the goal model structure (AND-OR- refinements) when performing the adaptation cycle; it supports control-theoretic concepts; it performs qualitative reasoning on incomplete and imprecise knowledge, which is relevant in the context of a highly abstract and subjective domain (requirements and architecture); it is well accepted within the software engineering community; it is an ongoing work, still evolving; it is well documented; it has a prototype implementation of its reasoning algorithms, which facilitates the enactment of the feedback loop.

In order to better support the development of adaptive systems, LAPOUCHNIAN (2011) proposes two additional concepts for goal models: *awareness requirements* and *parameters*. The former are requirements about the state of other requirements — such as their success or failure — at runtime. Therefore, awareness requirements express the *reference value* of a feedback loop. In other words, awareness requirements define indicators for requirements convergence at runtime. As a side effect they also indicate how critical each requirement is, by specifying the degree of failure that can be tolerated.

Awareness requirements may refer to any element of the goal model: goals, quality constraints, tasks, and domain assumptions. In our example, three awareness requirements were defined (Fig. 2.6), represented as small circles. For instance, the awareness requirement with the identifier *AR1* defines that the quality constraint *Characterization Done in Under 5 Minutes* (to which it is attached) must have a success rate of at least 90% (*SuccessRate(90%)*). The second awareness requirement, *AR2*, defines that the *Collect Timetables* goal must present a stable or positive trend at seven days' intervals, with a tolerance of two occurrences (*NotTrend-*

Decrease(7d,2)). Lastly, *AR3* defines that the *Define Schedule* goal cannot ever fail (*NeverFail*).

Parameters are variables that affect the fulfillment of awareness requirements. That is, it represents the *control input* related to that awareness requirement. For example, consider the parameter that specifies the percentage of meeting participants that we need to collect timetables from before generating a schedule (*FhM – From how Many*, from Fig. 2.6). It affects the fulfillment of the root *Schedule meeting* requirement: if we need to collect from all meeting participants (*FhM* = 100%) chances of failure are high, as some participants may be unavailable. Chances of success for the root goal improve, however, as the value of *FhM* is lowered.

Fig. 2.6 shows two additional parameters of the Meeting Scheduler system: *VPA – View Private Appointments*, related to the *Collect Automatically* task; and *MCA – Maximum number of Conflicts Allowed* when scheduling a meeting, related to the *Schedule Automatically* task. The former defines whether the system will be able to obtain information about private appointments on users' calendars, or only public ones. The latter, *MCA*, defines how many scheduling conflicts are allowed for a given meeting, where the number of conflicts is the number of people that cannot attend that meeting at a certain time.

When awareness requirements fail at runtime, a possible adaptation strategy is to *reconfigure* the system — i.e., to change one or more parameter values in order to improve the chances of success. This change is driven by qualitative differential expressions relating parameters and awareness requirements (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2011). For instance, the success rate of *Collect Timetables* is a function of *FhM - From how Many*:

$$success\ rate\ of\ Collect\ Timetables = f(FhM) \tag{2.1}$$

From how Many may vary from 0 to 1 (100%). Likewise, the success rate of *Collect Timetables* may vary from 0 to 1 (100%). Thus, this function's domain and co-domain present the same interval,

$$f : [0,1] \rightarrow [0,1] \tag{2.2}$$

where $[0,1]$ is a subset of $\mathbb{R}$.

Fig. 2.7 shows a graph with estimated values for this function. If the value of *FhM* is zero, then the success rate is of 1 (100%), since collecting the timetables of zero participants is trivial. The higher the value of *FhM*, the lower the value of the success rate, on a range of zero to one.

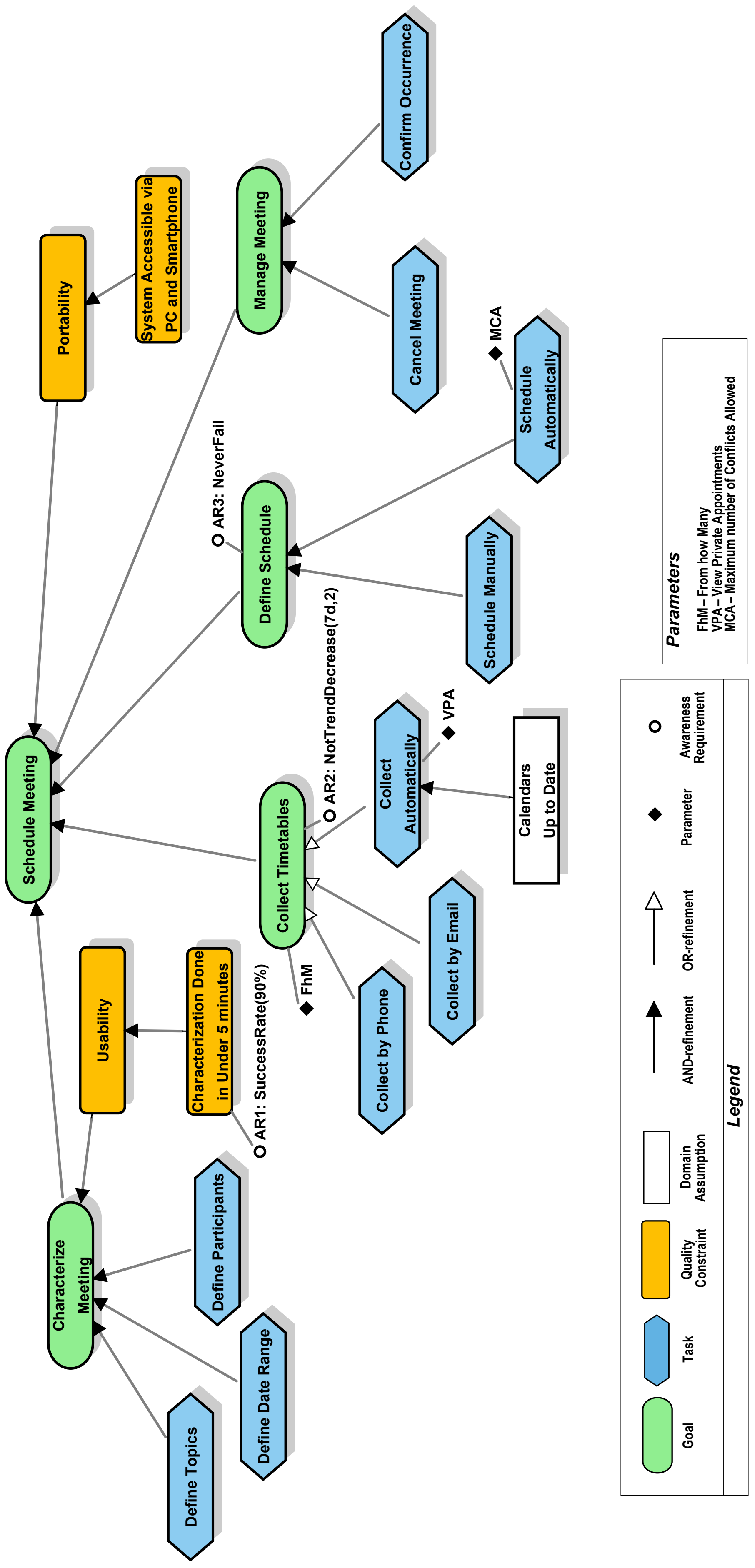
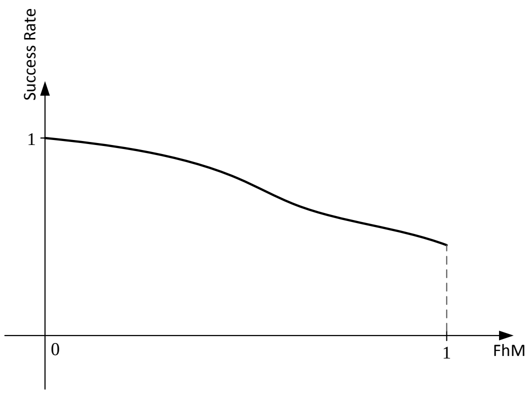


Figure 2.6: Partial requirements model of a Meeting Scheduler system, including adaptation elements

Figure 2.7: Graph of the success rate of *Collect Timetables* as a function of *FhM (From how Many)*



The derivative measures the change rate of the value of a function according to changes of the variable. As such, it provides information about the behavior of a function. If the derivative is positive, then the value of that function is increasing when the variable is increasing, and decreasing when the variable is decreasing. If, otherwise, the derivative is negative, then the value of that function is decreasing when the variable is increasing, and increasing when the variable is decreasing. The latter is the case for *success rate of Collect Timetables*. Thus,

$$\frac{d\,(success\ rate\ of\ Collect\ Timetables)}{d(FhM)} < 0 \tag{2.3}$$

Zanshin adopts a simplified notation to express these differential relations, as follows:

$$\Delta\,(awareness\ requirement/parameter)\,[a,b]\ \ \langle op\rangle\ C \tag{2.4}$$

On this notation, $[a,b]$ is the domain of the function — if not defined, it is assumed to be $\mathbb{N}$. $\langle op\rangle$ is the comparison operator to be used ($>$, $\geq$, $<$, $\leq$, $=$ or $\neq$), and C is any constant. In our example, the relation between the success rate of *Collect Timetables* and the *FhM* parameter can be defined as follows:

$$\Delta\,(AR2/FhM)\,[0,1] < 0 \tag{2.5}$$

The next section presents the Zanshin framework, which uses the extended goal model presented in this section as the basis for executing an adaptation cycle.

2.7 Zanshin Framework

In Zanshin (SOUZA, 2012), the change of parameters aiming to improve the satisfaction of awareness requirements is called reconfiguration. Using Control Theory terminology, this operation corresponds to adjusting *control input* in order to achieve a given *reference value*.

Besides *Reconfigure*, SOUZA; LAPOUCHNIAN; MYLOPOULOS (2012b) defines a set of operations (called *adaptation strategy patterns*), as follows:

- **Abort —** Specifies that the system should fail gracefully, by taking the necessary actions to reduce the potential damages of the failure.
- **ChangeParameter —** Modify a parameter to a pre-defined value.
- **Delegate —** Delegates the solution of the failure to an external actor and wait until the problem is fixed.
- **RelaxDisableChild —** Ignores the satisfaction/execution of the child of a given element.
- **Replace —** Replaces a given requirement for another one. It can be used, among others, to replace a task with another one, as well as to replace an awareness requirement with a less restrictive one.
- **Retry —** Re-attempts to satisfy the failed awareness requirement.
- **StrengthenEnableChild —** The opposite of **RelaxDisableChild**, it requires the satisfaction/execution of the child of a given element.
- **Warning —** Notifies an external actor about the present failure.

The *Abort*, *Delegate*, *Retry* and *Warning* strategy patterns only notify the target system that these actions must be performed — it is up to the target system to act on these requests sent by Zanshin. Other patterns also affect the internal reasoning of the Zanshin's adaptation component. For instance, when an awareness requirement *ARX* is *replaced* by an awareness requirement *ARY*, Zanshin's reasoning will start to ignore *ARX* and only monitor *ARY*. The *Delegate* pattern, besides requesting the notification of an external actor, will also make Zanshin wait for the resolution of the failure. Similarly, the *RelaxDisableChild* and *StrengthenEnable-Child* will influence Zanshin's reasoning.

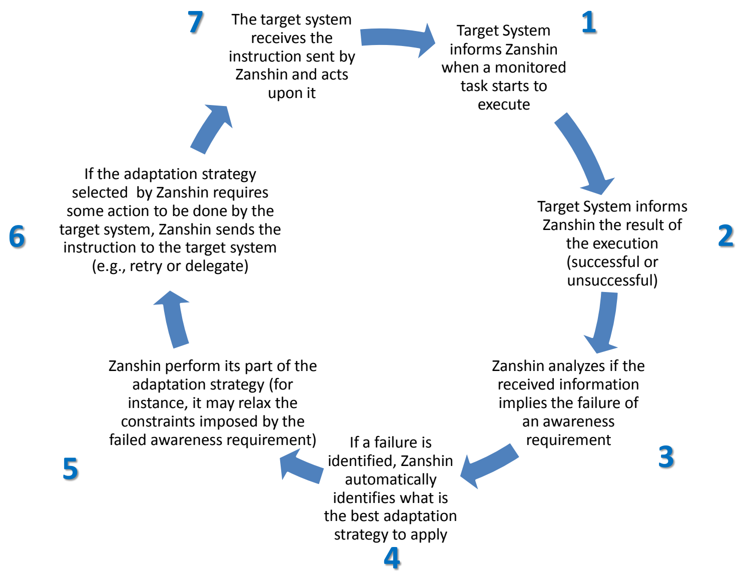
A prototype implementation of Zanshin's adaptation component is available for public use [2]. The target system provides execution data to Zanshin's component and enacts any adaptation action suggested by the component. On the other hand, Zanshin's component analyzes data provided by the target system and, when necessary, suggests the execution of adaptation

[2] Zanshin's wiki: https://github.com/sefms-disi-unitn/Zanshin/wiki

actions. In other words, this component facilitates the execution of the adaptation cycle depicted on Fig. 2.8.

The first two steps of the adaptation cycle are performed by the target system, consisting of sending data about the system execution to Zanshin's component. Based on this data and on the system's goal model, the component will perform its reasoning to identify whether an adaptation is required and, if necessary, what is the best adaptation to perform (steps 3 and 4). Then the component will perform the part of the adaptation strategy that affects the reasoning itself, and send instructions to the target system with the operations that need to be performed (steps 5 and 6). Finally, the target system will act on the instructions received from Zanshin (step 7), and the cycle continues. Further information on the reasoning algorithms used by Zanshin's component are available at SOUZA; LAPOUCHNIAN; MYLOPOULOS (2012a) and SOUZA (2012).

Figure 2.8: Adaptation cycle of the Zanshin framework


1
Target System informs Zanshin when a monitored task starts to execute
2
Target System informs Zanshin the result of the execution (successful or unsuccessful)
3
Zanshin analyzes if the received information implies the failure of an awareness requirement
4
If a failure is identified, Zanshin automatically identifies what is the best adaptation strategy to apply
5
Zanshin perform its part of the adaptation strategy (for instance, it may relax the constraints imposed by the failed awareness requirement)
6
If the adaptation strategy selected by Zanshin requires some action to be done by the target system, Zanshin sends the instruction to the target system (e.g., retry or delegate)
7
The target system receives the instruction sent by Zanshin and acts upon it

2.8 Summary

In this chapter the baseline of this work is presented: Requirements Engineering, Goal Models, Statecharts, Flow Expressions, Control Theory and Zanshin.

Goal Models, expressing the requirements of the system to be, are the starting point of the architectural design process comprised in the MULAS framework. This model will be incrementally refined with more information regarding adaptation, architectural design, and requirements themselves. The output is an enriched version of the requirements goal model, called Design Goal Model (DGM), which will be described in Chapter 3. One of the possible refinements concerns the execution flow of the system, described with Flow Expressions. These expressions allow the definition of the following flow concepts: sequentiality, parallelism, optionality and multiplicity. The DGM, containing flow expressions, can then be used to generate

Statecharts, which represent the behavioral view on a software system architecture.

Control Theory provides principles that can be used for managing the adaptation in a software system, e.g., feedback loop, parameters, indicators and relations. Different approaches for developing self-adaptive systems are based on these principles, such as Zanshin (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2011), which extends requirements goal models to include the information required to enact feedback loops. Moreover, it provides a standard set of components that, linked to a instrumented system, reads that extended goal model and control the execution of adaptation within that system. In Chapter 3 we further extend that goal model in order to handle not only requirements concerns but also architectural ones.

3 Adaptation on Requirements and Architecture

In this chapter we analyze which adaptation information is relevant at both requirements and architectural design levels, later introducing our proposal to present such information in an integrated model. The first section discusses adaptation concerns related to requirements, whereas the second section discusses adaptation concerns related to architectural design. The third section presents our proposal for integrating these concerns through a single model: the Design Goal Model (DGM).

3.1 Requirements

In the topic of requirements, an important distinction is that between early requirements and late requirements (CASTRO; KOLP; MYLOPOULOS, 2002). Early requirements focuses on understanding the problem being addressed, identifying and analyzing stakeholders, their intentions, and their relationships. In doing so, early requirements models describe the current state of affairs of a group of stakeholders, i.e., it is an *as-is* representation. On the other hand, late requirements is focused on the system-to-be, i.e., it concerns the definition of a system that will help satisfy the stakeholder goals. Since the scope of this thesis lies on the relationship between requirements and architecture, we are not concerned with links between early requirements and late requirements. Thus, our starting point is the late requirements model.

Use Cases is a largely adopted approach for documenting and analyzing late requirements, on which each use case describes (completely or partially) a usage or interaction scenario. While there is no standard template for documenting use cases, some of the fields usually included in their description are: actors, preconditions, triggers, postconditions, basic flow, and alternative flows (or extensions). The latter field can be used to explicitly define variability and adaptation for that scenario, as exemplified in Table 3.1.

Table 3.1 partially describes an use case for a meeting scheduler system, which is able to automatically define the schedule for a meeting given the timetable of the individuals expected

Table 3.1: Partial Use Case for *Define Schedule*

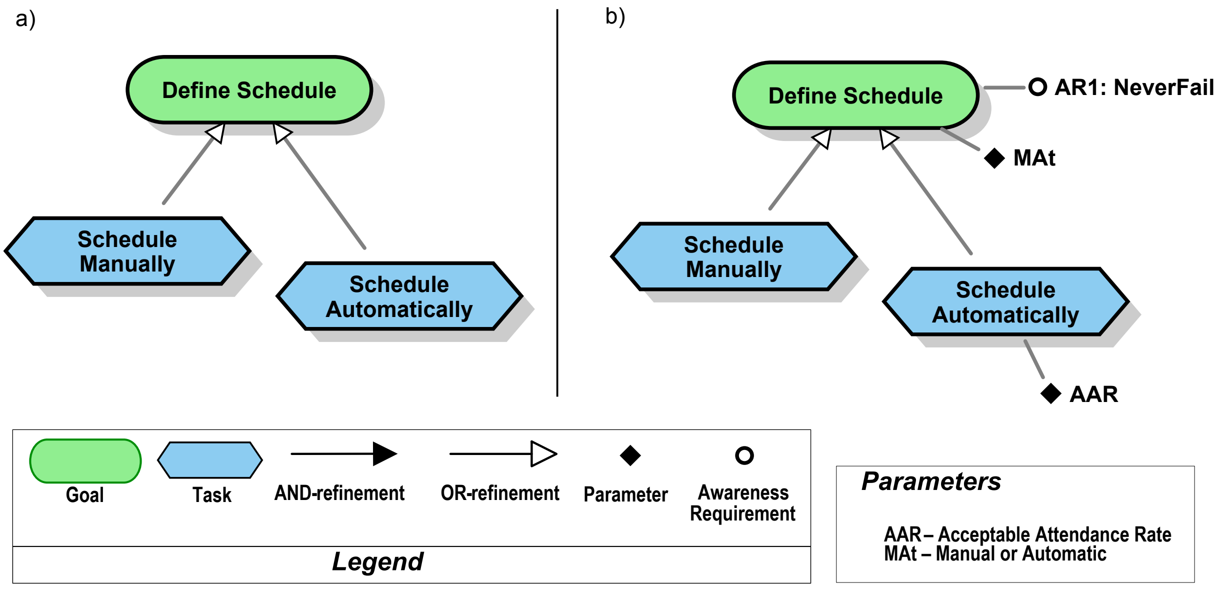
Actor	**End-User**
Preconditions	The End-User is responsible for this meeting Attendees for this meeting have been defined
Postconditions	Date and time for this meeting has been registered in the system
Basic Flow	1) End-User selects a meeting to schedule 2) End-User defines an acceptable attendance rate for the meeting 3) End-User requests a schedule 4) System identify possible schedules 5) End-User selects a schedule
Alternative Flows	3a) End-User request to input a schedule 3a.1) End-User input a schedule 3a.2) System estimates attendance rate for the meeting 3a.3) End-User confirm schedule 4a) There is no schedule that satisfies the acceptable attendance rate 4a.1) System notifies End-User 4a.2) System reduces acceptable attendance rate by 1 4a.3) Repeat step 4 4b) Timetables of attendees are not registered in the system 4b.1) System notifies End-User 4b.2) Go to step 3a.1

to attend the meeting. The first alternative flow (3a) illustrates the variability of this use case, where the scheduling can be done automatically as described in the basic flow, or manually as described in this alternative flow. The selection of which option to adopt will rely on the End-User, according to what is requested in step 3. The other two alternative flows represent system adaptation. In 4a, the system will reduce the acceptable attendance rate and try to define a schedule again whenever it is not possible to define a schedule that satisfies the acceptable attendance rate, while keeping the user informed. In 4b, the system is unable to schedule the meeting automatically due to lack of information (attendees' timetables), thus it switches to manual scheduling.

Goal models also allow to explicitly define requirements variability, in terms of OR-refinements. Fig. 3.1a shows the OR-refinement of *Define schedule*. Similar to the use case, there are two ways of defining the schedule, manually or automatically. However, traditional goal models are not capable of representing additional adaptation information, such as when to adapt and what parameters can be changed. Hence the need for goal modeling extensions, such as those proposed by the Zanshin framework (SOUZA et al., 2013).

Based on control theory, Zanshin allows to define *parameters* and *indicators* with respect to elements of goal models, as well as relationships between the former and the latter. Parameters express the variability that can be exploited during an adaptation. I.e., parameters describe what can be changed with respect to a given goal model element. For instance, Fig. 3.1b shows the Acceptable Attendance Rate (*AAR*) parameter of the *Schedule Automatically* task, as well as the Manual or Automatic (*MAt*) parameter of the *Define Schedule* goal. Indicators, which define what needs to be monitored during system execution, are expressed through awareness requirements. In the example of Fig. 3.1b the awareness requirements *AR1* shows that the *Define schedule* goal should *NeverFail*. Thus, this goal needs to be monitored at runtime so that, in case of failure, some reaction can be performed.

Figure 3.1: Partial Goal Model for *Define schedule*



The impact of parameters on indicators is expressed in terms of differential qualitative relations (Section 2.7). In this example, smaller numbers for AAR increase the likelihood of the system being able to successfully define a schedule. This relationship can be expressed as follows:

$$\Delta(AR1/AAR) < 0 \tag{3.1}$$

Lastly, adaptation strategies describe how to react in case of failure — i.e., what adaptation to perform. The strategies for the *Define schedule* example are presented in Fig. 3.2. If the scheduling is not performed successfully even though the timetables of the expected attendees are available (*applicability condition*), the system will reconfigure, notify the user and try again (*actions*). Given the relation on Eq. 3.1, the reconfiguration will be to decrease the value of Acceptable Attendance Rate (*AAR*). On the other hand, if the timetables are not available (*applicability condition*), the system will perform a specific change of parameter (switch *MAT* to *schedule manually*), notify the user, and try again (*actions*), similarly to the alternative flows presented in Table 3.1.

In summary, adaptation at the late requirements level is defined in terms of what can be changed, what needs to be monitored, the relationship between control variables and indicators, as well as when to check the indicators and how to react in case they are not satisfactory. Using the Zanshin terminology, this corresponds to *control variables*, *indicators*, *qualitative relations* and *adaptation strategies*, respectively. The next sections presents consideration on the topic of adaptation at the architectural design level..

Figure 3.2: Specification of the adaptation strategies for *Define schedule* with Zanshin

```
AwReq AR1: goal Define schedule should never fail
    - Adaptation Strategy 1.1:
        - Action: Reconfigure( )
        - Action: Warning(end-user)
        - Action: Retry(1000)
        - Applicability Condition: timetables available
    - Adaptation Strategy 1.2:
        - Action: ChangeParam(MAt, schedule manually)
        - Action: Warning(end-user)
        - Action: Retry(1000)
        - Applicability Condition: timetables not available
```

3.2 Architectural design

While on the previous section we discussed adaptation at the requirements level, here we discuss adaptation at the architectural level: first, considering architectural design as whole; then, focusing on the behavioral view with statecharts.

3.2.1 General adaptation

Architectural design is concerned with the generation of architectural models, which can include: components & connectors models for describing the system structure (GARLAN; MONROE; WILE, 1997); statecharts for describing the system behavior (BACHMANN et al., 2002); feature model for expressing the variability of the system (GURP; BOSCH; SVAHNBERG, 2001); and so on. These different models are complementary, each one capturing a particular view of the system being designed. Thus, different techniques are required in order to derive these different models.

In order to allow for flexible systems, it has been argued that architectural models need to support the variability contained in the requirements models (YU et al., 2008)(YU et al., 2008). Thus, instead of constraining the space of alternatives by selecting only one of the options to satisfy a goal in the requirements model, all of the options would be included in the system (when feasible). Indeed, this high variability is welcome for adaptive systems, since it increases the number of possible adaptations. Nonetheless, it is also important to consider that the design process is not simply a translation process, but it also involves the addition or creation of new elements that complement the requirements model. Thus, for the particular case of architectural design for adaptive systems, there are three new concerns that arise:

a. *Additional variability* — in the same way that alternative options can be identified throughout requirements elicitation, new alternative options may be identified during architectural design. For instance, different algorithms can be used to schedule a meeting automatically, each with its different benefits and drawbacks. Moreover, different behaviors could be defined for a set of elements. Considering the *Schedule Automatically* and *Schedule Manually* tasks, different behaviors are possible, such as: first schedule automatically, if not successful then schedule manually; let the user decide which scheduling to use, and then perform the scheduling; and so on. The alternatives identified during architectural design will expand the space of adaptation possibilities.

b. *Additional adaptation elements* — since we aim to support adaptation not only at the requirements level, but also at the architectural level, it is required to support the definition of adaptation elements that refer to architectural concerns. The adaptation elements are: indicators, parameters, qualitative relations and adaptation strategies. For instance, if a statechart shows that the meeting scheduler system sends reminders for a meeting in a set interval, this interval could be defined as a control variable of the system.

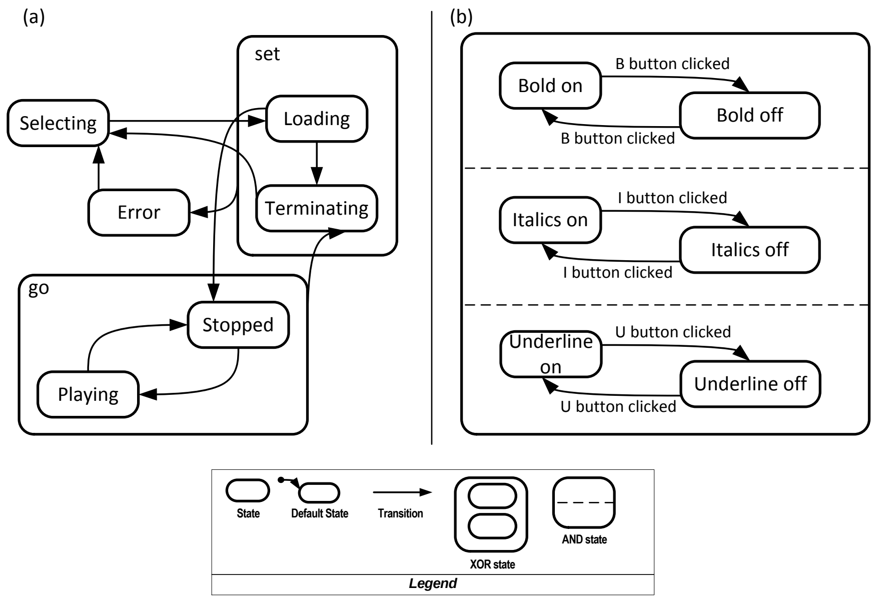
c. *Additional features to support adaptation* — the support of adaptation may require the inclusion of new features in the system. This is the case, for instance, when the system requires some kind of instrumentation in order to monitor the satisfaction of indicators.

3.2.2 Adaptation on Statecharts

As previously discussed, the development of adaptive systems allows the postponement of some decisions from design time to runtime. For instance, instead of deciding beforehand whether to use a web-service X or Y, an adaptive system may support the use of both web-services, making a decision at runtime based on criteria such as availability and performance. Other example is to support multiple resolutions on a video streaming app, to be selected based on bandwidth, instead of being predefined. This kind of flexibility found on adaptive systems allows the system to perform optimally, according to its context. Since adaptation is related to decisions, in order to discuss statechart adaptation it is necessary to understand the key decisions that are made when creating a statechart: which are the possible states of the system, and which are the possible transitions from one state to another.

The decision of which states to include in a statechart may vary according to the granularity and the concerns of interest of the architect. For instance, if an architect is concerned with network communication, the architect may describe a set of connection-related operations, as depicted in Fig. 3.3.a (adapted from EGYED; WILE (2001b)). This statechart shows the possible states for a video streaming client. Initially, the client selects a streaming server to use, and then start loading. From the *Loading* state, the system may go to the *Stopped* state, and

Figure 3.3: Examples of statecharts with different concerns: (a) shows the set of operations required to connect with a stream server, while (b) shows the possible states of buttons in a graphical user interface



then it may alternate between the *Stopped* and *Playing* states. From the *Loading*, *Stopped*, or *Playing* states, the system may enter the *Terminating* state, and then go either to the *Error* state or go back to the *Selecting* state. The *Error* state can also be entered from the *Loading*.

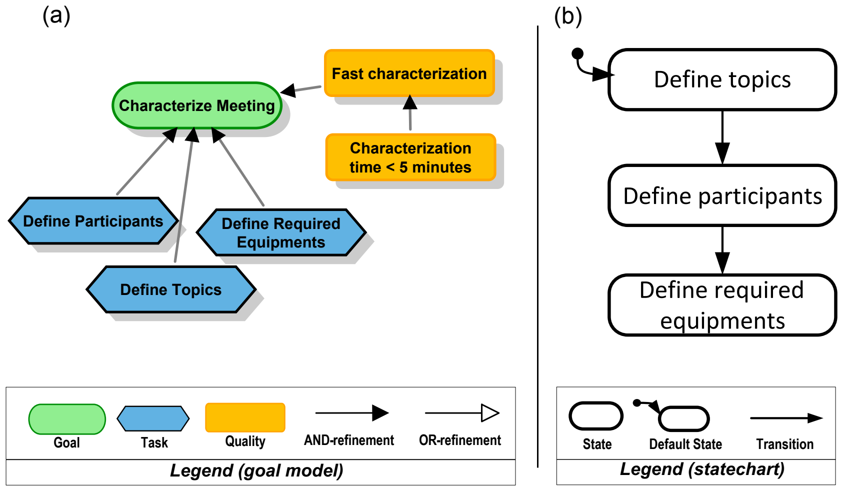
On the other hand, if the concern is the user interface of a system, the architect may define super states representing graphical elements of the user interface and sub states that describe the current situation of that element, such as in Fig. 3.3.b (adapted from KAYE; CASTILLO (2003)). This statechart shows the possible states for three user-interface buttons in a text editor: Bold, Italics, and Underline. The transition between the on and off states is determined by mouse clicks.

In this thesis, we propose the MULAS framework, which is concerned with orchestrating the execution of the different tasks of the system so that the stakeholder goals may be achieved. Thus, as described in Chapter 4, our approach assumes a direct mapping from leaf tasks of a goal model to states in a statechart. This mapping is illustrated in Fig. 3.4: the *Define Participants*, *Define Topics*, and *Define Required Equipments* tasks are mapped to the *Define Participants*, *Define Topics*, and *Define Required Equipments* state. Each state corresponds to the execution of its respective task. Because of this mapping, the proposed framework does not support the creation or removal of states at runtime: only behaviors defined at design time can be adopted at runtime.

On the subject of transitions between states, at the macro level we can analyze different possibilities for the execution flows of a system, while at the micro level there can be different specifications of the transitions.

The statecharts in Fig. 3.5 exemplify different possibilities for defining the execution

Figure 3.4: An example of the direct mapping from tasks in goal models (a) to states in statecharts (b)



flow of an excerpt of the Meeting Scheduler System, containing three tasks: *Define topics*, *Define participants*, and *Define required equipments*. In Fig. 3.5.a, the tasks are executed sequentially. In Fig. 3.5.b the tasks are also executed in sequence, with the difference that *Define participants* and *Define required equipments* are considered optional, which is evidenced by these additional transitions: from *Define topics* to *Define required equipments*, from *Define topics* to the exit of the superstate, and from *Define participants* to the exit of the superstate. Lastly, in Fig. 3.5.c the tasks may be concluded in any order.

Traditionally, only one of these alternative flows would be selected and then included in the system's statechart. However, in the context of adaptive systems, it can be interesting to develop the system supporting more than one of the possible flows, allowing to switch between different flows at runtime.

Moving towards the micro level, it is necessary to specify the details of each transition in the statechart, which are composed of triggers and conditions. The triggers are events that may happen during the system execution and provoke a transition from a source state to a target state, as long as the conditions of the transition hold true. In order to support adaptation of the system's behavior, we propose the parameterization of transitions.

Fig. 3.6.a shows an excerpt of a statechart with a timed transition, on which an e-mail will be sent every twenty-four hours. Fig. 3.6.b shows the same excerpt but now with a parameterized event, where TIR stands for *Time Interval between Reminders*. With a parameterized transition, the designer can define an initial value and then let the controller adjust it automatically through a feedback loop, aiming to achieve the best results possible.

In summary, in order to accommodate the new alternatives identified during behavioral design, we may create new parameters that can be modified by an adaptive framework. In the statechart model, these parameters may refer to (i) the selection of alternative behaviors; (ii) the definition of parameterized events; and (iii) the definition of parameterized conditions (as discussed in the previous subsection).

Figure 3.5: Different statecharts exemplifying different execution flows for the same set of tasks

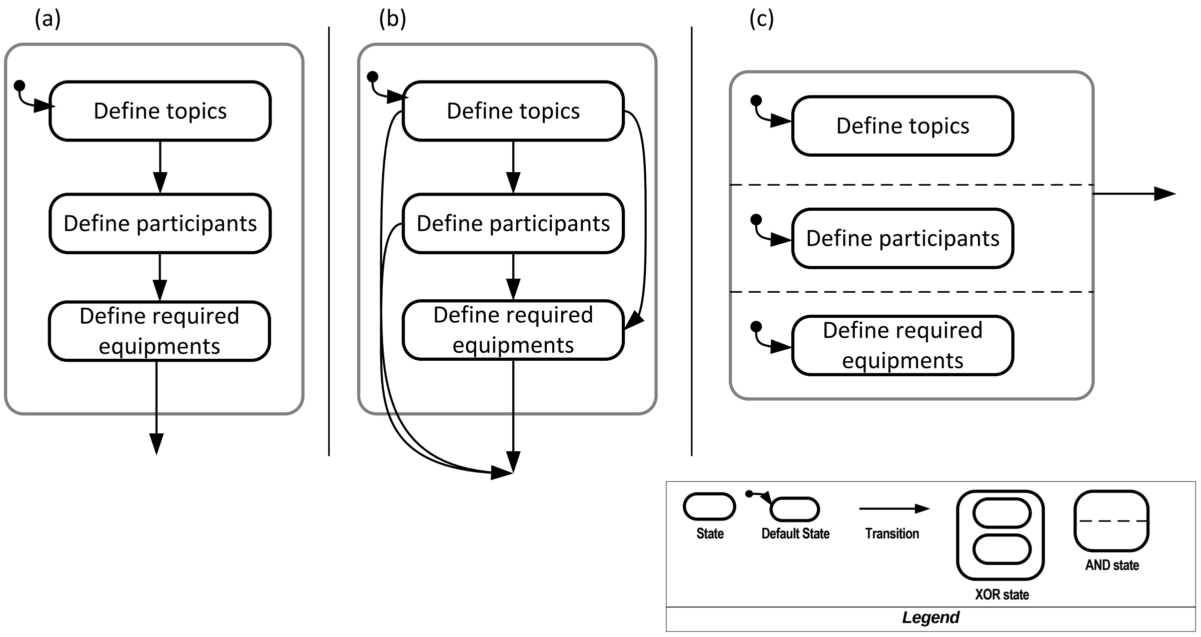
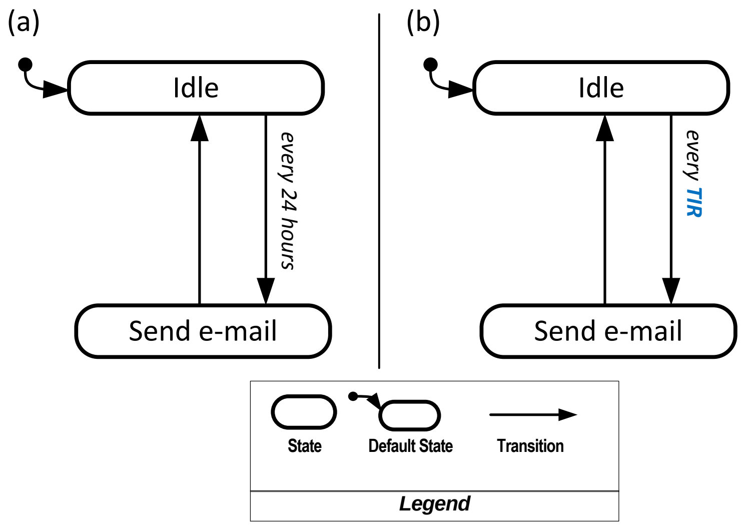


Figure 3.6: In the left-hand side, a timed transition with a static time interval; in the right-hand side, a timed transition with a parameterized time interval



In the next section we present our proposal on how to integrate adaptation at the architectural designed level, described in this section, with adaptation at the requirements level, described in the previous section.

3.3 Multi-Level Adaptation - the Design Goal Model

A key aspect of architectural design, as of any design activity, is its decision-making nature. This is evidenced by the Rational Unified Process (RUP) definition of software architecture: "the set of significant decisions about the organization of a software system (...)" (KRUCHTEN, 2004). Given a problem, which in this case are the software requirements, software architects will decide what is the *best* solution to be implemented, where *best* can be defined in terms of performance, cost, maintainability, and other criteria.

Some systems implement only one solution, while others, called flexible or configurable systems, implement different solutions. In the latter kind of system, the decision of what solution to use is then postponed from design time to deployment time or run time. In adaptive systems, this decision is made by the system itself, at run time. In order to capture the information required for supporting adaptation both at the requirements level and at the architectural level, we propose the Design Goal Model (DGM).

The Design Goal Model extends the goal model from the Zanshin framework in order to capture adaptation information at the architectural level. This model can be fed to the Zanshin framework in order to support runtime adaptation both at the requirements and the architectural level. Moreover, it can be used to automatically generate architectural models. Besides goals, tasks, quality constraints, parameters and awareness requirements, the Design Goal Model contains design tasks, design constraints, assignments and behavioral annotations.

In order to capture design decisions that refine how a certain goal can be achieved, how a certain task can be performed, and how a given quality constraint can be satisfied, we propose the inclusion of design representations of these same concepts: **(i)** *design tasks*, **(ii)** *design constraints*, and **(iii)** *design assumptions* in the Design Goal Model.

The use of requirements constructs for representing architectural design, in the context of goal modeling, has already been explored in the literature (CASTRO; KOLP; MYLOPOULOS, 2002; LAMSWEERDE, 2003; GRAU; FRANCH, 2007; PIMENTEL; FRANCH; CASTRO, 2011). As argued in (BOER; VLIET, 2009), there is a large similarity between architecturally significant requirements and design decisions. Nonetheless, we opt to differentiate design elements from their requirements cousins in order to make it clear, among other things, who (stakeholders or designers) is responsible for making decisions with respect to those elements, and in which phase of the project they appear. We are assuming that requirements are stated by stakeholders (customers, users). Thus, their rationale is mostly domain-related and changes must be negotiated and approved by stakeholders. Unlike design elements, stated by designers. In this case, the rationale is mostly technology-related, and changes are negotiated internally by designers. Visually, this differentiation is presented by using dashed borders for the design elements.

Design tasks are tasks that need to be supported by the system in order to achieve a goal or perform another task, but are not explicitly required by stakeholders. For instance, *Connect to Database*, *Parse Data*, *Run Quicksort Algorithm* (HOARES, 1962) are tasks that may be executed by a system, even though they may not be required by customers or clients.

Similarly, design constraints are constraints identified by architects which refine other elements of the system. It may refine stakeholders' quality constraints, or it may refer to technical elements of the system, such as algorithms, components and connections. Examples of design constraints: *run algorithm in less than 2 seconds*, *use 128-bits encryption*, *Three way handshake communication protocol*, *adopt Java as the main programming language*, *use a specific Java library*.

Lastly, design assumptions are assumptions made by the system architects which, if true, imply the satisfaction of its parent element. For instance, it can be assumed that a scalability requirement will be satisfied given that no more than fifty users will interact with the designed system simultaneously. However, if more than fifty users interact with the system simultaneously, that scalability requirement will no longer be satisfied.

Besides these additional elements, assignment annotations allow architects to define which actor, component, module or external system will be responsible for performing specific tasks of the system. Lastly, behavioral annotations define the possible execution flows of the system. With this set of refinement elements the architecture team will be able to design the system by refining the stakeholders' requirements.

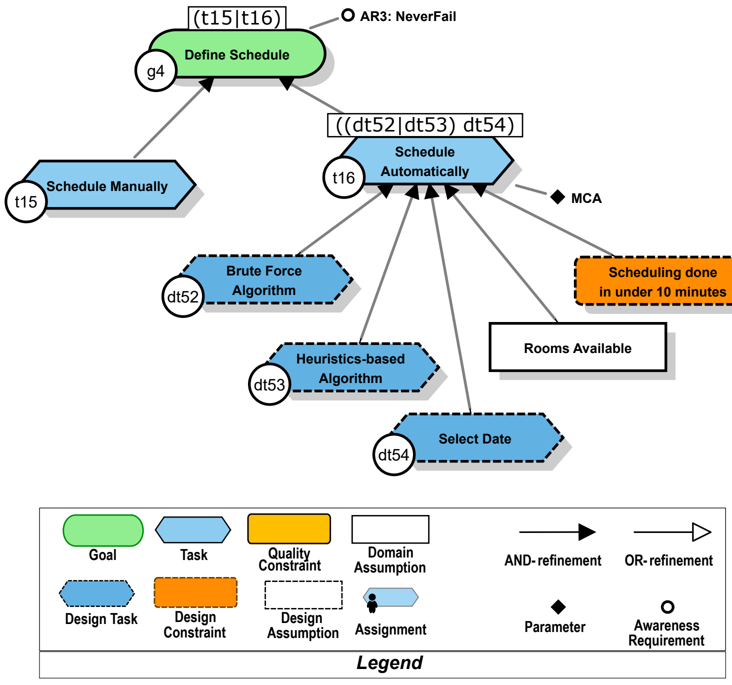
Fig. 3.7 shows an excerpt of the Design Goal Model for the Meeting Scheduler system, where the *Schedule Automatically* task is refined with three design tasks: *Brute Force Algorithm*, *Heuristics-based Algorithm*, and *Select Date*. A design constraint was defined, in agreement with the stakeholders, in order to limit the execution of these algorithms: *Scheduling done under 10 minutes*. Moreover, the domain assumption *Rooms Available* means that the automatic scheduling in only possible if rooms are available. The awareness requirement *AR3* indicates that the *Define Schedule* goal should never fail, while the *MCA* parameter defines the *Maximum amount of Conflicts Allowed* for the scheduling. The higher the number of conflicts allowed, the easier it will be to *Define Schedule* successfully. Lastly, the behavioral annotations define the flow of the system in terms of flow expressions. The annotation on *Define Schedule* consists of a selection between *Schedule Manually* (*t15*) and *Schedule Automatically* (*t16*). Similarly, the annotation on the *Schedule Automatically* task shows that only one of the algorithms will be used throughout a single execution flow: *Brute Force Algorithm* (*dt52*) or *Heuristics-based Algorithm* (*dt53*). After the execution of one of these algorithms, the *Select Date* task (*dt54*) will be performed.

The possible refinements in the Design Goal Model are summarized in Table 3.2. The first column shows a list of parent elements, while the remaining columns indicate children

Table 3.2: Possible refinements in the Design Goal Model

Parent	Children elements								
	Requirements elements				Design elements			Adaptation elements	
	Goal	**Task**	**Quality Constraint**	**Domain Assumption**	**Design Task**	**Design Constraint**	**Design Assumption**	**Awareness Requirement**	**Parameter**
Goal	✓	✓	✓	✓	✓	✓	✓	✓	✓
Task			✓	✓	✓	✓	✓	✓	✓
Quality Constraint		✓	✓	✓	✓	✓	✓	✓	✓
Domain Assumption				✓			✓	✓	✓
Design Task					✓	✓	✓	✓	✓
Design Constraint					✓	✓	✓	✓	✓
Design Assumption							✓	✓	✓
Awareness Requirement					✓			✓	
Parameter					✓				

Figure 3.7: Excerpt of the Design Goal Model for a Meeting Scheduler System



elements. A checkmark indicates that the parent can be refined with that kind of child element. For instance, the first checkmark on the *Goal* row indicates that a goal can be refined with other goals; the second checkmark on the *Goal* row indicates that a goal can be refined with tasks; so on and so forth. The absence of a checkmark indicates that the parent element cannot be refined with that particular child. For instance, the absence of a checkmark on the intersection of the *Task* row with the *Goal* column means that a task cannot be refined with goals.

All of the requirements elements (goal, task, quality constraint, and domain assumption) can be refined with any of the design elements (design task, design constraint, and design assumption), as shown on Table 3.2. Goals can also be refined with (sub-)goals, tasks, quality constraints and domain assumptions. Quality constraints, while non-functional requirements, can be refined not only with other quality constraints and domain assumptions but also with tasks, similarly to operationalizations in the NFR Framework (CHUNG et al., 2000). The same rationale applies to refining design constraints with design tasks. We are adopting Zanshin's notion of having tasks as the lowest level requirements, hence tasks cannot be refined with further tasks. Domain assumptions cannot be operationalized, thus they can only be refined with other domain assumptions, design assumptions, or adaptation elements. Similarly, design assumptions can only be refined with other design assumptions or with adaptation elements. Design elements can be refined only with other design elements or adaptation elements. In fact, any requirements element or design element can be refined with adaptation elements. Awareness requirements can be refined with another awareness requirements or with design tasks, with the latter meaning that the design task is related to the monitoring of the parent awareness requirement. Parameters can also be refined with design tasks, meaning that the task is related to enacting changes to that parameter. For instance, considering a parameter *door open or closed* in a smarthome system, the non-trivial tasks *Open Door* and *Close Door* are required in order to enact changes in that parameter.

Table 3.3 presents the possible annotations in the Design Goal Model. Goals, tasks and design tasks can be annotated with behavioral annotations, whereas only tasks and design tasks can be annotated with assignments.

Table 3.3: Elements that can be annotated with flow expressions and assignments

Element	Annotation	
	Behavioral Annotation	**Assignment**
Goal	✓	
Task	✓	✓
Quality Constraint		
Domain Assumption		
Design Task	✓	✓
Design Constraint		
Design Assumption		
Awareness Requirement		
Parameter		

The metamodel of the Design Goal Model (Fig. 3.8), along with its constraints (Listing 3.1), formalize the possible refinements previously described. Each node in the model can be either a requirements element (goal, task, quality constraint and domain assumption), a design element (design task, design constraint and design assumption), or an adaptation element (awareness requirement and parameter). These nodes may have behavioral annotations, in the form of flow expressions, whereas assignments are defined as attributes of tasks and design tasks.

Additionally, the constraints of the metamodel are defined with OCL (OMG, 2012) and presented in Listing 3.1, in conformity with Table 3.2. The first invariant (*mustHaveDifferentTargets*) prevents a node from having a link with itself. The remaining invariants formally define the possible refinements for each element: goal, task, quality constraint, domain assumption, design task, design constraint, design assumption, awareness requirement and parameter.

In order to mitigate the complexity of the Design Goal Model, different views were defined: requirements, design, assignment and behavior. These views prevent the visualization of elements that are not necessary for the task at hand. For instance, when modifying the system requirements it is not relevant to visualize design elements. Similarly, when including the design elements, it is not essential to observe behavioral annotations. Table 3.4 specifies the elements visible on each view, with the visible elements marked with check marks and optionally visible elements are marked with question marks.

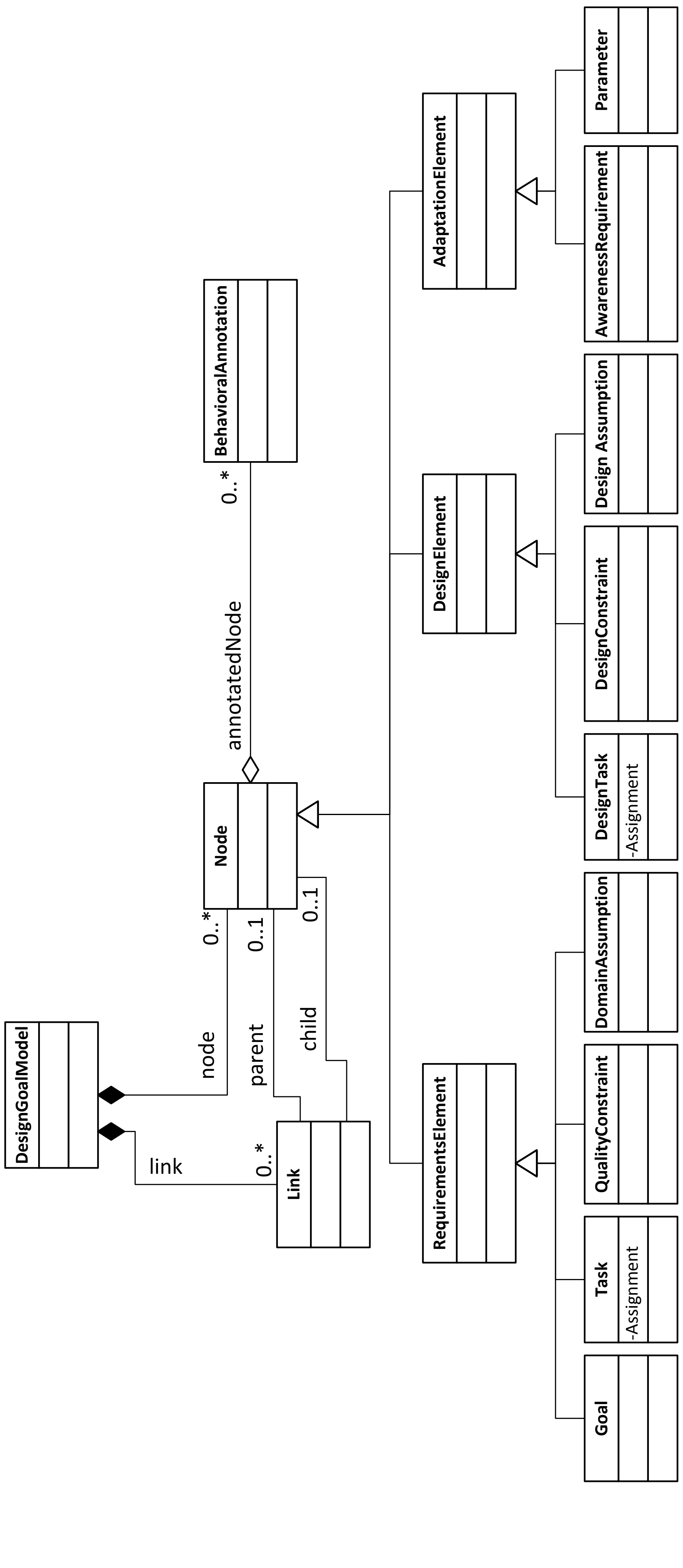


Figure 3.8: Metamodel of the Design Goal Model

Listing 3.1: OCL constraints that define the possible refinements in the Design Goal Model

```
context Link inv mustHaveDifferentTargets:
  self.parent <> self.child

context Link inv goalAsParent:
  self.parent.oclIsTypeOf(Goal) implies
    self.child.oclIsKindOf(RequirementsElement) or
    self.child.oclIsKindOf(DesignElement) or
    self.child.oclIsKindOf(AdaptationElement)

context Link inv taskAsParent:
  self.parent.oclIsTypeOf(Task) implies
    self.child.oclIsTypeOf(QualityConstraint) or
    self.child.oclIsTypeOf(DomainAssumption) or
    self.child.oclIsKindOf(DesignElement) or
    self.child.oclIsKindOf(AdaptationElement)

context Link inv qualityConstraintAsParent:
  self.parent.oclIsTypeOf(QualityConstraint) implies
    self.child.oclIsTypeOf(Task) or
    self.child.oclIsTypeOf(QualityConstraint) or
    self.child.oclIsTypeOf(DomainAssumption) or
    self.child.oclIsKindOf(DesignElement) or
    self.child.oclIsKindOf(AdaptationElement)

context Link inv domainAssumptionAsParent:
  self.parent.oclIsTypeOf(DomainAssumption) implies
    self.child.oclIsTypeOf(DomainAssumption) or
    self.child.oclIsTypeOf(DesignAssumption) or
    self.child.oclIsKindOf(AdaptationElement)

context Link inv designTaskAsParent:
  self.parent.oclIsTypeOf(DesignTask) implies
    self.child.oclIsKindOf(DesignElement) or
    self.child.oclIsKindOf(AdaptationElement)

context Link inv designConstraintAsParent:
  self.parent.oclIsTypeOf(DesignConstraint) implies
    self.child.oclIsKindOf(DesignElement) or
    self.child.oclIsKindOf(AdaptationElement)

context Link inv designAssumptionAsParent:
  self.parent.oclIsTypeOf(DesignAssumption) implies
    self.child.oclIsTypeOf(DesignAssumption) or
    self.child.oclIsKindOf(AdaptationElement)

context Link inv awarenessRequirementAsParent:
  self.parent.oclIsTypeOf(AwarenessRequirement) implies
    self.child.oclIsTypeOf(DesignTask) or
    self.child.oclIsTypeOf(AwarenessRequirement)

context Link inv parameterAsParent:
  self.parent.oclIsTypeOf(Parameter) implies
    self.child.oclIsTypeOf(DesignTask)

context BehavioralAnnotation inv elementsThatCanBeAnnotated:
  self.annotatedNode.oclIsTypeOf(Goal) or
  self.annotatedNode.oclIsTypeOf(Task) or
  self.annotatedNode.oclIsTypeOf(DesignTask) or
```

Table 3.4: The different views of the Design Goal Model. Question marks indicate that the visualization of the element is optional in that view

Element	Views			
	Requirements	**Design**	**Assignment**	**Behavior**
Goal	✓	✓	✓	✓
Task	✓	✓	✓	✓
Quality Constraint	✓	✓	✓	✓
Domain Assumption	✓	✓	✓	✓
Design Task		✓	✓	✓
Design Constraint		✓	✓	✓
Design Assumption		✓	✓	✓
Awareness Requirement	?	?	?	?
Parameter	?	?	?	?
Assignment			✓	
BehavioralAnnotation				✓

The syntax of the design goal model, illustrated in Fig. 3.7, was based on existing goal modeling notations for the sake of familiarity. Nonetheless, it was adapted taking into consideration principles for designing effective visual notations (MOODY, 2009):

1. **Principle of Semiotic Clarity: there should be a 1:1 correspondence between semantic constructs and graphical symbols —** this principle was followed since there is no symbol redundancy (multiple symbols for the same concept), no symbol overload (different concepts for the same symbol), and no symbol excess (symbols that do not represent any concept). There is symbol deficit (concept not represent by a symbol), such as differential relations. However, as noted in (MOODY, 2009), this is welcome in a software engineering context in order to reduce diagrammatic complexity.
2. **Principle of Perceptual Discriminability: different symbols should be clearly distinguishable from each other —** this principle is against notations that adopt too similar symbols for different concepts, such as the Data Flow Diagrams from Gane & Sarson (GANE; SARSON, 1979), which relies solely on rectangles. In the design goal model, besides different shapes we also adopted different colors, which is an example of coding redundancy. This helps to distinguish between, among others, goals and quality constraints, which are represented by shapes within the same family of rounded rectangles. The following design elements present the same shapes of their requirements counterparts: design tasks, design constraints, and design assumptions. This similarity is intentional, since they represent similar concepts at different abstraction levels. Nonetheless, they are discernible not only by their border lines but also by their brightness. Moreover, these elements are visually

distinguished from adaptation elements (awareness requirements and parameters), as well as the behavioral element (flow expression), through the use of different structures.

3. **Principle of Semantic Transparency: use visual representations whose appearance suggests their meaning —** ideally, it would be possible to infer the meaning of a symbol from its appearance. As an example, it is suggested to represent actors in *i** models with stick figures (as in use cases), to represent resources with trees, and so on (MOODY; HEYMANS; MATULEVICIUS, 2010). However, considering that goal modeling notations such as *i** and Tropos are already well known in the requirements engineering community, we decided to keep using their standard shapes. The one instance where we applied this principle is the assignment symbol: we use the icon of a person to express the assignment of a task to someone.
4. **Principle of Complexity Management: include explicit mechanisms for dealing with complexity —** the complexity management mechanism of the design goal model are its different views. Moreover, *i** levels of complexity are prevented by adopting a strict tree structure, rather than a graph one.
5. **Principle of Cognitive Integration: include explicit mechanisms to support integration of information from different diagrams —** the design goal model is lacking with regard to this principle, since it does not provide visual indications of its integration with statecharts.
6. **Principle of Visual Expressiveness: use the full range and capacities of visual variables —** the different concepts expressed in the design goal model are differentiated by: shape (rectangle, rounded rectangle, hexagon, circle and diamond), color (e.g., goals are green, while tasks are blue), brightness (design elements are darker than requirements elements), structure (e.g., the textual element of a goal is contained within its symbol, while the textual element of a parameter is at the side of its symbol), and textual cues (e.g., awareness requirements are preceded by an identificator in the form of ARxx).
7. **Principle of Dual Coding: use text to complement graphics —** three concepts can be distinguished through textual cues: awareness requirements, parameters and behavioral annotations. Awareness requirements are preceded by an identificator in the form of ARxx. Parameters are represented with three letters acronyms, even though this constraint is not mandatory. Lastly, the text of behavioral annotations are identifiable for being in the form of regular expressions.
8. **Principle of Graphic Economy: the number of different graphical symbols should be cognitively manageable —** the design goal model exceeds the recommended limit of six symbols. In order to mitigate this issue, we decided to use similar shapes for similar concepts (e.g., tasks and design tasks), while strengthening their visual expressiveness.

9. **Principle of Cognitive Fit: use different visual dialects for different tasks and audiences —** this principle suggests that different dialects (e.g., different symbols) can be used in different contexts, considering the different needs of expert and novice modelers, as well as the characteristics of differentia media (e.g., software tools and paper sketching). Considering that the design goal model is still incipient, this principle has not been applied yet.

3.4 Summary

The Design Goal Model (DGM), presented on this chapter, allows to specify software systems' adaptation based not only on high level concerns expressed by stakeholders, but also on low level, technical aspects of the system. However, creating such a model is not a trivial endeavor. In order to facilitate the system design, leading to the creation of a DGM, we propose an architectural design process, described in the next chapter. Starting with an initial requirements goal model, this process proposes a series of iterative and incremental refinements to be applied in the model resulting from architecture- and adaptation-related decisions.

Moreover, the process described in the next chapter also supports the generation of statecharts from a DGM, which provide a proper visualization of the behavioral view on the system architecture. This statechart can then be refined to include adaptation elements, which allow to reify the adaptation strategies defined for the system.

4 From Requirements to Architectural Design

In this chapter we present a process for moving from requirements towards architectural design, illustrated with examples from a meeting scheduler system. In the first section, the complete requirements of a Meeting Scheduler system are presented, based on its requirements goal model. The next section describes the process itself, with its eight steps: *Identify design tasks, constraints and assumptions*; *Assign tasks*; *Define basic flows*; *Identify indicators, parameters and relations*; *Specify adaptation strategies*; *Generate base statechart*; *Specify transitions*; and *Include adaptation elements*.

4.1 Requirements for the Meeting Scheduler system

The requirements for a Meeting Scheduler system were briefly introduced in Section 3.1. Building on that, this current section presents the requirements resulting from a series of iterations of the Requirements Engineering and Architectural Design process. The final requirements model is shown in Fig. 4.1.

The main goal of the Meeting Scheduler system is to *Schedule Meeting*. In order to achieve that goal, it is necessary to fulfill the following sub-goals: *Characterize Meeting*, *Collect Timetables*, *Define Schedule*, *Manage Meeting*, and *Administer System*. Meeting characterization is the moment when a user start to call for a meeting, setting up its properties such as the people to be invited and a date range where the meeting may happen. The gathering of timetables from the invited personnel is required in order to identify at what time a meeting could be scheduled so that the attendance rate is maximized. The *Define Schedule* goal refers to the actual scheduling of a meeting. Tasks related to updating a meeting, such as canceling and changing rooms are part of the *Manage Meeting* goal. Lastly, *Administer System* is a supporting goal related to updating and visualizing data in the system, such as managing users and visualizing reports.

Besides the aforementioned sub-goals, there are two overarching quality constraints that the stakeholders expect to be satisfied: *Scalability* and *Portability*. *Scalability* refers to the ability of the system to be performant even with a large amount of users or requests. *Portability*

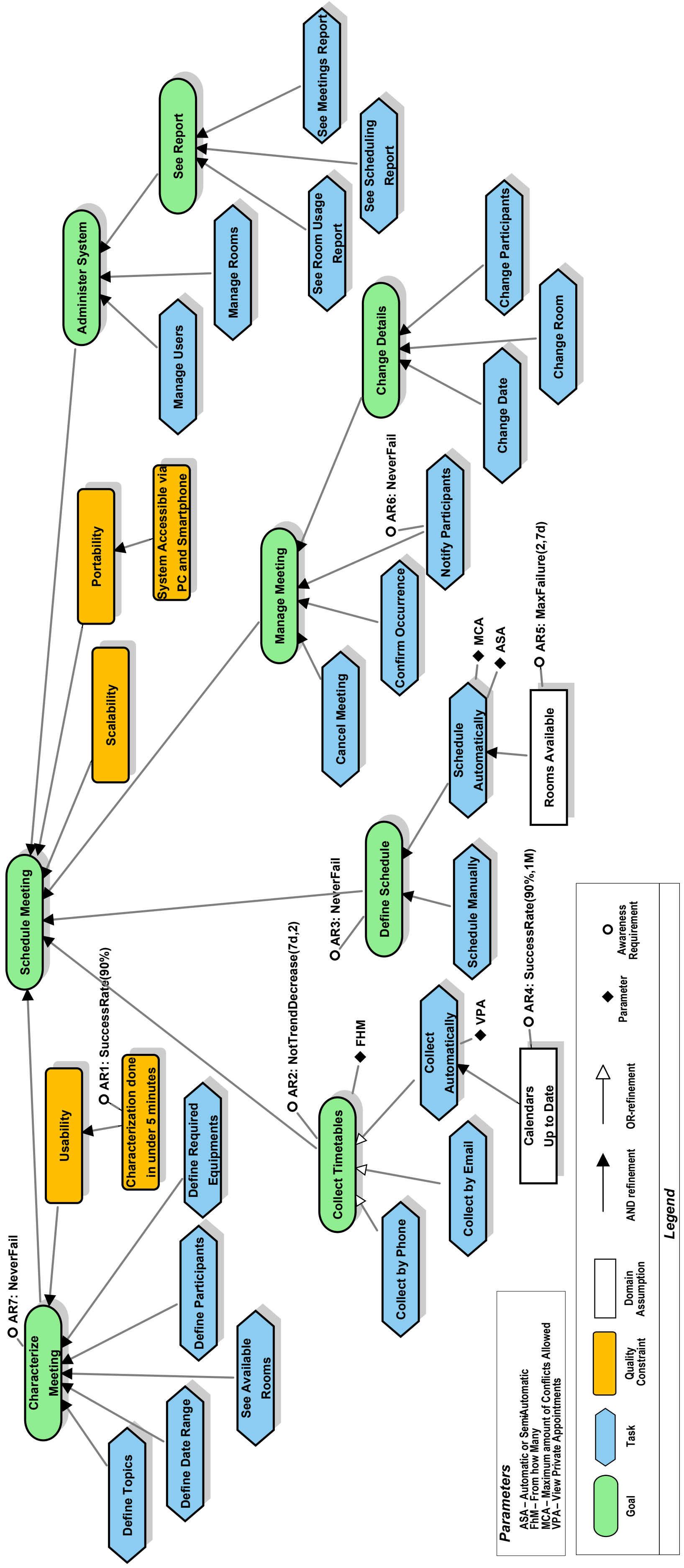


Figure 4.1: Requirements of the Meeting Scheduler system

was defined as being able to access the meeting scheduler system through different devices, so that users can interact with the system on the fly, as stated by the *System Accessible via PC and Smartphone* quality constraint.

The *Characterize Meeting* goal is refined onto five tasks: *Define Topics*, *Define Date Range*, *See Available Rooms*, *Define Participants*, and *Define Required Equipments*. The *Define Topics* task is where the user, that is calling a meeting, defines its agenda, while *Define Date Range* delimits the boundaries for the meeting scheduling. The *See Available Rooms* and *Define Required Equipments* tasks are related to room reservation: the first one provides a list of rooms available for that meeting, whereas the second one allows the user to specify which equipments will be needed for that meeting, which may constrain the room selection. For instance, if a video projector is required for a meeting, only rooms with that equipment can be chosen. Moreover, the *Define Participants* task is used to declare who is going to be invited. Considering that this characterization could be cumbersome if poorly implemented, the stakeholders defined a *Usability* constraint on which the characterization must be completed in less than five minutes, as defined by its sub-constraint: *Characterization done in under 5 minutes*.

In order to select the best meeting date it is important to obtain the timetables that express at which time the invited participants are available. Three alternative means for achieving the *Collect Timetables* goal were identified: *Collect by Phone*, *Collect by Email*, and *Collect Automatically*. Since automatic collection requires less human effort, it is the preferred option; nonetheless the other alternatives are not necessarily discarded.

The actual scheduling may be performed automatically or manually, as defined by the *Define Schedule* goal. Although in most cases the automatic scheduling is preferred, in some cases the meeting organizer may prefer to schedule manually, thus both options need to be included in the system. This is the case, for instance, when the meeting involves external guests whose timetables are not available for the system.

In order to be able to *Manage Meeting*, the Meeting Scheduler systems needs to support the following tasks: *Cancel Meeting*, *Confirm Occurrence* after the meeting is held, *Change Room*, *Change Date*, and *Notify Participants* of changes in meetings.

The *Administer System* goal can be achieved through the *Manage Users* and *Manage Rooms* tasks, as well as the *See Report* goal. These tasks are example of requirements that were identified during the architectural design phase, i.e., while creating a solution to the earlier elicited requirements. Considering that different people would interact with the Meeting Scheduler system, some with different permissions, some that would receive e-mails from the system, and so on, it was observed that user information would be required. Considering personnel turnaround, this information would be modified throughout the system's service life time, thus some kind of management would be necessary. Similar management would most likely be necessary for the meeting rooms and their equipments. These tasks were then proposed and accepted to be included in the requirements goal model. Lastly, the *See Report* goal is related to the managers' need of obtaining information about the meetings and about the system itself, manifested in three tasks: *See Room Usage Report*, *See Scheduling Report*, and *See Meetings Report*.

Besides goals, tasks, quality constraints and domain assumptions, the requirements goal model also consists of awareness requirements and parameters. SOUZA (2012) suggests the following sources for the identification of awareness requirements: critical requirements, non-functional requirements, preferable solutions, trade-offs, preemptive adaptation, other awareness requirements (meta-awareness), and qualitative elicitation.

The awareness requirements for the meeting scheduler system, indicating which elements of the goal model need to be monitored at runtime, are as follows:

- **AR1: SuccessRate(90%)** — Quality constraint *Characterization done in under 5 minutes* should have a success rate of at least 90%.
- **AR2: NotTrendDecrease(7d,2)** — The success rate of the *Collect timetables* goal should not decrease two weeks in a row.
- **AR3: NeverFail** — The *Define Schedule* goal should never fail.
- **AR4: SuccessRate(90%,1M)** — The *Calendars Up to Date* domain assumption should have a success rate of at least 90%, measured every month.
- **AR5: MaxFailure(2,7d)** — The *Rooms Available* domain assumption should fail at most twice every week.
- **AR6: NeverFail** — The *Notify Participants* task should never fail.
- **AR7: NeverFail** — The *Characterize Meeting* goal should never fail.

While awareness requirements express what needs to be monitored in the system, parameters define what can be modified at runtime, aiming to improve the satisfaction level of the awareness requirements. The parameters of the Meeting Scheduler system are:

- **ASA: Automatic or Semi-Automatic** — This parameter provides two options to the *Schedule Automatically* task: fully automatic or semi-automatic. With the former, the system will define a single best date for the meeting, which will then be confirmed or not by the user. With the latter, the system will provide a range of good dates, leaving the selection of the best one to the user.
- **FHM: From How Many** — This parameter defines the satisfactory percentage of meeting participants' timetables that must be collected. Ideally, all timetables should be collected. However, if it is difficult to obtain all timetables, the expected percentage can be reduced.

- **MCA: Maximum amount of Conflicts Allowed** — When scheduling a meeting, the best scenario would be to select a date where all the invited personnel is able to attend. Considering that this scenario is not always possible, this parameter determines what is the maximum amount of conflicts that will still lead to a satisfactory scheduling. The higher the value of MCA, the easier it is to schedule a meeting.

- **VPA: View Private Appointments** — This parameter defines whether the system will be able to view private appointments in its users' calendars. While accessing such data may raise privacy concerns, it potentially provides better results in terms of attendance rate.

The next section shows the incremental refinement of goal models with the inclusion of design elements, as part of the Architectural Design process. In particular, it is also possible to identify new awareness requirements and parameters, as it will be described in Section 4.2.4.

4.2 Architectural Design

The goal of this architectural design process is to guide systems designers and architects on the definition of system behavior. By following the process, they will be able to: refine the requirements model with design elements, resulting on a design goal model; identify adaptation opportunities related to system design, as well as to specify them; define system behavior in such a way as to enact the specified adaptation.

The process comprises eight steps, as depicted by its Business Process Model and Notation (BPMN) diagram[1] (OMG, 2011) in Fig. 4.2. The first five steps are related to the refinement of design goal models: *Identify design tasks, constraints and assumptions*; *Assign tasks*; *Define basic flows*; *Identify indicators, parameters and relations*; and *Specify adaptation strategies*. The other three steps are related to statecharts: *Generate base statechart*; *Specify transitions*; and *Include adaptation elements*.

While these steps may be followed mostly sequentially, waterfall-like, in realistic settings it is expected that the architect will go back and forth, by introducing additional refinements to already refined elements. Furthermore, this process constitutes a loop, as shown by the circular arrow on the bottom-center region of the figure. In fact, in accordance with the Twin Peaks directive of iterative and incremental design (NUSEIBEH, 2001), it is possible to generate statecharts from partial design goal models, i.e., from models that have not been fully refined all the way down to its leaf elements. For instance, if only the behavior of the immediate children of a root goal is defined, the result is a very high-level statechart. As further refinements are specified and new elements are included in the model, the resulting statechart gets

[1] In the text of this thesis the word *step* is adopted when referring to BPMN tasks, as to prevent confusion with goal models' tasks

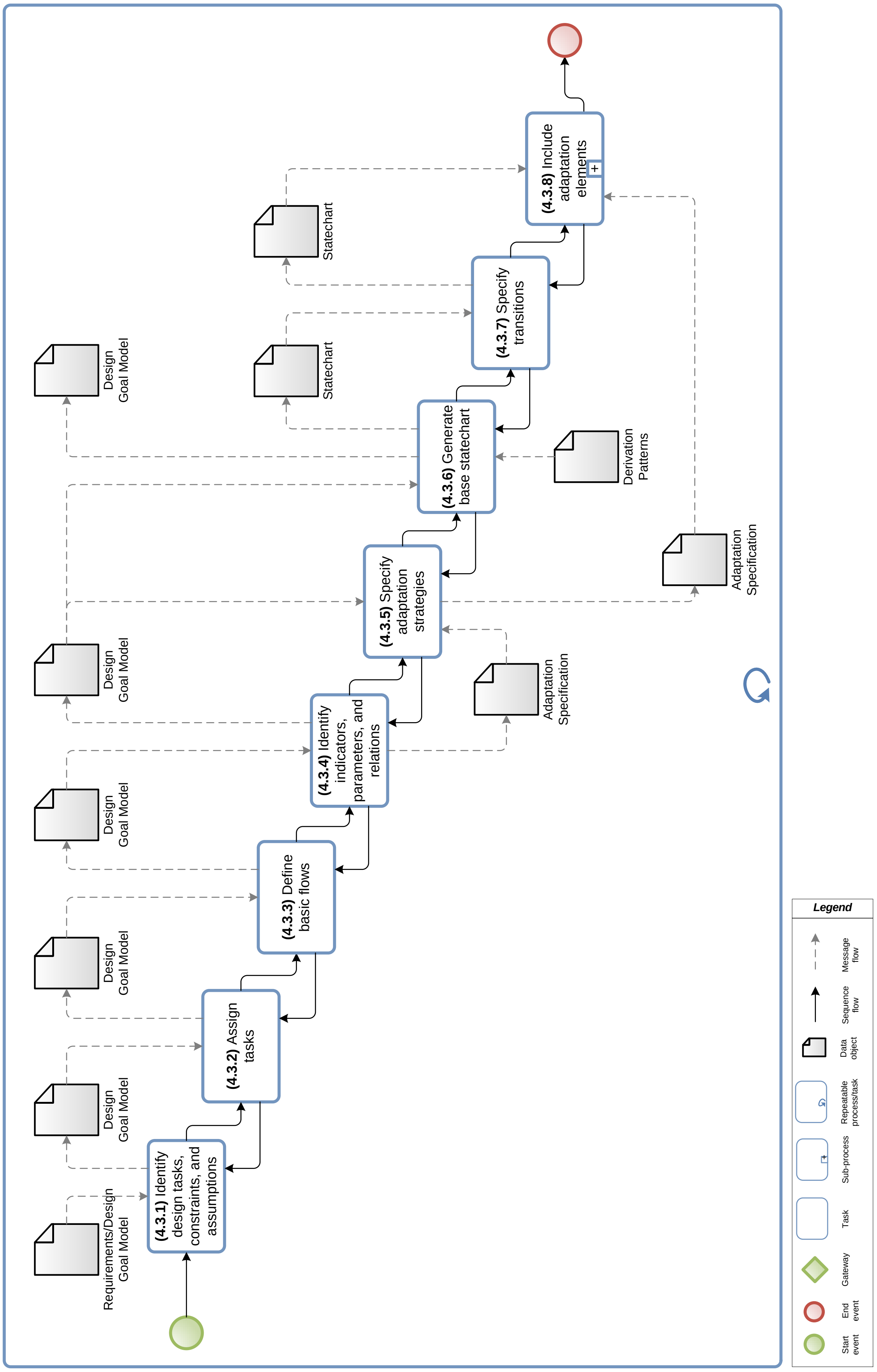

(4.3.1) Identify design tasks, constraints, and assumptions
(4.3.2) Assign tasks
(4.3.3) Define basic flows
(4.3.4) Identify indicators, parameters, and relations
(4.3.5) Specify adaptation strategies
(4.3.6) Generate base statechart
(4.3.7) Specify transitions
(4.3.8) Include adaptation elements
Requirements/Design Goal Model
Design Goal Model
Design Goal Model
Design Goal Model
Design Goal Model
Design Goal Model
Adaptation Specification
Adaptation Specification
Derivation Patterns
Statechart
Statechart
Legend
Start event
End event
Gateway
Task
Sub-process
Repeatable process/task
Data object
Sequence flow
Message flow

Figure 4.2: Overview of the architectural design process

more complete. Thus, it is not necessary to fully complete one step in order to proceed to the next one.

The first step, *Identify design tasks, constraints and assumptions*, supports the refinement of a goal model by including elements that are not initially required by the stakeholders, but are relevant from the architectural point of view, expressed as design tasks, design constraints and design assumptions. The second step, *Assign tasks*, consists of assigning the responsibilities for the execution of tasks —- e.g., tasks that will be performed by an external actor (human or otherwise). This assignment is helpful for defining the scope of the system.

In the next step, *Define basic flows*, the architect introduces possible flows for every sub-tree in the goal model. Roughly, these flows describe in what order are the sub-elements to be fulfilled or executed, so that their parent element can be considered fulfilled or executed. These flows are expressed as alternative flow expressions, introduced as annotations to a goal model using a top-down, bottom-up, or middle-out strategy. These expressions are later used to automatically generate a statechart that represents the system's behavior.

The next two steps are related to the adaptation capabilities of the system: *Identify indicators, parameters and relations* and *Specify adaptation strategies*. The former is concerned with including in the design goal model those elements proposed by Zanshin (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2011), but now also considering the design elements previously included in the first step. In the *Specify adaptation strategies* step it is defined how the system will react to failures — e.g., by retrying the execution of a task, or by changing the parameters described in the goal model.

The second part of the process is related to system behavior. The first step, *Generate base statechart*, makes use of derivation patterns to automatically create a statechart from the flow expressions previously defined. Although flow expressions are a useful intermediate abstraction between goal models and statecharts, they are not as expressive as statecharts. Thus, in the next step, *Specify transitions*, the transitions of the statechart are refined with their events and conditions, which are identified by analyzing when any given transition should take place.

The last step, *Include adaptation elements*, is concerned with reifying the adaptation cycle. Adaptation elements included during this step allow to monitor the system execution, as well as to enact the adaptation strategies that were defined in the *Specify adaptation strategies* step.

The following subsections further describe each step of the architectural design process, using the Meeting Scheduler system as a running example.

4.2.1 Identify Design Tasks, Constraints and Assumptions

Requirements expressed as goal models describe the problem space for the system-to-be, capturing concerns from different stakeholders such as customers, users and domain experts. During design, i.e., as we move towards the solution space, the different elements of the requirements model need to be further refined, reflecting design decisions that have been made. These decisions can be classified in three categories: existence decisions, property decisions, and executive decisions (KRUCHTEN, 2004).

Existence decisions declare that some element will be a part of the system (positively), or that some element will not be included in the system (negatively, or a non-existence decision). Property decisions describe qualities that the system must present (or, negatively, not present). Executive decisions do not refer to the system design itself, but instead are related to the process of designing the system. For instance, decisions about process steps, team size and tools to be used. This latter kind of decision is not addressed by the MULAS architectural design process. The other positive decisions can be expressed in the Design Goal Model.

In order to capture design decisions that refine how a certain goal can be achieved, how a certain task can be performed, and how a given quality constraint can be satisfied, we propose the inclusion of *design tasks* and *design constraints* in the goal model (PIMENTEL et al., 2014). As argued in BOER; VLIET (2009), there is a large similarity between architecturally significant requirements and design decisions. Nonetheless, we opt to differentiate design elements from their requirements cousins in order to make it clear, among other things, (i) who (stakeholders or designers) is responsible for making decisions with respect to those elements, and (ii) in which stage of the project they appear. This differentiation is done by using dashed borders for the design elements (e.g., *design tasks* and *design constraints* in Fig. 4.3).

Design tasks and *design constraints* are included in the goal model through AND/OR refinements. By including these elements in the goal model, rather than using a separate design decisions model, we can take advantage of existing goal-reasoning infrastructure. For instance, MULAS currently adopts the Zanshin framework (SOUZA et al., 2013) to perform a feedback loop over goal models. In future work, the MULAS framework could be extended to benefit from capabilities provided by other goal-based frameworks, such as the ability to adapt a running system to specific contexts (LAPOUCHNIAN; MYLOPOULOS, 2009), to calculate metrics (FRANCH, 2009; PIMENTEL; FRANCH; CASTRO, 2011), to generate components & connectors models (CASTRO et al., 2012; SOARES et al., 2012), and so on.

4.2.1.1 Example

In Fig. 4.3 we present the additional elements of the design goal model for the Meeting Scheduler system, which will be explained in the remainder of this subsection. Each refinement in the model is the result of a design decision, and as such may refer to the system as a whole or to a particular aspect of the system. For instance, the *Persistence* design constraint, which refines onto *Files* or *Database*, which is then refined onto *MySQL* or *NoSQL*, refers to the persistence mechanism to be used by the system as a whole. On the other hand, the *Use Web Services* constraint refers to a particular aspect of the system: automatic scheduling.

Another design constraint of the system is the decision to develop a *Client-Server System*, aiming to satisfy the *System Accessible via PC and Smartphones* constraint. That constraint

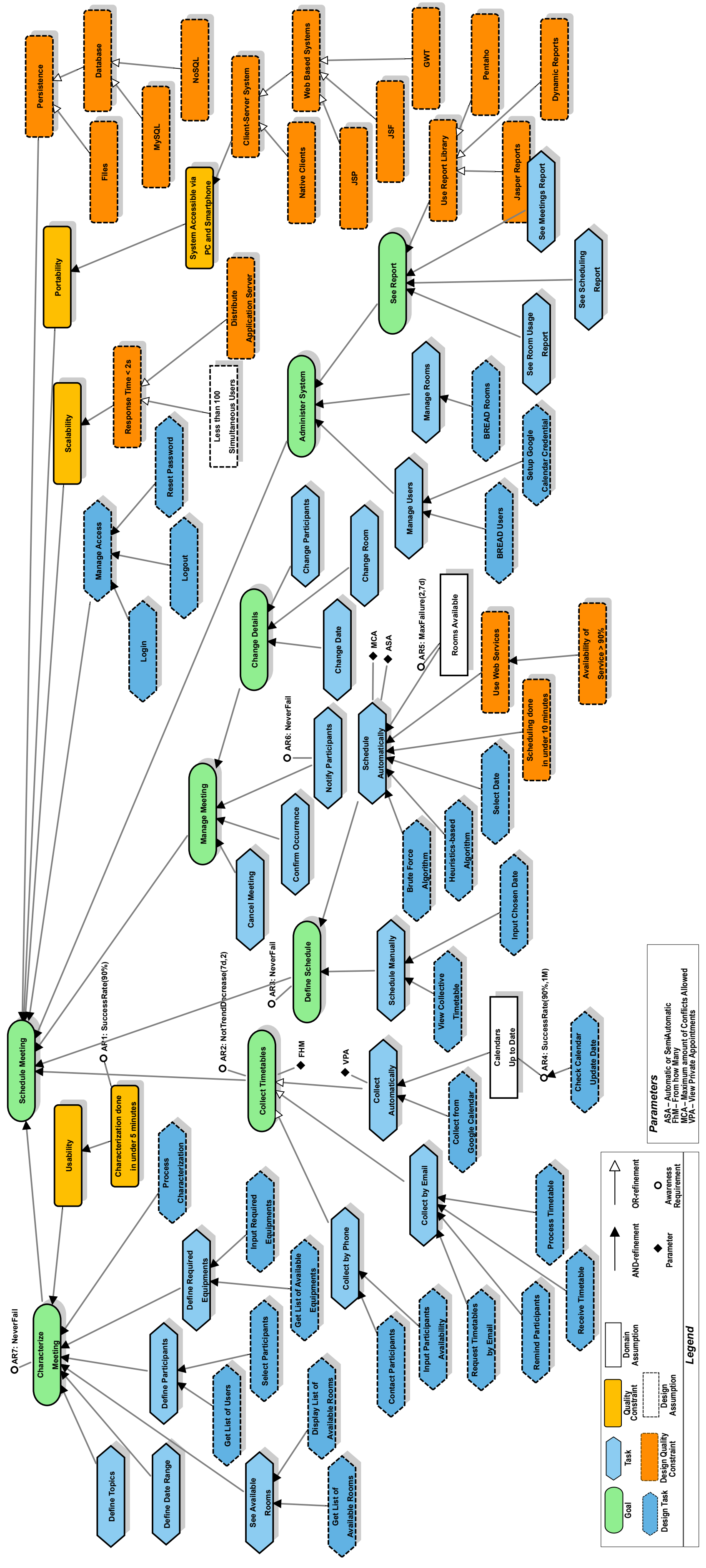


Figure 4.3: Meeting Scheduler Design Goal Model, with Design Tasks and Design Constraints

may be satisfied either by developing *Native Clients* for each platform, such as Windows, Android, and iOS, or by developing a *Web Based System* accessible through Internet browsers. Different frameworks can be used to develop a web based system, such as JavaServer Pages (JSP), JavaServer Faces (JSF), and Google Web Toolkit (GWT).

Scalability was defined as the following design constraint: *Response Time < 2s*. It was assumed that if the system has less than 100 simultaneous users (design assumption), that design constraint would be satisfied. Otherwise, it is necessary to *Distribute Application Server*.

Regarding the *See Report* goal, it was decided to user a *Report Library*, choosing between the following options: *Jasper Reports*, *Dynamic Reports*, and *Pentaho*. Lastly, the *Scheduling done in under 10 minutes* was included with respect to automatic scheduling in order to prevent long executions of the scheduling algorithms.

Design tasks provides additional information on how the requirement tasks can be enacted. The *See Available Rooms*, *Define Participants* and *Define Required Equipments* tasks have similar design refinements, related to the need to obtain information from the system: *Get List of Available Rooms*, *Get List of Users* and *Get List of Available Equipments*. Furthermore, in order to enable the user to *See Available Rooms*, the system needs to *Display List of Available Rooms*. Similarly, in order to support the *Define Participants* task, the system needs to provide the option to *Select Participants*, while to *Define Required Equipments* it is necessary to *Input Required Equipments*.

The collection of timetables can be achieved through three alternatives: *Collect by Phone*, *Collect by Email*, and *Collect Automatically*. In order to *Collect Participants* it is necessary to *Contact Participants*, i.e., to actually make phone calls to the people who were invited for the meeting. The *Input Participants Availability* design task was also deemed necessary. These two design tasks result from the decision to delegate the actual calls for a user of the system, who would then input the required information into the system. This delegation will be made explicit in the next step of the process: *Assign tasks*.

Continuing on the topic of timetable collection, another possibility is to *Collect by Email*. In order to perform this task, the following design tasks were devised: *Request Timetables by Email*, i.e., send emails to each person invited for a meeting with information about the meeting and instructions for sending their timetables; considering that some participants may forget to send their timetables, the *Remind Participants* design task was also included in the design goal model; lastly, since the timetables will be sent by email, the system will need to *Receive Timetables* and subsequently *Process Timetable*; .

The last alternative for achieving the *Collect Timetables* goal is to *Collect Automatically*. Instead of developing calendar management functionalities, it was decided to import the timetable data from a third-party system, in our example the Google Calendar, with the assumption that *Calendars Are Up to Date*.

Concerning the *Define Schedule* goal, the stakeholders required that the system must provide both the options to *Schedule Manually* and to *Schedule Automatically*. In order to

Schedule Manually, a user will *View Collective Timetable*, i.e., the joint timetable of all invited participants, and then *Input Chosen Date*. For the automatic scheduling, it was decided to provide two different scheduling algorithms: a simple *Brute Force Algorithm* that tries all the possibilities, and a *Heuristics-based Algorithm*.

Another set of design tasks included in the design goal model is contained within the *Administer System* sub tree. The *BREAD Users* and *BREAD Rooms* design tasks are refinements, respectively, of the *Manage Users* and *Manage Rooms* tasks. BREAD is an acronym for browse, read, edit, add, and delete (STOLZE et al., 2007), which are the basic operations for creating and maintaining data about a certain entity in software systems. Furthermore, user management also includes the *Setup Google Calendar Credentials* design task, in order to be able to access the Google Calendar data from each user.

Lastly, the *Manage Access* design task is refined with the following design tasks: *Login*, *Logout*, and *Reset Password*. These are examples of tasks that are not required by the stakeholders, since they are not related to the main goal of *Schedule Meeting*, but that are relevant to consider in terms of architectural design.

Besides deciding what tasks need to be performed, it is also necessary to decide *who* or *what* is going to execute them — i.e., tasks can be delegated not only to a person, but also to an organization or to a software system, among others. For this reason, the MULAS architectural design process includes an *Assign Tasks* step, described in the following subsection.

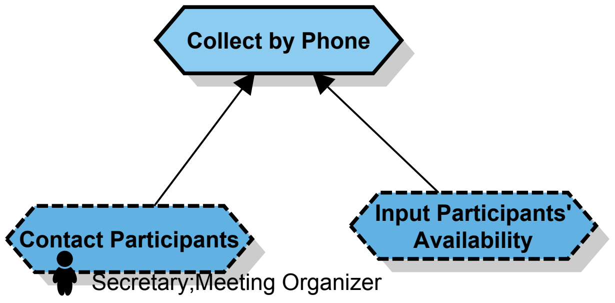
4.2.2 Assign Tasks

The design goal model presents the tasks that need to be performed by the system-to-be, but does not prescribe who is responsible for performing them: human actors, organizations, software systems, etc. This step of the process consists of defining such responsibilities, which is particularly relevant in the context of socio-technical systems, where human actors are not mere users that interact with a technical system, being instead contributing components of the system (EMERY, 1959).

The importance of different kinds of actors is acknowledged in the *i** modeling language and framework (YU et al., 2011), where not only actors are explicitly represented but also their specializations: agent, position, and role. This explicit representation is helpful in the early requirements phase, allowing to represent the needs and capabilities of each actor. In late requirements and architecture we are focused on the system itself, thus the notion of actors is not included in the design goal model. However, considering the broader picture of socio-technical systems, it is still relevant to be able to represent the delegation of certain tasks to human actors, organizations, or even other (possibly already existing) software systems.

The *Assign tasks* step takes a design goal model as input, resulting in an updated design goal model with assignments as output. A task or design task may be assigned to one or more actors, with the meaning that it may be performed by any one of them. If, instead, part of the

Figure 4.4: Example of assignment in the Meeting Scheduler system



task will be performed by an actor and another part by another actor, the original task can be further refined with sub-tasks assigned to different actors.

Assignments are expressed by labels below the assigned element. The left side of the label show the icon of a person, to represent the assignment. The actors to whom the task are assigned to are listed to the right of the icon, as shown in Fig. 4.4.

4.2.2.1 Example

In our running example, the *Contact participants (by phone)* task may be performed either by a secretary or by the meeting organizer (Fig. 4.4). All the other tasks are either performed or assisted by the system-to-be. This assignment was chosen since developing the capability of making automatic phone calls and collecting timetables would be too costly. In order to make this kind of decision, it may be necessary to consult with the project stakeholders in order to find the most beneficial option.

As seen in this step, as well as in the previous step, there are different ways to perform a task. For instance, a single task may be refined with different design tasks. Furthermore, tasks may be executed by different actors, including social actors and other software/hardware systems. The next step is concerned with yet another way on which the execution of tasks may vary: their execution flow.

4.2.3 Define Basic Flows

In this step we define the order of goal fulfillment and task execution, through the use of flow expressions. This is done by, for each element that will be refined, defining a flow expression which describes the behavior of its children elements. For instance, if a hypothetical goal *g1* is annotated with the following flow expression

$$t2\ (t3|t4)\ t5 \tag{4.1}$$

it means that whenever *g1* is to be fulfilled, the *t2* task needs to be executed, followed by *t3* or *t4*, and lastly *t5*. Fig. 4.5 shows an example of a flow expression on an excerpt of the Meeting Scheduler DGM. There, the *See Available Rooms* task (*t9*) is annotated with the following flow expression:

$$(dt37\ dt38) \tag{4.2}$$

This annotation means that the execution of *t9* corresponds to the execution of *Get List of Available Rooms* (*dt37*), followed by the execution of *Display List of Available Rooms* (*dt38*).

The symbols that can be used in flow expressions are informally presented in Table 4.1, while a formal grammar for such expressions is described in Section 5.2. By writing the flow of each sub-tree in the goal model, instead of considering the system as a whole at once, it becomes easier to define the system flow. Nevertheless, while the goal model can facilitate the definition of the system's behavior, it is necessary to consider the following concerns: (i) the meaning of the AND/OR goal refinements, (ii) variations on structure, and (iii) intermediate states, as follows.

Figure 4.5: Example of behavioral annotation (flow expression) on a DGM

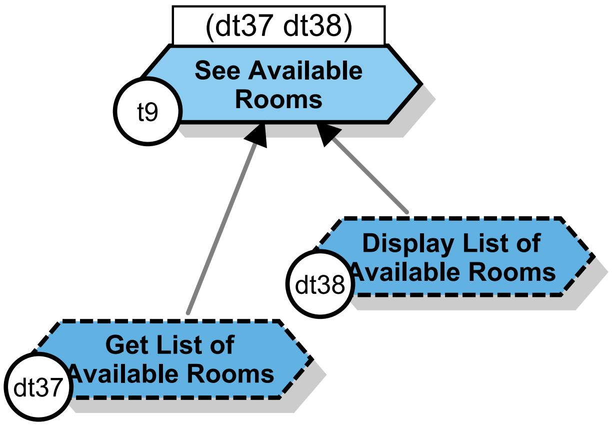


Table 4.1: Symbols adopted in the flow expressions

Symbol	Meaning	Example
blank space	Sequence	`(t1 t2)`, first *t1* and then *t2*
|	Alternative	`(t1|t2)`, *t1* xor *t2*
?	Optional	`(t1 t2? t3)`, first *t1* and then *t3*, or first *t1* followed by *t2* and *t3*
⋆	Zero or more times	`(t1 t2* t3)`, first *t1*, then *t2* zero or more times, then *t3*
+	One or more times	`(t1 t2+)`, first *t1*, then *t2* one or more times
−	Concurrency	`(t1-t2)`, *t1* and *t2* are executed concurrently

line break	Alternative flows	`(t1 t2)`
		`(t2 t1)`
		, *t1* and then *t2*, or *t2* and then *t1*

Before defining the flow expressions it is important to reconsider AND refinements present in the original model. At the requirements level, an AND refinement means that the system must support functionality for fulfilling all the children elements of that refinement. At runtime, though, this does not mean that all of these children need to be executed every time in order to achieve the parent goal. For instance, consider the *Manage Meeting* goal in the Meeting Scheduler system (Fig. 4.1) and two of its sub-tasks: *Cancel Meeting* and *Confirm Occurrence*. This AND refinement states that both options must be supported by the system. Nonetheless, for any meeting only one of these tasks would be executed, never both, since it is not possible to confirm the occurrence of a canceled meeting. Thus, the flow expression containing these tasks will present them as alternatives, even though this may seem counter-intuitive: *Cancel Meeting* | *Confirm Occurrence*.

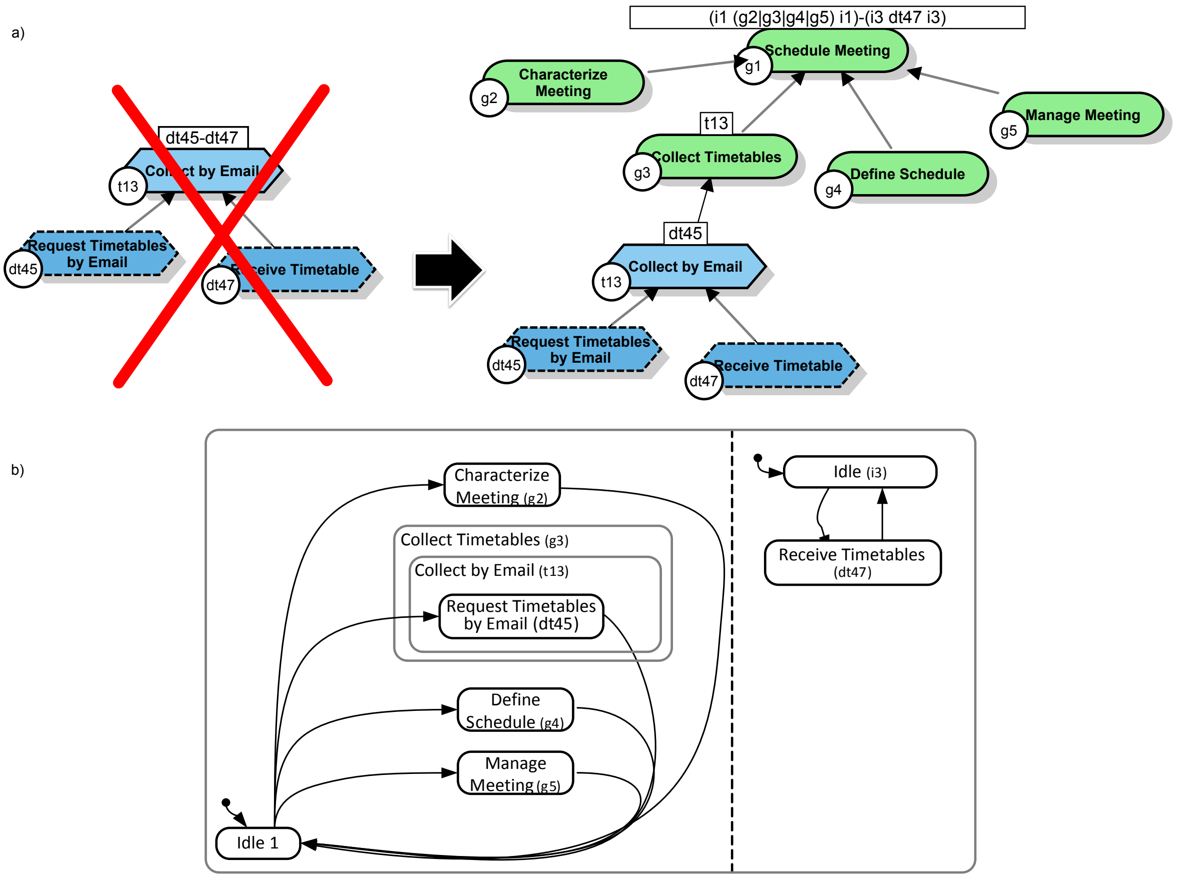
Another concern is related to variations on structure, when the flow expression of a given node refer to nodes that are not its direct children. This is the case, for instance, of the excerpt shown on Fig. 4.6-A). When trying to arrange a meeting, a user may select to collect timetables by email. When that option is selected, the system should send an Email to meeting participants (*Request Timetables by Email, dt45*). In order to collect timetables, it is not sufficient to just request timetables; it is also necessary to receive them. However, replies from participants may be sent at any time. Thus, it is necessary to be able to *Receive Timetable* at any time, regardless of whatever other interaction may be happening on the system. Since the act of receiving timetables is independent from the remainder of the system, it can be represented as a concurrent flow.

At first glance, one may consider the possibility of expressing this concurrency as shown on the left-hand side of Fig. 4.6-A: *dt45-dt47* (i.e., *dt45* is concurrent with *dt45*). However, *dt47* (textitReceive Timetable) is concurrent not only to *dt45* (*Request Timetables by Email*), but to the system as whole. Thus, this concurrency must be stated on the root goal, as shown on the right-hand side of Fig. 4.6-A: *(i1 (g2|g3|g4|g5) i1)-(i3 dt47 i3)*. The statechart corresponding to this flow is shown on Fig. 4.6-B. From an idle state (*Idle 1, i1*), a user may *Characterize Meeting (g2)*, *Collect Timetables (g3)*, *Define Schedule (g4)*, or *Manage Meeting (g5)*, always returning to the idle state (*i1*). Concurrently to that, the system may *Receive Timetables*.

Lastly, a common practice when creating statecharts is to use intermediate states as a point where the system is idle, waiting for some input, e.g., waiting for a selection by the user. One possible way to include this kind of state here would be to include in the design goal model a design task representing the intermediate state — e.g., *Select Option* — and then include that design task in the flow expression. However, considering how frequently these states appear, and aiming to reduce visual pollution in the design goal model, we decided to support the inclusion of such states directly in the flow expressions. Thus, intermediate states can be specified by inserting them in a flow expression, identified as *iX*, where *X* is any integer.

Considering that there are different ways for a system to perform a set of tasks, determining the behavioral refinement (through flow expressions) is not a matter of direct translation,

Figure 4.6: Example of variation on structure on an excerpt of the Meeting Scheduler system. A) Design Goal Model; B) Statechart

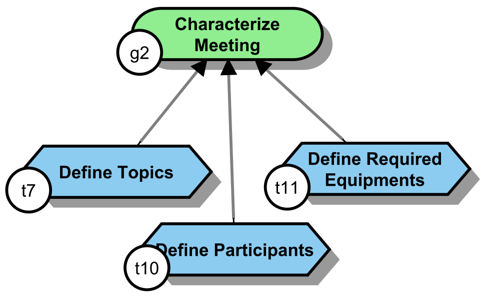


but it rather constitutes an important design decision. It is, thus, influenced by non-functional requirements, reuse of components, previous decisions (for instance, regarding the architectural style for the system structure), among other factors. For instance, we are going to consider the *Characterize Meeting* goal (*g2*). The goal model dictates that to be able to *Characterize Meeting*, the system must provide the following capabilities: *Define Topics* (*t7*), *Define Participants* (*t10*) and *Define Required Equipments* (*t11*). There are different ways of performing these tasks, four of which are represented in Fig. 4.7.

The first option is to execute them in sequence —- first *Define Topics* (*t7*), then *Define Participants* (*t10*), and lastly *Define Required Equipments* (*t11*). The second option expresses a different sequence, where *Define Participants* is performed before *Define Topics*. The third option is the same sequence of the second one, but with the execution of *Define Participants* and *Define Required Equipments* flagged as optional. Lastly, option four presents options one and three separated by a line break, meaning that they are alternative behaviors — i.e., both are valid behaviors of the system.

This kind of decision-making has to be performed for every node in the Design Goal Model. Different strategies can be adopted: top-down, bottom-up, and middle-out. Based on our experience, and on informal feedback received from students during an experiment (which is described on Chapter 7), top-down seems to be the most difficult strategy. This is likely due to the fact that the flow expression for root goals is usually more complex than the flow expression of other goals. On the other hand, middle-out and bottom-up are both good options, since the flow of elements at lower levels are usually easier to define. Since the algorithms described on Chapter 5 support the statechart derivation of partially annotated models, software designers can proceed iteratively and incrementally: annotate an excerpt of the model, generate statechart, analyze the resulting statechart, then go back to the design goal model, modify or expand its flow expressions, and repeat the cycle. Moreover, simulation tools[2] can be used to help ascertaining whether the specified behavior is correct.

Figure 4.7: Possible flows for the *Characterize meeting* goal



```
Option 1) t7  t10  t11
Option 2) t10  t7  t11
Option 3) t10  t7?  t11?
Option 4) t7  t10  t11
          t10  t7?  t11?
```

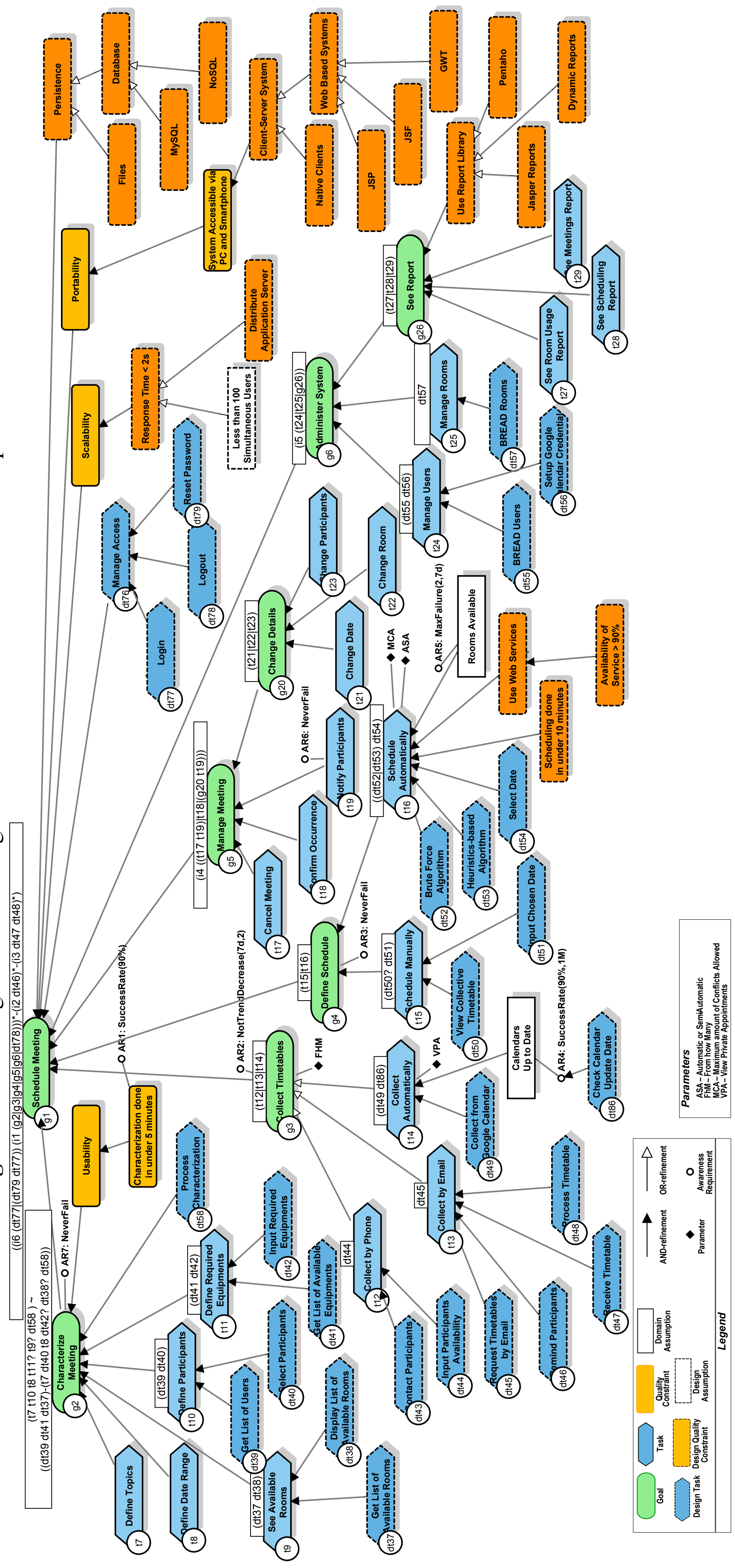


Figure 4.8: Meeting Scheduler Design Goal Model annotated with Flow Expressions

4.2.3.1 Example

The resulting flow expressions for the Meeting Scheduler system are depicted in Fig. 4.8. The flow expression for the *Characterize Meeting* goal, considering all of its sub-tasks, was defined as follows:

$$\begin{gathered}(t7\ t10\ t8\ t11?\ t9?\ dt58)\\((dt39\ dt41\ dt37) - (t7\ dt40\ t8\ dt42?\ dt38?\ dt58))\end{gathered} \tag{4.3}$$

The line break in this expression shows that there are two valid behaviors. The first one is a sequence: first execute *Define Topics* (*t7*), then *Define Participants* (*t10*), then *Define Date Range* (*t8*), then optionally execute *Define Required Equipments* (*t11*), then optionally execute *See Available Rooms* (*t9*), and lastly *Process Characterization* (*dt58*). The flows of the tasks *t9*, *t10* and *t11*, in their turn, are simple sequential executions of their respective children, as can be seen in Fig. 4.8.

The second behavior for the *Characterize Meeting* goal (flow expression 4.3) is composed by two concurrent flows. The first concurrent flow executes a sequence of design tasks: *Get List of Users* (*dt39*), *Get List of Available Equipments* (*dt41*), and *Get List of Available Rooms* (*dt37*). The second concurrent flow is a sequence similar to the first alternative behavior: first execute *Define Topics* (*t7*), then *Select Participants* (*dt40*), then *Define Date Range* (*t8*), then optionally execute *Input Required Equipments* (*dt42*), then optionally execute *Display List of Available Rooms* (*dt38*), and lastly *Process Characterization* (*dt58*). The idea is that the required lists are being loaded while the user is interacting with the system.

The flow expression for the *Collect Timetables* goal (*g3*) is a simple set of alternative choices — either *Collect by Phone* (*t12*), *Collect by Email* (*t13*), or *Collect Automatically* (*t14*), as defined by the flow expression 4.4. The selection of which alternative to execute depends on user requests, as it is going to be defined in the *Specify transitions* step of the architectural design process.

$$(t12|t13|t14) \tag{4.4}$$

The flow of the *Define Schedule* goal (flow expression 4.5) is defined as a choice between *Schedule Manually* (*t15*) and *Schedule Automatically* (*t16*). In its turn, *Schedule Manually* (flow expression 4.6) is executed by optionally executing *View Collective Timetable* (*dt50*), and then *Input Chosen Date* (*dt51*). The flow of *Schedule Automatically* (flow expression 4.7), is defined as a choice between *Brute Force Algorithm* (*dt52*) and *Heuristics-based Algorithm* (*dt53*), followed by *Select Date* (*dt54*).

$$(t15|t16) \tag{4.5}$$

$$(dt50?\ dt51) \tag{4.6}$$

[2]The ATM case study on Chapter 6 describes the use of a simulation tool for verifying the system's behavior.

$$((dt52|dt53)\ dt54) \tag{4.7}$$

Different intermediate states were defined in the Meeting Scheduler system. For instance, in the flow expression of the *Manage Meeting* goal (flow expression 4.8) an intermediate state *i4* is used to let the user decide which comes next: *Cancel Meeting* (*t17*) and *Notify Participants* (*t19*), *Confirm Occurrence* (*t18*), or *Change Details* (*g20*) and *Notify Participants* (*t19*). Similarly, the flow for the *Administer System* goal (flow expression 4.9) has an intermediate state *i5* before a choice between *Manage Users* (*t24*), *Manage Rooms* (*t25*), and *See Report* (*g26*).

$$(i4\ ((t17\ t19)|t18|(g20\ t19))) \tag{4.8}$$

$$(i5\ (t24|t25|g26)) \tag{4.9}$$

The flow expression of the root goal of the Meeting Scheduler system is defined as follows:

$$((i6\ (dt77|(dt79\ dt77))\ (i1\ (g2|g3|g4|g5|g6|dt78)))* - (i2\ dt46)* - (i3\ dt47\ dt48)*) \tag{4.10}$$

This flow contains three concurrent flows: the first one is the main interaction with the system, where the user characterize meetings, define schedules, manage meetings, and so on (flow expression 4.11). The second one, flow expression 4.12, represents the periodical sending of reminders by the system — at certain points in time, the system will *Remind Participants* (*dt46*). The third flow, also executing concurrently to the remainder of the system, refers to the ability to receive (*dt47*) and process (*dt48*) the timetables sent by email, as stated by the flow expression 4.13.

$$(i6\ (dt77|(dt79\ dt77))\ (i1\ (g2|g3|g4|g5|g6|dt78)))* \tag{4.11}$$

$$(i2\ dt46)* \tag{4.12}$$

$$(i3\ dt47\ dt48)* \tag{4.13}$$

According to the flow expression 4.11, the system starts in an idle state (*i6*), from which it is possible to *Login* directly (*dt77*), or to *Reset Password* (*dt79*) before logging in. Then, from the intermediate state *i1*, it is possible to *Characterize Meeting* (*g2*), to *Collect Timetables* (*g3*), to *Define Schedule* (*g4*), to *Manage Meeting* (*g5*), to *Administer System* (*g6*), or to *Logout* (*dt78*), where the flows pertaining to these goals have already been described.

Although the definition of flow expressions for a system may seem to be a cumbersome task, it is made easier by the ability of generating partial statecharts from partially refined design goal models. In other words, it is not necessary to define the behavioral refinement of the entire model in order to visualize the resulting statechart, which is consonant with the Twin Peaks notion of iteratively and incrementally refining requirements and architecture (NUSEIBEH, 2001). Moreover, it is not required to include every node within the flow expressions. For instance, the *Contact Participants* task (dt43) was not included in any flow expression — thus, not included in the system's behavior — since it has been previously assigned to human actors (see Fig. 4.4).

The three steps of the architectural design process presented so far are generic; they are not specific to any particular class of systems: *Identify Design Tasks, Constraints, and Assumptions*, *Assign Tasks*, and *Define Basic Flows*. The next two steps are related to adaptation specification, hence only necessary when designing adaptive systems. They are: *Identify indicators, parameters and relations* and *Specify adaptation strategies*.

4.2.4 Identify indicators, parameters and relations

In this step the design goal model is enriched with additional awareness requirements and parameters, referring to the elements that were included during the previous architectural design steps: design tasks, design constraints, design assumptions, assignments, and flow expressions. When an awareness requirement is attached to an element in a goal model, the implications are twofold: (i) that element will be monitored at runtime; (ii) it will be possible to react to failures related to that element.

The following elements can be analyzed to help the elicitation of awareness requirements: critical requirements; critical design tasks, constraints, or assumptions; preferable solutions; trade-offs; tasks that are difficult to test offline; functionalities provided by third-parties; transition triggers; transition conditions; preemptive adaptation; other awareness requirements (meta-awareness); qualitative elicitation.

Besides awareness requirements, which indicate what needs to be monitored, it is necessary to define the system parameters — i.e., what can be modified in the system. In order to accommodate the new alternatives identified throughout the architectural design process, we may create new parameters that can be modified by adaptation mechanisms when a reconfiguration of the system is required. Parameters related to design tasks, design constraints and design assumptions are equivalent to parameters related to their requirements counterparts: tasks, quality constraints and domain assumptions, respectively. Additionally, in the context of statecharts, parameters may refer to (i) the selection of alternative behaviors; (ii) the definition of parameterized events; and (iii) the definition of parameterized conditions.

4.2.4.1 Example

Considering those design elements which need to be monitored and which failures are strongly undesired, three additional awareness requirements were defined for the Meeting Scheduler system:

- **AR8: NeverFail** — Design task *Login* should never fail. This design task is critical because, if it is not completed properly, users will not be able to access the system's functionalities.

- **AR9: MaxFailure(5,7d)** — Design constraint *Response Time < 2s* should fail at most twice every week. If this constraint is not satisfied it is necessary to analyze whether these failures were due to some occasional hiccup, or if evolutive maintenance is due in order to prevent further slowdowns.

- **AR10: NotTrendDecrease(1d,2)** — The success rate of the design constraint *Availability of Service > 90%* should not decrease two days in a row. This constraint will be monitored because services are provided by third-party companies, and thus cannot be trusted completely.

Additionally, four new parameters were created based on the new refinements of the Design Goal Model included throughout the architectural design process, as follows:

- **Pre - Preload —** This parameter defines which of the alternative behaviors of the *Characterize Meeting* goal will be executed (flow expression 4.3): the regular one (first alternative behavior) or the second one, with preloading of its different lists (second alternative behavior). **Range:** *WithoutPreload* or *WithPreload*.

- **SeS - Selected Service —** To exemplify a scenario of service selection (FRANCH et al., 2011), we are assuming that different web service providers can be used to perform the functionality of scheduling automatically. Instead of selecting a single service, the selected service was defined as a parameter, which can thus be modified throughout the system execution. **Range:** *Service1*, or *Service2*, or *Service3*, or *Service4*.

- **TIR – Time Interval between Reminders —** As described in Section 4.2.3, the Meeting Scheduler system sends periodic reminders to its users, with the view of collecting their timetables (flow expression 4.12). Instead of pre-defining a fixed time interval for these reminders, it will be dynamically defined by this parameter. Hence, it will be possible to change this interval at runtime, aiming to find an interval which improves user response. **Range:** from *1* to *336*, in hours.

- **Sce - Scheduling algorithm —** As there are different algorithms that can be used to perform the scheduling automatically, with different tradeoffs between performance and number of conflicts, this parameter will allow to choose between them. **Range:** *HeuristicsBasedAlgorithm* or *BruteForceAlgorithm*.

The DGM with these additional parameter highlighted with dashed circles is presented in Fig. 4.9. The first of these parameters, *Pre* is related to alternative behaviors —- different ways of executing the same set of tasks. The second one, *SeS*, specifies which service will be used to provide the *Schedule Automatically* functionality. The third one, *TIR*, will be used to define the trigger event for the *Remind Participants* task (Section 4.2.7). Lastly, the *Sce* parameter is used to select which algorithm to use for the *Schedule Automatically* task.

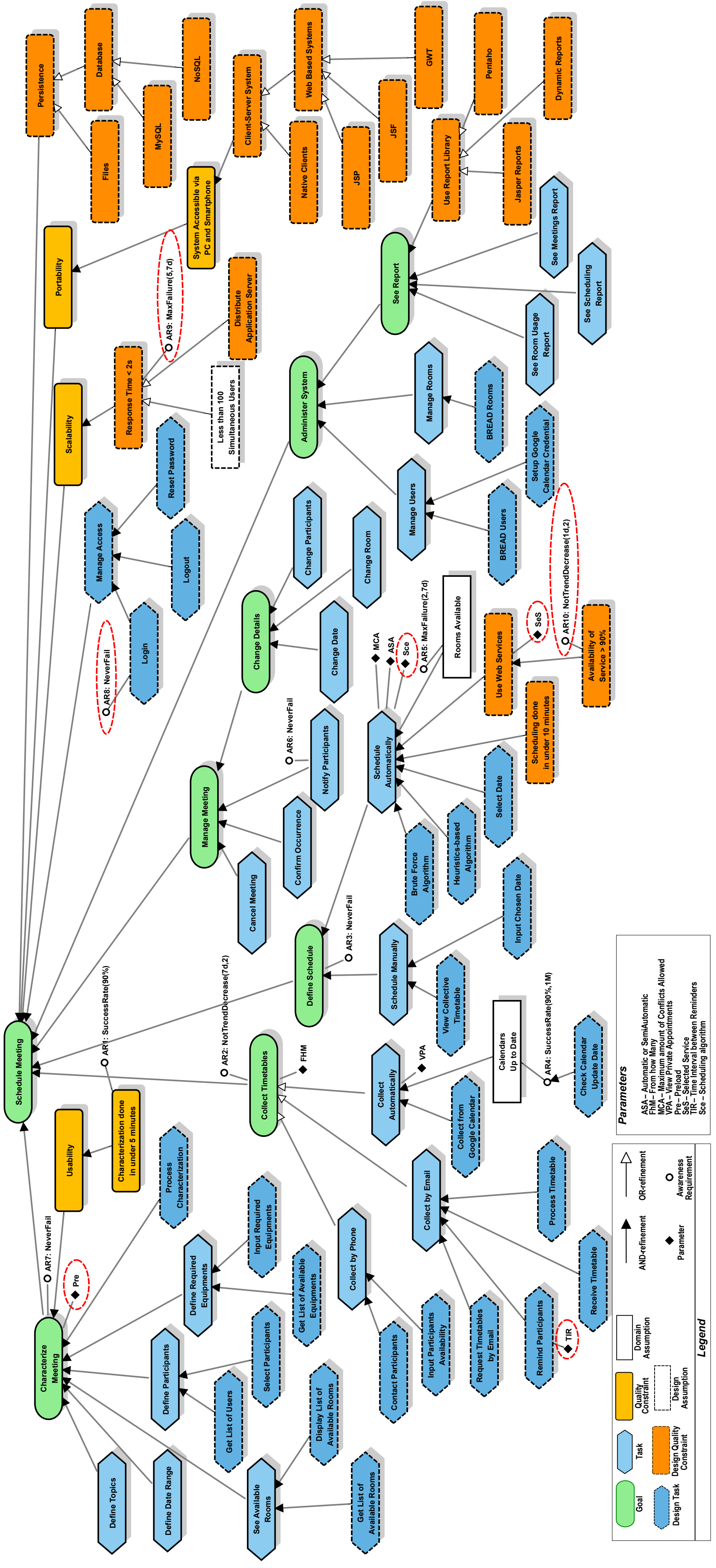


Figure 4.9: Design Goal Model of a Meeting Scheduler system with additional adaptation elements (highlighted with dashed circles)

After defining new parameters and awareness requirements, the architect must also define new differential relations that describe the impact of parameters on each awareness requirement. For our running example, we identified that **Pre** is related to *Characterization done in under 5 minutes* (**AR1**, Eq. 4.14). The first value in an enumeration is its default value. Therefore, the default value of **Pre** is *WithoutPreload*. If its value is changed to *WithPreload*, the *Characterization done in under 5 minutes* constraint is more likely to be satisfied.

$$\Delta(AR1/Pre) > 0 \tag{4.14}$$

TIR is inversely proportional to **AR2** (Eq. 4.15), in a range from 1 to 336 hours (two weeks). **AR2** refers to *Collect Timetables*. Thus, the smaller the value of **TIR** — i.e., the more frequently reminders are sent — the more likely it is that the timetables will be collected successfully.

$$\Delta(AR2/TIR) < 0 \tag{4.15}$$

The scheduling algorithm to be used (**Sce**) influences **AR3** (Eq. 4.16), which is related to the *Define Schedule* goal. The default value for this parameter is *HeuristicsBasedAlgorithm*. If the *Define Schedule* goal fails, it is possible to switch to *BruteForceAlgorithm*, which is slower but provides better results.

$$\Delta(AR3/Sce) > 0 \tag{4.16}$$

Lastly, the **SeS** parameter affects the satisfaction of the *Availability of Service > 90%* constraint (**AR10**, Eq. 4.17). If the currently selected service has poor availability, another service may be selected.

$$\Delta(AR10/SeS) > 0 \tag{4.17}$$

In summary, whether the meeting characterization is performed with preloading impacts the success of performing the characterization in less than five minutes; higher intervals between sending reminders decrease the likelihood of successfully collecting timetables; the scheduling algorithm to be used influences the successful achievement of the *Define Schedule* goal; and the selected service affects the satisfaction of the *Availability of Service > 90%* design constraint.

The complete sets of awareness requirements and adaptation parameters for the Meeting Scheduler system are presented in Table 4.2 and Table 4.3, respectively. These tables contain the awareness requirements and parameters related both to the requirements elements, described in Section 4.1, and those related to the design elements, introduced in this section. The next step is concerned with defining how to react in case those awareness requirements are not satisfied.

4.2.5 Specify adaptation strategies

Adaptation strategies define what should happen once an awareness requirement has failed. One of the strategies provided by Zanshin is the reconfiguration, where Zanshin itself will decide which parameters to change based on the relations between awareness requirements and parameters (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012b).

Besides reconfiguration, Zanshin provides a set of strategy patterns that represent possible adaptations at a higher abstraction level, such as relaxing the constraints imposed by an awareness requirements and ignoring a child element when analyzing the satisfaction of its parent (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012b). Since the MULAS framework is concerned not only with the system requirements but also with its behavior, we have extended Zanshin's adaptation strategy patterns, as follows.

- **Abort —** Give up on satisfying a given awareness requirement and move on to a specific state. This can be used to go back to a specific system menu, or even to shut down the entire system.
- **Abort With Action —** Similar to *Abort*, but with a set of intermediate actions to allow the graceful failure of the system. Examples of possible actions are to show an error message, to save data on a database, and to notify system administrators about the failure.
- **Delegate —** Some problems may only be solved by human actors. This pattern is about notifying the actors in charge of a certain awareness requirement, delegating the responsibility for addressing its failure. Among others, this may refer to performing maintenance on the software system.
- **Retry —** Retry the execution of a given task by going back to its respective state, after a certain delay (specified in milliseconds).
- **Step Back —** Instead of continuing with the normal flow of the system, go back to the last executed state. This strategy pattern is particularly useful when the last transition was based on a user choice, since it will give the user an opportunity for making a different choice.
- **Notify —** The system should notify some actor about the present failure.

Table 4.2: Description of the awareness requirements for the Meeting Scheduler System, both requirements- and architecture-related

Id	Awareness Requirement	Description
AR1	SuccessRate(90%)	The *Characterization done in under 5 minutes* quality constraint should have a success rate of at least 90%.
AR2	NotTrendDecrease(7d,2)	The success rate of the Collect timetables goal should not decrease two weeks in a row.
AR3	NeverFail	The *Define Schedule* goal should never fail.
AR4	SuccessRate(90%,1M)	The *Calendars Up to Date* domain assumption should have a success rate of at least 90%, measured every month.
AR5	MaxFailure(2,7d)	The *Rooms Available* domain assumption should fail at most twice every week.
AR6	NeverFail	The *Notify Participants* task should never fail.
AR7	NeverFail	The *Characterize Meeting* goal should never fail.
AR8	NeverFail	The *Login* design task should never fail.
AR9	MaxFailure(5,7d)	The *Response Time < 2s* design constraint should fail at most five times every week.
AR10	NotTrendDecrease(1d,2)	The success rate of the *Availability of Service > 90%* design constraint should not decrease two days in a row.

Table 4.3: Parameters of the Meeting Scheduler System, both requirements- and architecture-related

Id	Description	Range
ASA	Perform **Automatic or Semi-Automatic** scheduling	Enumerated [Automatic, SemiAutomatic]
FHM	Collect timetables **From How Many** people	Percentage [0%, 100%]
MCA	**Maximum amount of Conflicts Allowed** when scheduling	Integer [0, 50]
Pre	**Preload** on the meeting characterization	Enumerated [WithoutPreload, WithPreload]
SeS	**Selected Service** for automatic scheduling	Enumerated [Service1, Service2, Service3, Service4]
TIR	**Time Interval between Reminders**	Integer [1, 336]
VPA	**View Private Appointments** on users' calendars	Enumerated [NotAllowed, Allowed]
Sce	**Scheduling algorithm** selected for the Schedule Automatically task	Children [HeuristicsBasedAlgorithm, BruteForceAlgorithm]

Table 4.4: Adaptation strategies for the Meeting Scheduler system

Awareness Requirement	**Adaptation Strategy**
AR1(CharacterizationDoneInUnder5m): *SuccessRate(90%)*	1. Reconfigure()
AR2(CollectTimetables): *NotTrendDecrease(7d,2)*	1. Reconfigure()
AR3(DefineSchedule): *NeverFail*	1. Reconfigure() StepBack()
AR4(CalendarsUpToDate): *SuccessRate(90%,1M)*	1. RelaxReplace(AR4, AR4_85%) 2. Delegate(CompanyManager)
AR5(RoomsAvailable): *MaxFailure(2,7d)*	1. Notify(CompanyManager)
AR6(NotifyParticipants): *NeverFail*	1. Retry(5000)
AR7(CharacterizeMeeting): *NeverFail*	1. Retry(ProcessCharacterization, 5000)
AR8(Login): *NeverFail*	1. Retry(5000)
AR9(ResponseTime<2s): *MaxFailure(5,7d)*	1. Delegate(SoftwareArchitect)
AR10(AvailabilityOfService>90%): *NotTrendDecrease(1d,2)*	1. Reconfigure()

4.2.5.1 Example

Table 4.4 shows the adaptation strategies devised for the Meeting Scheduler system, i.e., how the system will react to each failure. For *AR1*, *AR2*, and *AR10*, the strategy is to *Reconfigure*, based on the differential relations defined on the previous step. For *AR3*, besides reconfiguring, the system also steps back, returning to a previous step. *AR6* and *AR8* adopt simple *Retry* strategies, whereas *AR7* retries by returning to a specific state: *Process Characterization*. In case of failures related to *AR5*, which refers to the number of rooms available for meetings, a *Company Manager* is notified. When *AR9* is not satisfied, the issue is delegated to a *Software Architect*. Lastly, the first reaction to failures on *AR4* is to decrease the required success rate, from 90% to 85%. If, even so, *AR4* fails again, this issue is delegated to a *Company Manager*.

In the next steps, the focus moves from Design Goal Models to Statecharts. A base statechart is generated automatically and then refined through further design decisions.

4.2.6 Generate base statechart

Considering the behavioral refinements defined in the previous steps, it is now possible to generate a base statechart model, which presents a comprehensive view of the system behavior. In order to support the automatic generation of statecharts, we define a set of derivation patterns related to the different flows that may be expressed —- sequential, alternative, and concurrent —- as well as to their optionality and multiplicity. These patterns, grounded on the graphical representation from SHAW (1978), is depicted in Fig. 4.10. The formalization of these patterns is described on Chapter 5.

The simplest case is that of the sequential tasks. There is one state to represent the execution of each task. The completion of a task leads to a transition to a state that represents the execution of the next task in the sequence. The first state is that related to the first child element, which is expressed in the statechart through the "default state" arrow. The exit transition is triggered upon completion of the last task.

For the alternative tasks, there will be transitions going from the previous state(s) to the different states that represent the alternative tasks. Thus, the selection of which alternative to perform in a given moment will be defined by the events and conditions of the different transitions.

For the concurrent tasks we use the notion of orthogonal states, which are separated in the diagram by dashed lines. Thus, whenever the system enters the super-state, it will start the execution of each one of the concurrent tasks.

The multiple (one or more) execution of a flow is expressed by a transition from the end to the beginning of the flow. The optionality is achieved by creating a transition from the beginning to the end of the flow, without entering the optional states. The case of zero or more executions is a mix between one or more executions and the optional execution.

4.2.6.1 Example

The statechart resulting from the application of those patterns is displayed in Fig. 4.11. The system has three orthogonal regions. The top-right region handles the receiving and processing of timetables, whereas the bottom-right region is concerned with sending reminders to participants. The left area starts with an idle state, from which it is possible to *Login* or to *Reset Password*. After login, the system transitions to another idle state, from which it is possible to *Characterize Meeting*, *Collect Timetables*, *Define Schedule*, *Manage Meeting*, *Administer System*, or *Logout*. After the execution of each of its respective flows, with the exception of *Logout*, the system returns to that idle state, from which the same flows can still be transitioned to.

Even though this resulting statechart expresses the valid behaviors for the system, it does not define which of the possible states should be entered at a given moment. For instance, from this base statechart we know that the system goes from an idle state (*i6*) to the *Login* state (*dt77*), but when is this transition triggered? How can we know whether the next state after *i6* is

Figure 4.10: Visual representation of patterns for deriving statecharts from flow expressions

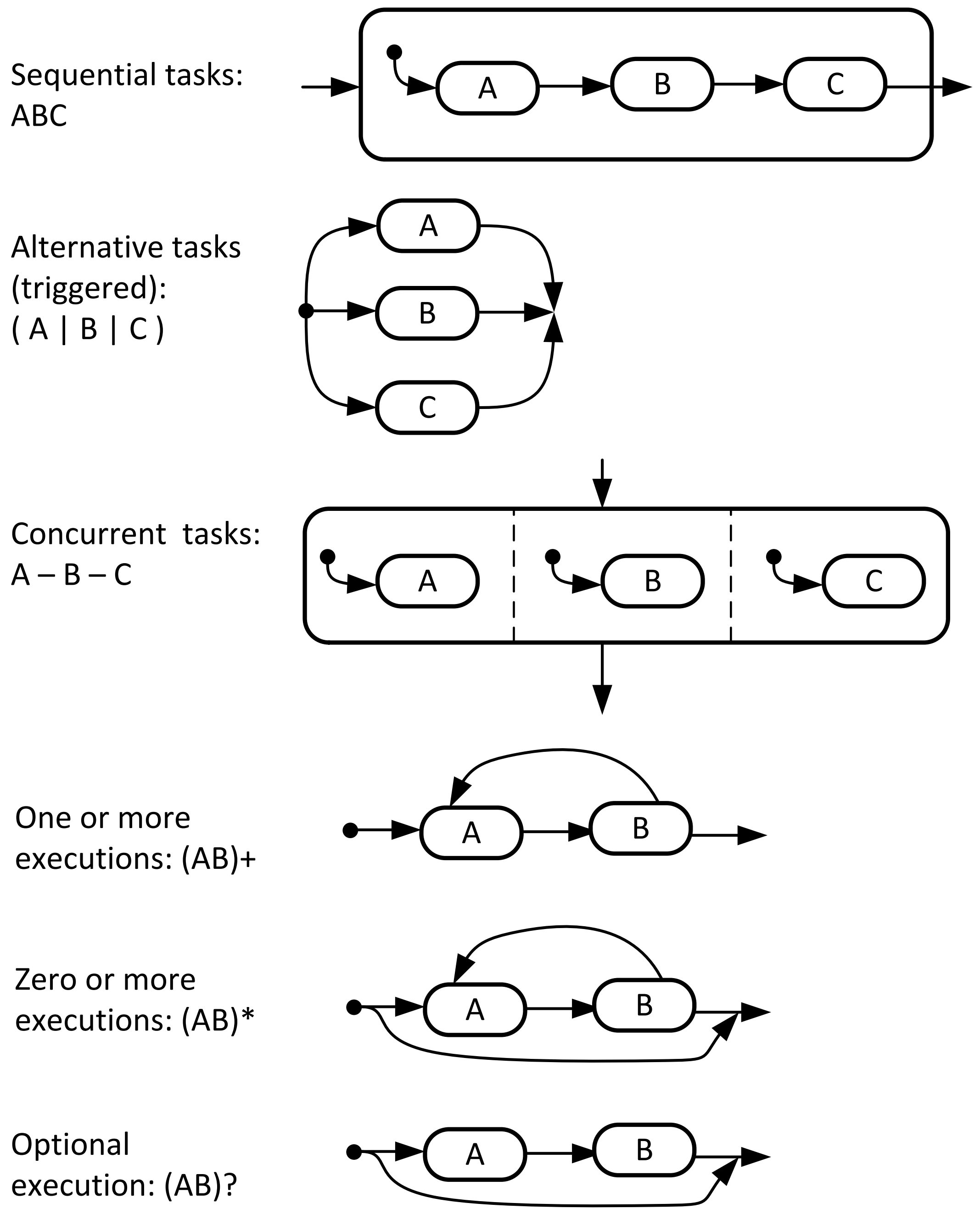
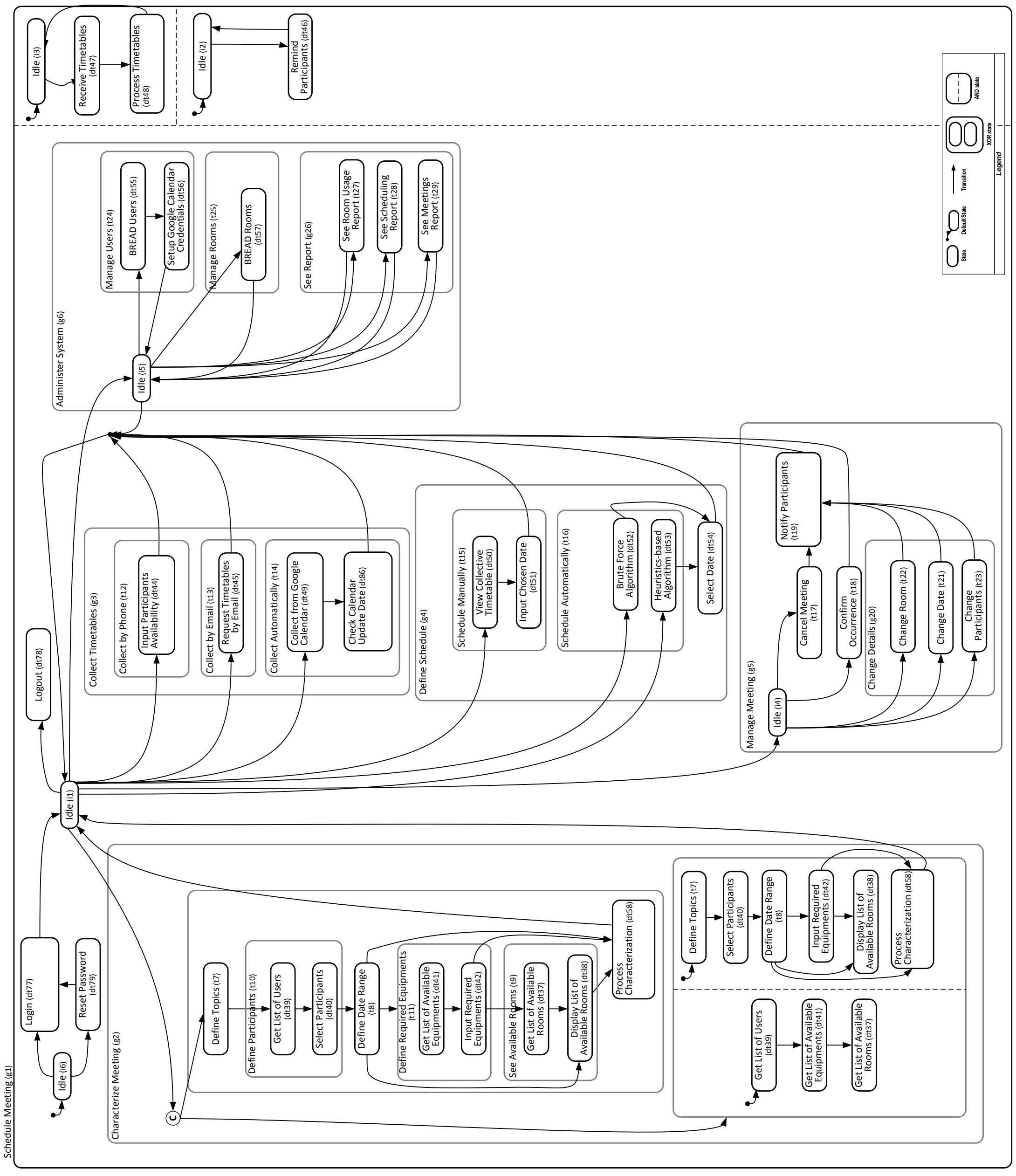

Sequential tasks: ABC
A
B
C
Alternative tasks (triggered): (A | B | C)
A
B
C
Concurrent tasks: A – B – C
A
B
C
One or more executions: (AB)+
A
B
Zero or more executions: (AB)*
A
B
Optional execution: (AB)?
A
B

Figure 4.11: Base statechart for the Meeting Scheduler system


Schedule Meeting (g1)
Idle (i6)
Login (dt77)
Reset Password (dt79)
Idle (i1)
Logout (dt78)
Characterize Meeting (g2)
Define Topics (t7)
Define Participants (t10)
Get List of Users (dt39)
Select Participants (dt40)
Define Date Range (t8)
Define Required Equipments (t11)
Get List of Available Equipments (dt41)
Input Required Equipments (dt42)
See Available Rooms (t9)
Get List of Available Rooms (dt37)
Display List of Available Rooms (dt38)
Process Characterization (dt58)
C
Collect Timetables (g3)
Collect by Phone (t12)
Input Participants Availability (dt44)
Collect by Email (t13)
Request Timetables by Email (dt45)
Collect Automatically (t14)
Collect from Google Calendar (dt49)
Check Calendar Update Date (dt86)
Define Schedule (g4)
Schedule Manually (t15)
View Collective Timetable (dt50)
Input Chosen Date (dt51)
Schedule Automatically (t16)
Brute Force Algorithm (dt52)
Heuristics-based Algorithm (dt53)
Select Date (dt54)
Manage Meeting (g5)
Idle (i4)
Cancel Meeting (t17)
Confirm Occurrence (t18)
Notify Participants (t19)
Change Details (g20)
Change Room (t22)
Change Date (t21)
Change Participants (t23)
Administer System (g6)
Idle (i5)
Manage Users (t24)
BREAD Users (dt55)
Setup Google Calendar Credentials (dt56)
Manage Rooms (t25)
BREAD Rooms (dt57)
See Report (g26)
See Room Usage Report (t27)
See Scheduling Report (t28)
See Meetings Report (t29)
Idle (i3)
Receive Timetables (dt47)
Process Timetables (dt48)
Idle (i2)
Remind Participants (dt46)
State
Default State
Transition
XOR state
AND state
Legend

Login (*dt77*) or *Reset Password* (*dt79*)? This additional information will be defined in the next step, by specifying transitions in terms of their triggers and conditions.

So far we have defined the basic flow of the system, making it possible to check at runtime if the trace of task execution is valid. However, in order to define the system behavior in the face of optionality, multiplicity and alternatives, we need to know when a particular task should be triggered, which is the focus of the next step.

4.2.7 Specify transitions

In this step it is defined which events trigger a particular alternative, the execution of an optional task, as well as whether a flow should be repeated or not. Any event can be used as a trigger, but there are four particular classes of events that are likely to appear in a statechart:

- **User request**: a task is triggered by solicitation from a user — this is the case for most of the tasks of our running example. For instance, the events *Login requested* and *Reset Password request* trigger the outgoing transitions from *i6* (Table 4.5, numbers 1 and 2).
- **Timer**: a temporal condition defines when a transition may happen, e.g., at 30 minutes intervals. In the Meeting Scheduler system, this is the case of the transition from *i1* to *dt78* (Logout): *Logout requested* OR *After 4 minutes* (Table 4.5, number 4).
- **Requested by another task**: a task is triggered by the completion of another task.
- **Requested by another system**: in cases where the system interoperates with external systems, a task may be triggered by such an external system.

It is also possible to define a combination of alternative events — for instance, a given task will be executed upon user request or at 30 minutes' intervals. (Table 4.5, number 4).

Besides events, it is often useful to associate conditions to transitions. Some elements of the design goal model can guide the definition of these conditions: domain and design assumptions, quality and design constraints, as well as other tasks and design tasks that must be executed first.

Domain and design assumptions indicate hypotheses that must hold true in order to satisfy its parent elements. Thus, when creating a statechart, some assumptions may be introduced as conditions for certain transitions, such as assumptions related to time of day (certain task can only be executed at night) or to available resources (the system must enter a given state only if bandwidth is higher than 800kb/s). Quality and design constraints may also impose conditions to the execution of a given task. In this case, the task could be considered completed only if that quality constraint is satisfied. Lastly, it is also important to identify possible inter-relations between tasks, for the cases where the execution of a task excludes the execution of another one, or when a task is a pre-condition for another task. This last case is only partially expressed in flow expressions — for instance, it can be inferred from the Meeting Scheduler's base statechart (Fig. 4.11) that it is only possible to characterize a meeting after *Login* has been executed. On other cases, it is necessary to include the execution of said tasks as a condition. E.g., the *Remind Participants* task can only be executed it the *Request Timetables by Email* has been executed, as stated by transition number 22 (Table 4.5).

In order to simplify the visualization of statecharts, we have adopted a *task completed* event as being the default trigger — thus, when a transition does not present any event or condition, it means that the transition should occur whenever the task represented by the source state has been completed. For instance, the system transitions from *Define Topics* (*t7*) to *Get List of Users* (*dt39*) when the *Define Topics* task is completed (Fig. 4.12). The other triggers need to be manually defined in a table, as shown on Fig. 4.5.

We have decided to specify transitions using a table, instead of specifying them directly on the statechart, in order to facilitate model updates: whenever there are changes on a design goal model, leading to the generation of a new base statechart, it will be possible to recover the transitions specification from the table and apply it to the newer statechart.

4.2.7.1 Example

Table 4.5 presents the triggers for the Meeting Scheduler system. Triggers 1 to 3, as well as triggers 5 to 21, are simple user requests. The fourth trigger defines a combination of user request and timer — *Logout* (*dt78*) is performed when it is requested or when it has been for four minutes in the idle state (*i1*). Similarly, the twenty-second trigger defines that *Remind Participants* (*dt46*) is triggered by a user request or at every **TIR**, where **TIR** is a parameter defined in the *Identify indicators, parameters and relations* step (Time Interval between Reminders — Section 4.2.4). Lastly, the *Receive Timetables* state is triggered every time a timetable is sent (trigger 23). Fig. 4.12 shows the statechart for the Meeting Scheduler with the specified transitions.

The next step consists of modifying a statechart in order to enact the adaptation strategies defined on the *Specify adaptation strategies* step (Section 4.2.5).

Table 4.5: Transitions specification for the Meeting Scheduler system

Number	**From → To**	**Triggers and conditions**
1	$i6 \rightarrow dt77$	Login requested
2	$i6 \rightarrow dt79$	Reset Password requested
3	$i1 \rightarrow g2$	Characterize Meeting requested
4	$i1 \rightarrow dt78$	Logout requested OR After 4 minutes
5	$i1 \rightarrow dt44$	Collect by Phone requested
6	$i1 \rightarrow dt45$	Collect by Email requested
7	$i1 \rightarrow dt49$	Collect Automatically requested
8	$i1 \rightarrow dt50$	Schedule Manually requested
9	$i1 \rightarrow t16$	Schedule Automatically requested
10	$i1 \rightarrow i4$	Manage Meeting requested
11	$i4 \rightarrow t17$	Cancel meeting requested
12	$i4 \rightarrow t18$	Confirm occurrence requested
13	$i4 \rightarrow t22$	Change Room requested
14	$i4 \rightarrow t21$	Change Date requested
15	$i4 \rightarrow t23$	Change Participants requested
16	$i1 \rightarrow i5$	Administer System requested
17	$i5 \rightarrow dt55$	Manage Users requested
18	$i5 \rightarrow dt57$	Manage Rooms requested
19	$i5 \rightarrow t27$	See Room Usage Report requested
20	$i5 \rightarrow t28$	See Scheduling Report requested
21	$i5 \rightarrow t29$	See Meetings Report requested
22	$i2 \rightarrow dt46$	Remind Participants requested (Request Timetables by Email executed) OR Every **TIR** (Request Timetables by Email executed)
23	$i3 \rightarrow dt47$	Timetable sent

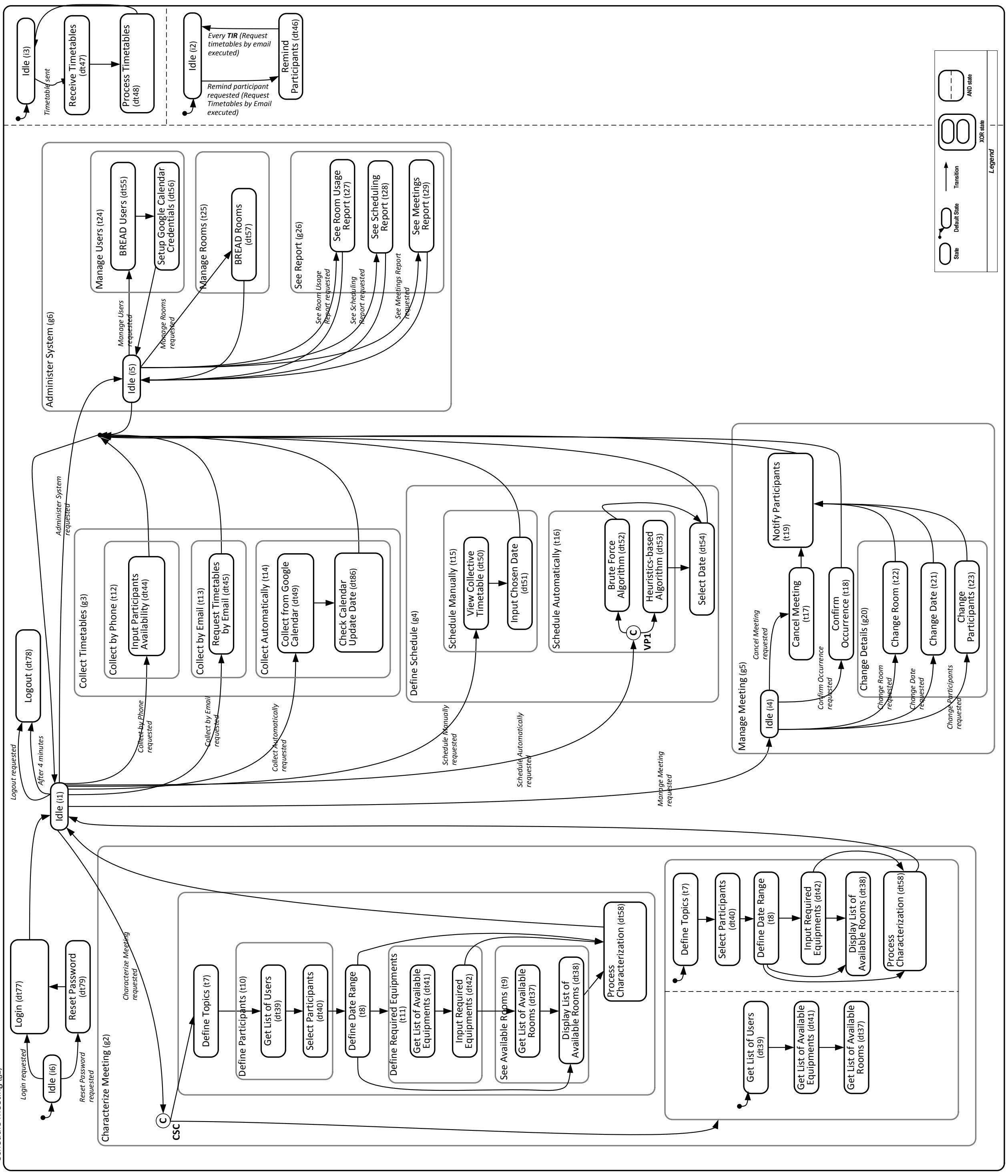

Schedule Meeting (g1)
Idle (i6)
Login requested
Login (dt77)
Reset Password requested
Reset Password (dt79)
Idle (i1)
Logout requested
After 4 minutes
Logout (dt78)
Administer System requested
Characterize Meeting requested
Characterize Meeting (g2)
CSC
Define Topics (t7)
Define Participants (t10)
Get List of Users (dt39)
Select Participants (dt40)
Define Date Range (t8)
Define Required Equipments (t11)
Get List of Available Equipments (dt41)
Input Required Equipments (dt42)
See Available Rooms (t9)
Get List of Available Rooms (dt37)
Display List of Available Rooms (dt38)
Process Characterization (dt58)
Collect Timetables (g3)
Collect by Phone (t12)
Input Participants Availability (dt44)
Collect by Phone requested
Collect by Email (t13)
Request Timetables by Email (dt45)
Collect by Email requested
Collect Automatically (t14)
Collect from Google Calendar (dt49)
Check Calendar Update Date (dt86)
Collect Automatically requested
Define Schedule (g4)
Schedule Manually (t15)
View Collective Timetable (dt50)
Input Chosen Date (dt51)
Schedule Manually requested
Schedule Automatically (t16)
Brute Force Algorithm (dt52)
Heuristics-based Algorithm (dt53)
VP1
Select Date (dt54)
Schedule Automatically requested
Manage Meeting (g5)
Idle (i4)
Manage Meeting requested
Cancel Meeting requested
Cancel Meeting (t17)
Confirm Occurrence requested
Confirm Occurrence (t18)
Notify Participants (t19)
Change Details (g20)
Change Room (t22)
Change Room requested
Change Date (t21)
Change Date requested
Change Participants (t23)
Change Participants requested
Administer System (g6)
Idle (i5)
Manage Users requested
Manage Users (t24)
BREAD Users (dt55)
Setup Google Calendar Credentials (dt56)
Manage Rooms requested
Manage Rooms (t25)
BREAD Rooms (dt57)
See Report (g26)
See Room Usage Report requested
See Room Usage Report (t27)
See Scheduling Report requested
See Scheduling Report (t28)
See Meetings Report requested
See Meetings Report (t29)
Idle (i3)
Timetable sent
Receive Timetables (dt47)
Process Timetables (dt48)
Idle (i2)
Every TIR (Request timetables by email executed)
Remind participant requested (Request Timetables by Email executed)
Remind Participants (dt46)
State
Default State
Transition
XOR state
AND state
Legend

Figure 4.12: Statechart for the Meeting Scheduler system with the specified transitions. Default transition events omitted for enhanced visualization.

4.2.8 Include adaptation elements

The adaptation components provided by frameworks such as Rainbow (GARLAN et al., 2004) and Zanshin (TALLABACI; SOUZA, 2013) are able to identify when an adaptation is required, and what is the best adaptation to perform. However, the system as a whole, often referred to as the target system, is still responsible for gathering the monitoring data that will be sent to and processed by these components, as well as for enacting the adaptation itself. For instance, considering the Meeting Scheduler system as the target system, and the *Login* task (*dt77*) and its *NeverFail* awareness requirement (*AR8*) as our focal point, we have that:

1. the Meeting Scheduler system will monitor the execution of the *Login* task;
2. the resulting data will be sent to the adaptation component, reporting whether the execution was successful or unsuccessful;
3. in case of failure, the adaptation component will identify the respective adaptation strategy: retry after a five seconds delay (Table 4.4);
4. the Meeting Scheduler system itself will wait the five seconds delay and retry to execute the *Login* task, at which point this cycle restarts.

The goal of this sub-process is to instrument the target system so that it is able to monitor the required information, send it to an adaptation component, and finally enact the required adaptations. This is achieved through the steps presented in Fig. 4.13: *Specify monitoring points* and *Apply adaptation strategy patterns*.

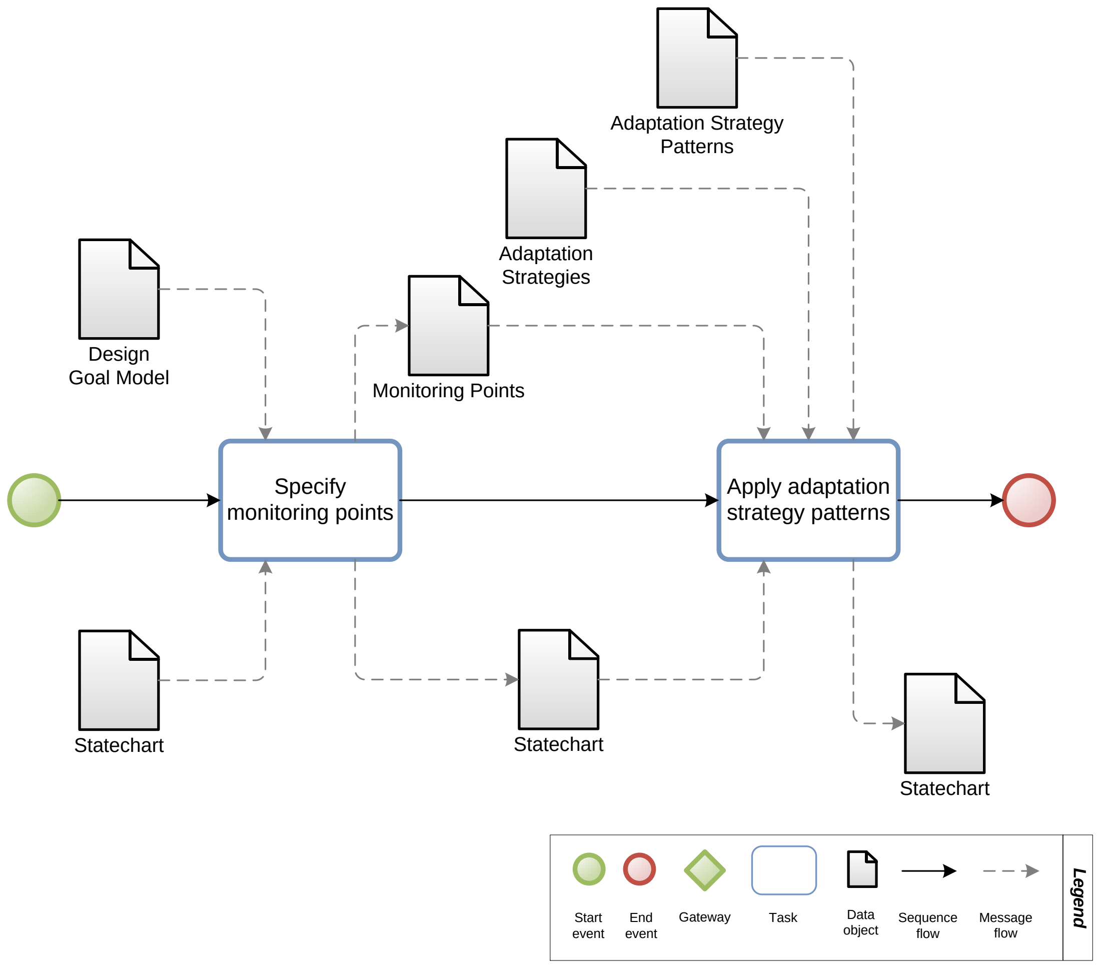
In the *Specify monitoring points* step, every awareness requirement will be mapped onto a *start action* and an *end action* — the former declares when the monitoring should start, and takes the form of *informStartARX*. The latter defines when it finishes, and is defined as *informResultARX*. Then, it is necessary to specify when these actions should be performed.

The instrumentation of the system not only determines runtime monitoring, but also how to process the responses provided by the adaptation component. For instance, how will the system react when the adaptation component informs that it needs to retry a task, or to notify an actor? These reactions are defined on the adaptation strategies specification (Section 4.2.5), and are translated onto the statechart through the *Apply adaptation strategy patterns* step (Fig. 4.13).

4.2.8.1 Example

The monitoring points for the Meeting Scheduler system are defined in Table 4.6. The specification of some points can be straightforward, such as those for *AR8*, which specifies that the *Login* task should never fail. Since this awareness requirements is local to a single task, both the start and end points are also localized. This is also the case for *AR1*, *AR4* to *AR7*, and *AR10* (related, respectively, to *Characterization done in under 5 minutes*, *Calendars Up to Date*, *Characterize Meeting*, and *Availability of Service > 90%*). The awareness requirement *AR2*

Figure 4.13: The *Include adaptation elements* sub-process



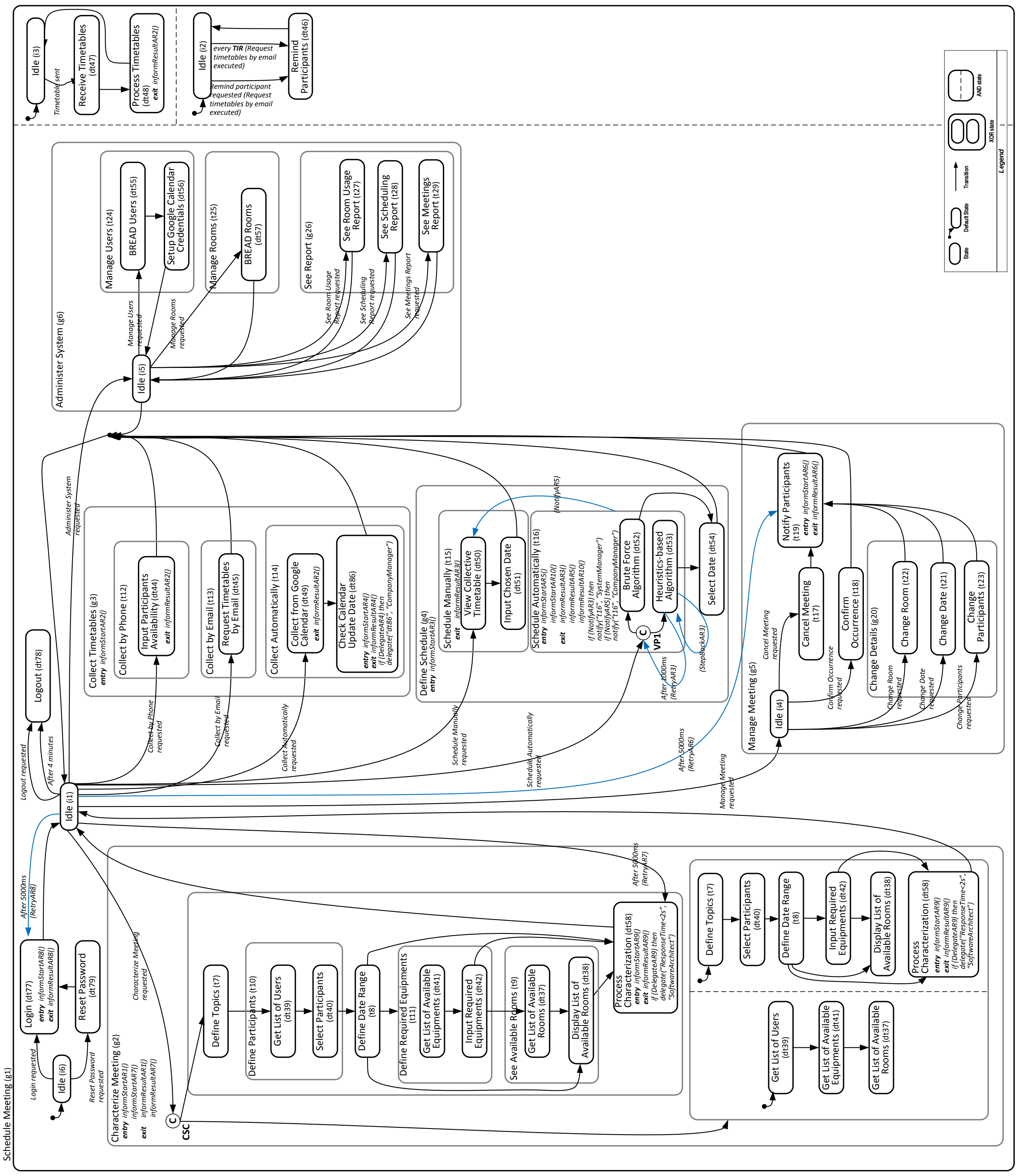
refers to the *Collect Timetables* goal. Since the children of that goal are spread into different flows, it was also necessary to spread its end actions into the different states that represent those children: *dt44*, dt48 and *dt49* (*Input Participants Availability*, *Process Timetable*, and *Collect from Google Calendar*, respectively). A similar separation was made with the end actions related to *AR3*, which is related to *Define Schedule*. The *informStartARX* and *informResultARX* actions are placed on each start and end point, respectively, as shown on Fig. 4.14.

Table 4.6: Start and end points for monitoring the awareness requirements of the Meeting Scheduler system

Awareness Requirement	**Start**	**End**
*AR*1	g2 (*Characterize Meeting*)	g2 (*Characterize Meeting*)
*AR*2	g3 (*Collect Timetables*)	dt44 (*Input Participants Availability*); dt48 (*Process Timetable*); dt49 (*Collect from Google Calendar*)
*AR*3	g4 (*Define Schedule*)	t15 (*Schedule Manually*) ; t16 (*Schedule Automatically*)
*AR*4	dt86 (*Check Calendar Update Date*)	dt86 (*Check Calendar Update Date*)
*AR*5	t16 (*Schedule Automatically*)	t16 (*Schedule Automatically*)
*AR*6	t19 (*Notify Participants*)	t19 (*Notify Participants*)
*AR*7	g2 (*Characterize Meeting*)	g2 (*Characterize Meeting*)
*AR*8	dt77 (*Login*)	dt77 (*Login*)
*AR*9	dt58 (*Process Characterization*)	dt58 (*Process Characterization*)
*AR*10	t16 (*Schedule Automatically*)	t16 (*Schedule Automatically*)

The *Login* (*dt77*) task can be used to exemplify the outcome of this sub-process. The awareness requirement *AR8*, which is attached to the *Login* task, determines that it should never fail (Table 4.2). Its monitoring points were defined having the task itself as both the start and end points (Table 4.6). Hence, the entry action *informStartAR8* and the exit action *informResultAR8* were include in the state that represents the execution of that task. The adaptation strategy for *AR8* determines that, in case of failure, the system should retry it, after a 5000 milliseconds delay (Table 4.4). This strategy is expressed in the statechart by a transition from the next state (*i1*) back to the *Login* state, with the respective event (*After 5000ms*) and condition (*RetryAR8*), where *RetryAR8* is a flag which indicates whether the retry strategy for *AR8* applies (Fig. 4.14).

The other inclusions on the statechart, besides the monitoring actions, are: new outgoing transitions from *Heuristics-based Algorithm* (*AR3*); new exit actions on *Schedule Automatically* (*AR3* and *AR5*); new exit actions on *Check Calendar Update Date* (*AR4*); new transition from *Schedule Automatically* to *Schedule Manually* (*AR5*); new transition from *i1* to *Notify Partic-*



Schedule Meeting (g1)
Idle (i6)
Login requested
Login (dt77)
Reset Password requested
Reset Password (dt79)
After 5000ms (RetryAR8)
Idle (i1)
Logout requested
After 4 minutes
Logout (dt78)
Administer System requested
Administer System (g6)
Idle (i5)
Manage Users requested
Manage Users (t24)
BREAD Users (dt55)
Setup Google Calendar Credentials (dt56)
Manage Rooms requested
Manage Rooms (t25)
BREAD Rooms (dt57)
See Report (g26)
See Room Usage Report requested
See Room Usage Report (t27)
See Scheduling Report requested
See Scheduling Report (t28)
See Meetings Report requested
See Meetings Report (t29)
Idle (i3)
Timetable sent
Receive Timetables (dt47)
Process Timetables (dt48)
Idle (i2)
every TIR (Request timetables by email executed)
Remind participant requested (Request timetables by email executed)
Remind Participants (dt46)
Collect Timetables (g3)
Collect by Phone (t12)
Input Participants Availability (dt44)
Collect by Email (t13)
Request Timetables by Email (dt45)
Collect Automatically (t14)
Collect from Google Calendar (dt49)
Check Calendar Update Date (dt86)
Define Schedule (g4)
Schedule Manually (t15)
View Collective Timetable (dt50)
Input Chosen Date (dt51)
Schedule Automatically (t16)
Brute Force Algorithm (dt52)
Heuristics-based Algorithm (dt53)
Select Date (dt54)
VP1
After 1000ms (RetryAR3)
(StepBackAR3)
(NotifyAR5)
Manage Meeting (g5)
Idle (i4)
Cancel Meeting (t17)
Confirm Occurrence (t18)
Notify Participants (t19)
Change Details (g20)
Change Room (t22)
Change Date (t21)
Change Participants (t23)
After 5000ms (RetryAR6)
Manage Meeting requested
Characterize Meeting requested
Characterize Meeting (g2)
CSC
Define Topics (t7)
Define Participants (t10)
Get List of Users (dt39)
Select Participants (dt40)
Define Date Range (t8)
Define Required Equipments (t11)
Get List of Available Equipments (dt41)
Input Required Equipments (dt42)
See Available Rooms (t9)
Get List of Available Rooms (dt37)
Display List of Available Rooms (dt38)
Process Characterization (dt58)
After 5000ms (RetryAR7)
Legend
State
Default State
Transition
XOR state
AND state

Figure 4.14: Complete statechart for the Meeting Scheduler system, including transitions and adaptation elements

ipants (*AR6*); new transition from *i1* to *Process Characterization* (*AR7*); new exit actions on *Process Characterization* (*AR9*);

With the monitoring points, additional transitions and additional actions included in this sub-process, the target system becomes able to reify the adaptation strategies previously defined.

4.3 Summary

The architectural design process described in this chapter allows to incrementally create a design goal model from a requirements goal model. Throughout the process, the model is enriched with **(i)** design tasks, **(ii)** design constraints and **(iii)** design assumptions (*Identify design tasks, constraints and assumptions* step); **(iv)** assignments (*Assign tasks* step); **(v)** flow expressions (*Define basic flows* step); **(vi)** additional awareness requirements; **(vii)** additional parameters; and **(viii)** the relations between parameters and indicators (*Identify indicators, parameters and relations* step). Moreover, the **(ix)** adaptation strategies for the system are defined (*Specify adaptation strategies* step); a **(x)** statechart model is derived (*Generate base statechart* step); its **(xi)** transitions are refined (*Specify transitions* step); and the system is instrumented through **(xii)** monitoring points and **(xiii)** the application of adaptation strategy patterns (*Include adaptation elements* step).

With this process it is possible to design an adaptive system handling both requirements- and architecture-related adaptation concerns. The Meeting Scheduler system was used to illustrate the outcome of each step of the process. For the sake of understandability, the process and the meeting scheduler example were presented sequentially, similarly to a waterfall-like process. An illustration of the process being enacted incrementally and iteratively is presented in Appendix A.

In order to facilitate the use of the proposed process, we developed a tool that supports the creation of design goal models, as well as the derivation of statecharts. This tool is described in Chapter 5.

5 GATO - Goal to Architecture tool

In the previous chapters we presented the steps required to refine goal models towards the generation of statecharts, as well as to further refine the statechart. Here we present the Goal to Architecture tool (GATO), developed in order to support the architectural design process by means of modeling capabilities. The first section of this chapter provides an overview of the tool, presenting its requirements and user interface. The second section describes the mechanisms we developed in order to automatically generate base statecharts. The third section discusses the tool integration with the Zanshin framework.

5.1 Overview

During initial experiments with our framework, we identified the need for providing tool support, mainly because otherwise the translation of flow expressions onto statecharts would be too cumbersome. Thus, we developed a prototype tool which supports the modeling required by the process, as well as the derivation itself. The modeling is performed on a web-based client using the JointJS library[1], which provides facilities for displaying diagrams on websites.

The requirements for our supporting tool are presented in Fig. 5.1. It supports the edition of requirements elements (goals, tasks, domain assumptions, and quality constraints), as well as of the edition of design elements (design tasks, design constraints, design assumption, assignments, and behavioral refinements) and the edition of adaptation elements (awareness requirements and parameters). After included in the model, each element may be moved, renamed, or deleted.

Besides editing and managing goal models, it is also possible to create adaptation specifications (adaptation strategies and parameters) and to specify transitions (events and conditions). The tool also supports the derivation of statecharts from the design goal model. Lastly, the functionality of *Export to Zanshin* was included in order to facilitate the integration with Zanshin's component.

The GATO tool has three main quality constraints: *Portability* of model editing, *Re-*

[1]Available at `http://www.jointjs.com`.

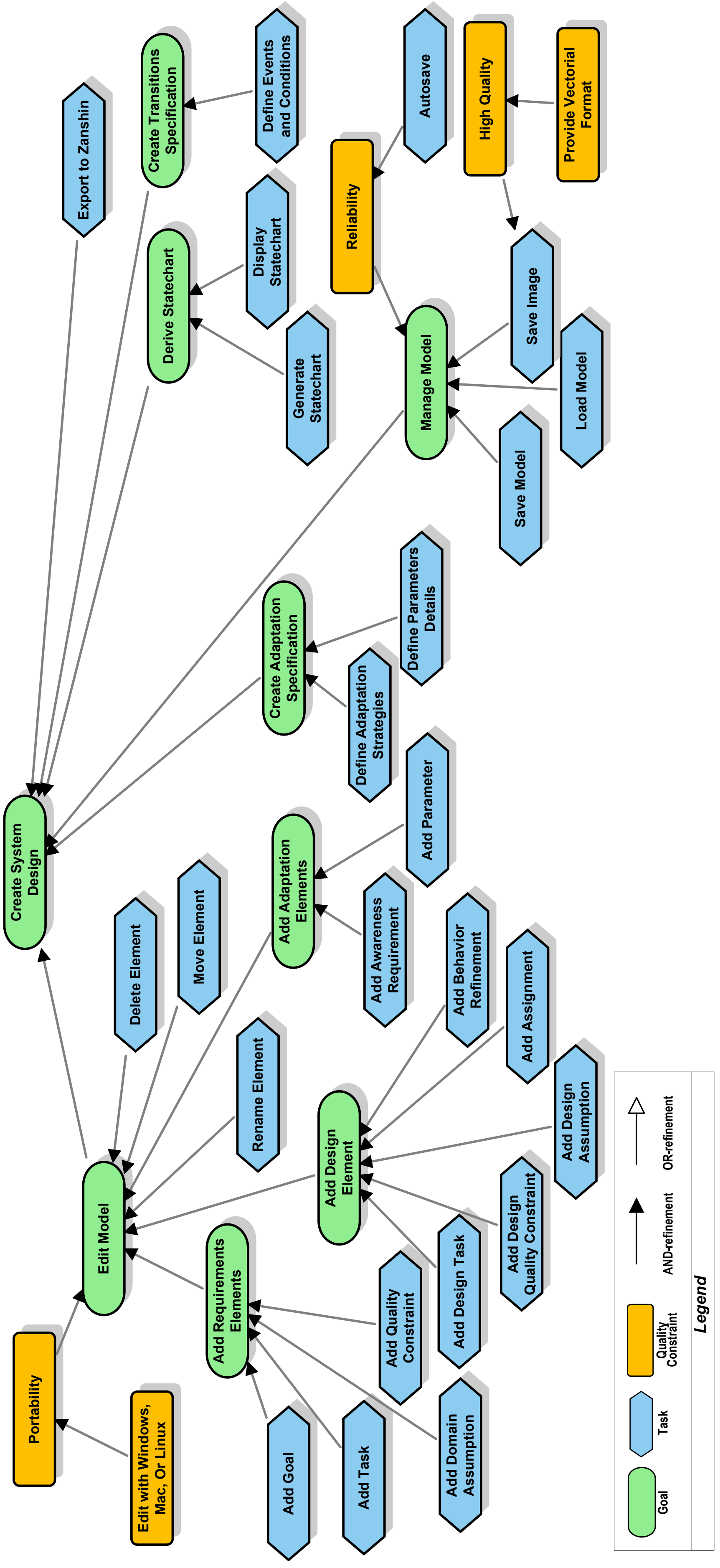


Figure 5.1: Requirements of the *Goal to Arch* tool

liability of model managing, and *High Quality* of saved images. By portability we mean the ability to run over different operating systems: Windows, Mac, and Linux. Reliability, which is too abstract, was made concrete with an *Autosave* task. I.e., if the tool presents an autosave functionality, the system is considered to be sufficiently reliable. Lastly, in order to satisfy the *High Quality* constraint, it was decided to *Provide Vectorial Format*[2], such as SVG[3] (Scalable Vector Graphics).

A screenshot of the tool is presented in Fig. 5.2. The central area (Fig. 5.2-B) contains the goal model itself, in this case showing the requirements of the Meeting Scheduler system. By pointing the mouse over the different elements of the model, users may add goals, tasks, quality constraints, domain assumptions , awareness requirements, and parameters. It is also possible to move, delete, and rename each element.

The panel on the bottom section of the tool is the toolbar (Fig. 5.2-C), which contains some visualization options: hide/show the ids of the elements, and hide/show adaptation elements (awareness requirements and parameters). It also provides some basic file management functionalities: save/load model, resize the drawing area, create a new model (clear), and export to image (Save SVG). Other two functionalities accessible through buttons in this panel are presented later in this chapter: *analyze model* and *metamodel*.

The top-left panel (Fig. 5.2-A) shows buttons to navigate between the four tabs of the tool, which are based on the proposed process: *1) Views*, *2) Adaptation specification*, *3) Derivation*, and *4) Transitions*. The first tab, which is the one shown on Fig. 5.2, is related to the design goal models. There, it is possible to edit the model and to navigate through its different views. This tab supports the first three steps of the architectural design process presented in the previous chapter: *Identify design tasks, constraints and assumptions*; *Assign tasks*; and *Define basic flows*. Furthermore, it partially supports the fourth step: *Identify indicators, parameters and relations*.

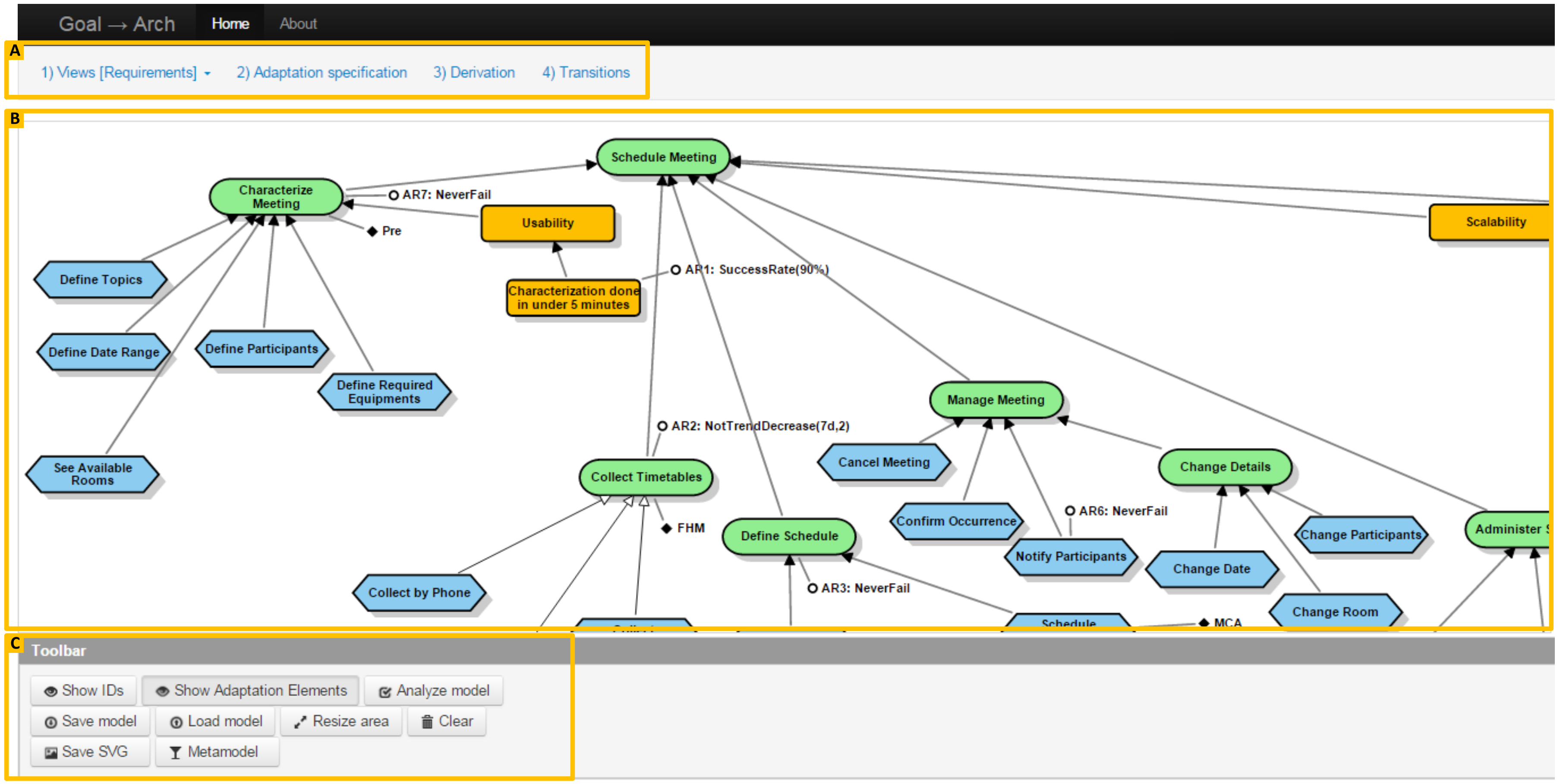
As described on Section 3.3, the Design Goal Model has four different views: the *Requirements view*, the *Design view*, the *Delegation view* , and the *Behavior refinement view*. All those views can be accessed through this first tab, where it is possible to switch back and forth between the views. For instance, when switching from Assignment to Requirements, all the design elements and assignments are hidden, so that the model can be visualized and edited as a regular requirements model.

The *Requirements view* allows to visualize and edit a requirements goal model, with goals, tasks, quality constraints, domain assumptions, awareness requirements and parameters. Fig. 5.2 shows the requirements view of the Meeting Scheduler system on the GATO tool.

[2]Vectorial image formats usually provide high quality images because, unlike bitmaps, they do not present quality loss under resizing.

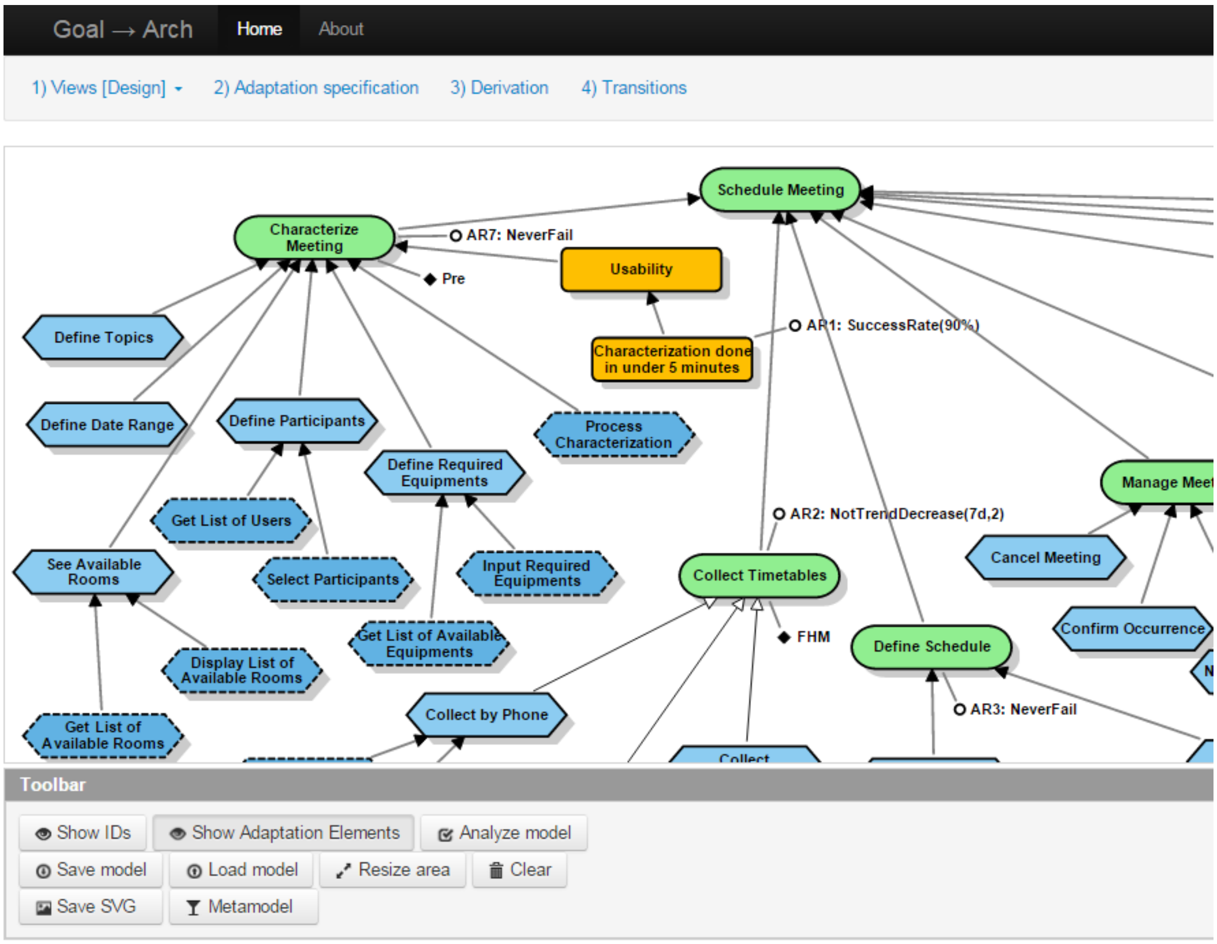
[3]SVG is a largely adopted image file format, specified through a World Wide Web Consortium standard.

Figure 5.2: Requirements view in the GATO tool. A) tab selection; B) main area; C) toolbar



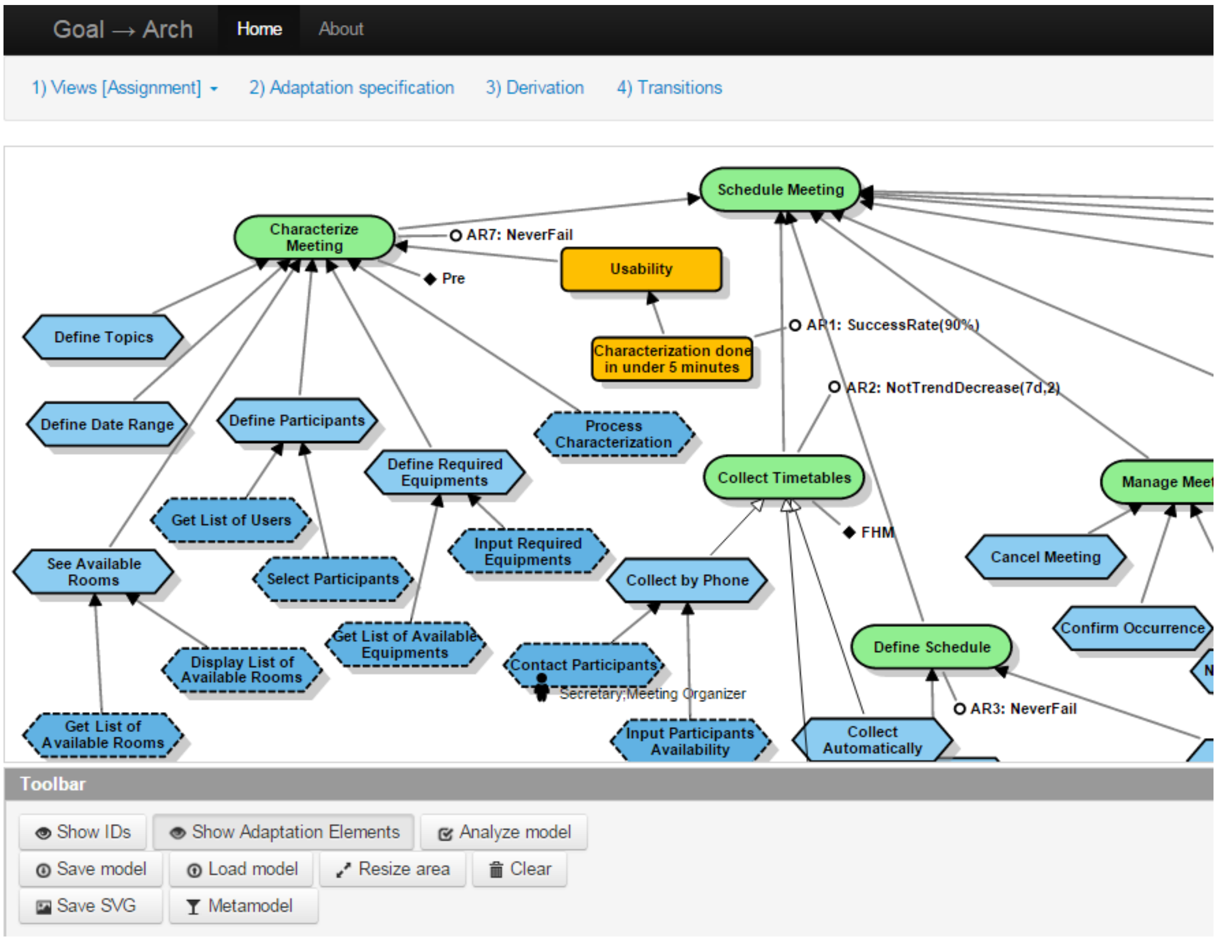
In the *Design view* it is possible to not only visualize requirements and adaptation elements, but also to visualize and edit design elements: design tasks, design constraints, and design assumptions. Fig. 5.3 shows the design view of the Meeting Scheduler system on the GATO tool, with its design elements: *Get List of Available Rooms*, *Display List of Available Rooms*, *Get List of Users*, *Select Participants*, *Get List of Available Equipments*, *Input Required Equipments*, and *Process Characterization*.

Figure 5.3: Design view in the supporting tool. Compared with the requirements view, this view also presents design tasks, design constraints, and design assumptions



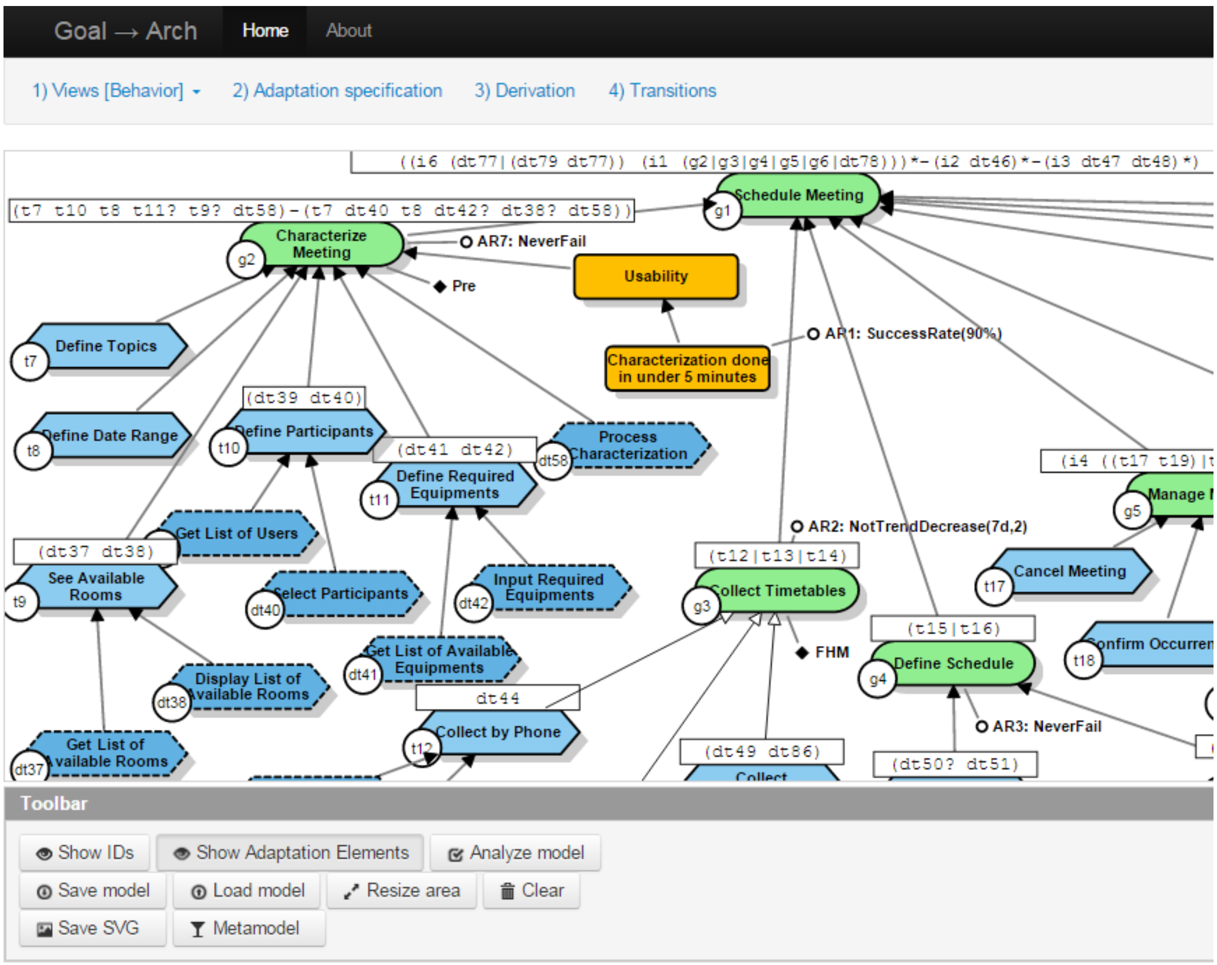
The *Assignments view* shows the same elements from the design view, but also allows to create and visualize assignments on the goal model. Fig. 5.4 shows the assignments view for the Meeting Scheduler system. Assignments are represented by the icon of a person. In this example, the only task that is assigned is the *Contact Participants* design task, which is assigned to a Secretary or to a Meeting Organizer.

Figure 5.4: Assignments view in the supporting tool. Compared with the design view, this view also presents task assignments, as in the *Contact Participants* design task



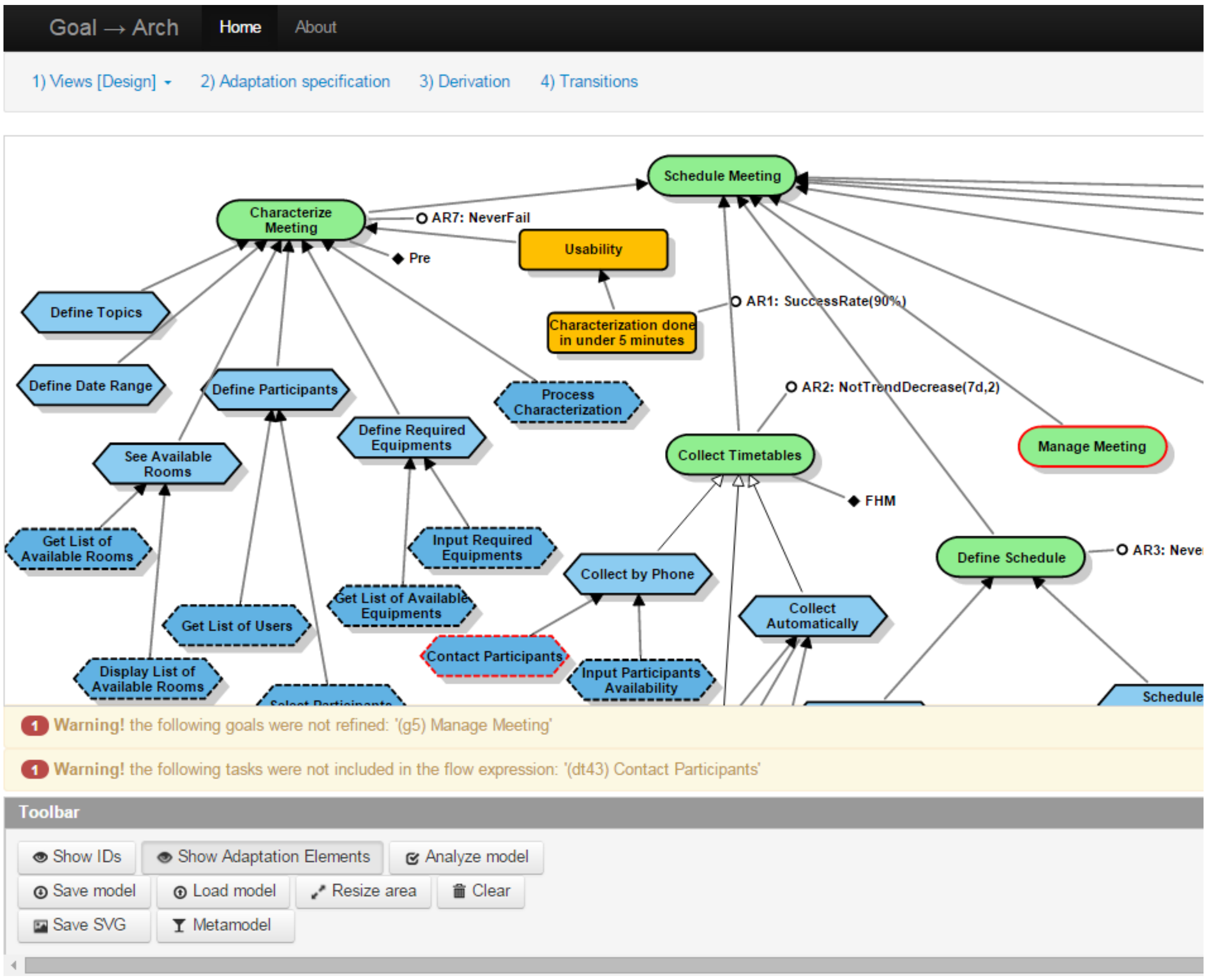
The *Behavior refinement view* allows to see and edit flow expressions. Since the ids of the model elements are referenced on the flow expression, the tool also displays the ids, on the bottom-left region of each model element. Fig. 5.5 shows the behavior view of the Meeting Scheduler system, with its flow expressions. For example, take the *Define Required Equipments* task, as its flow expression references *dt41* and *dt42*. By observing the ids displayed in this view, it is possible to know that *dt41* is the id of the *Get List of Available Equipments* design task, whereas *dt42* represents the *Input Required Equipments* design task.

Figure 5.5: Behavior Refinement view in the supporting tool. Compared with the assignments view, this view also presents the ids of each element and their behavior refinements (flow expressions)



Besides editing the design goal model, the GATO tool is also able to analyze the model. This automated analysis identifies two potential problems: goals that are not refined onto tasks, and elements that are still missing in the flow expressions. The results of this analysis are displayed in a warnings panel at the bottom of the screen, while the problematic elements are highlighted in the model itself. Fig. 5.6 shows the results of an automatic analysis on the Meeting Scheduler system. On this example, the warnings panel shows that the *Manage Meeting* goal is unrefined, whereas the *Contact Participants* task is not included in the flow expressions of the system.

Figure 5.6: Screenshot of the supporting tool showing the warnings panel below the design goal model

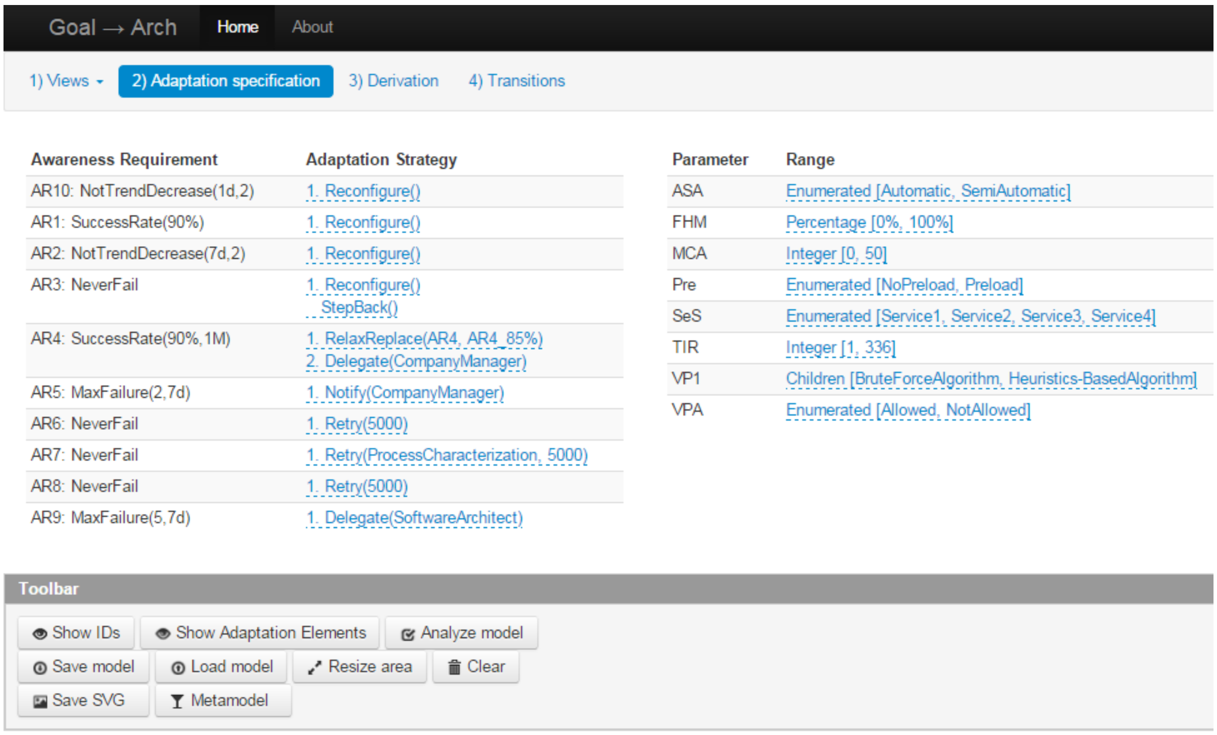

Warning! the following goals were not refined: '(g5) Manage Meeting'

Warning! the following tasks were not included in the flow expression: '(dt43) Contact Participants'

As described earlier on this section, the GATO tool has four tabs: *Views*, *Adaptation specification*, *Derivation*, and *Transitions* (Fig. 5.2-A). The previous screenshots show the first tab of the tool, with the different views of the design goal model. The second tab, *Adaptation specification*, is where the user can define the adaptation strategies of the system, as well as the range of its parameters, supporting the *Specify adaptation strategies* step of the architectural design process.

Fig. 5.7 shows the adaptation specification of the Meeting Scheduler system. The left-hand region of this tab shows the adaptation strategies for each awareness requirement of the system. The right-hand region of the tab presents the range of each parameter of the system. While awareness requirements and parameters are retrieved from the design goal model, the adaptation strategies and ranges are input by the user.

Figure 5.7: Screenshot of the supporting tool showing the adaptation specification



Awareness Requirement	Adaptation Strategy
AR10: NotTrendDecrease(1d,2)	1. Reconfigure()
AR1: SuccessRate(90%)	1. Reconfigure()
AR2: NotTrendDecrease(7d,2)	1. Reconfigure()
AR3: NeverFail	1. Reconfigure() StepBack()
AR4: SuccessRate(90%,1M)	1. RelaxReplace(AR4, AR4_85%) 2. Delegate(CompanyManager)
AR5: MaxFailure(2,7d)	1. Notify(CompanyManager)
AR6: NeverFail	1. Retry(5000)
AR7: NeverFail	1. Retry(ProcessCharacterization, 5000)
AR8: NeverFail	1. Retry(5000)
AR9: MaxFailure(5,7d)	1. Delegate(SoftwareArchitect)

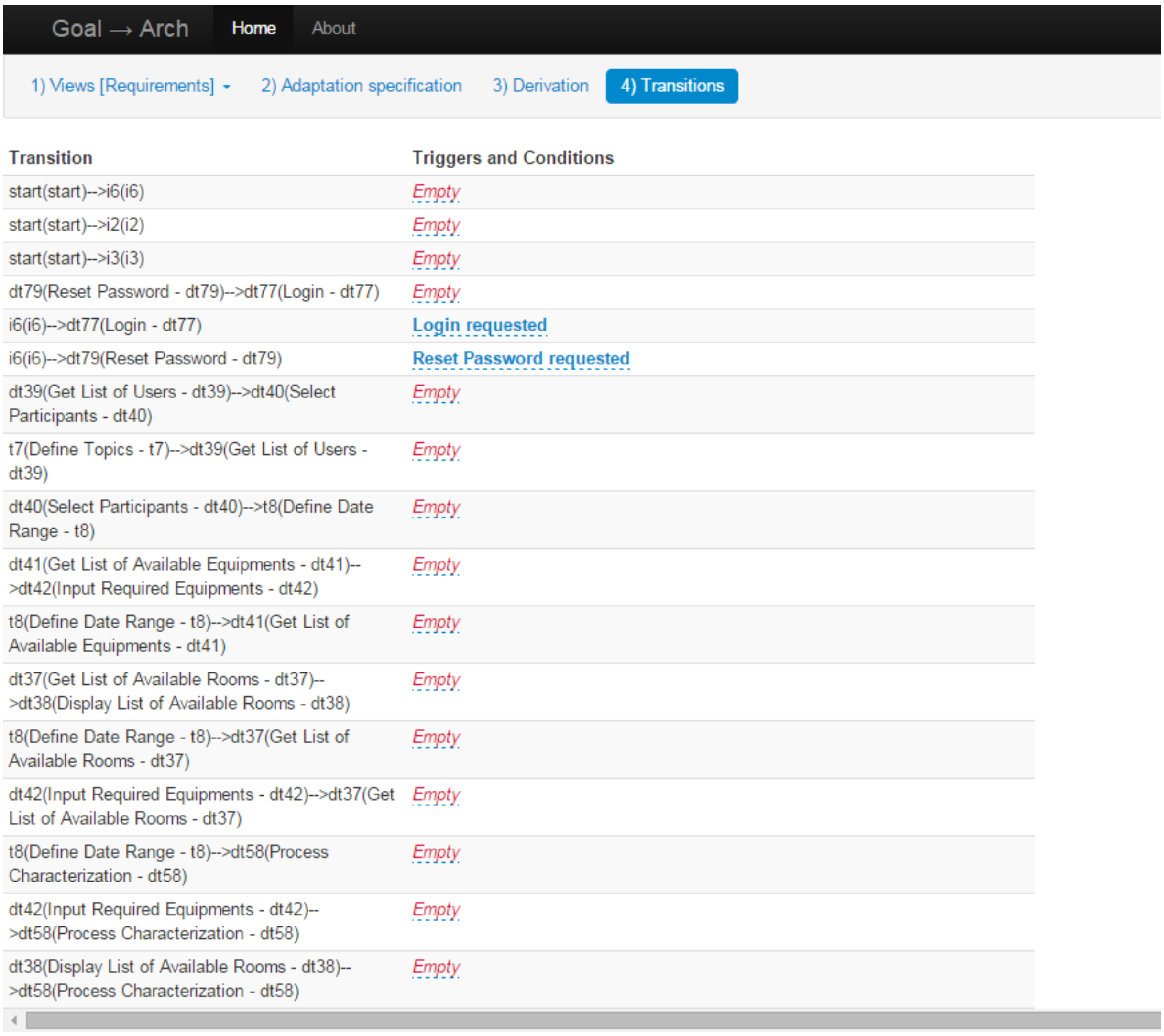
Parameter	Range
ASA	Enumerated [Automatic, SemiAutomatic]
FHM	Percentage [0%, 100%]
MCA	Integer [0, 50]
Pre	Enumerated [NoPreload, Preload]
SeS	Enumerated [Service1, Service2, Service3, Service4]
TIR	Integer [1, 336]
VP1	Children [BruteForceAlgorithm, Heuristics-BasedAlgorithm]
VPA	Enumerated [Allowed, NotAllowed]

The third tab of the GATO tool, *Derivation*, which is concerned with the automatic generation of a base statechart, is discussed in Section 5.2. The fourth tab, *Transitions*, is concerned with the seventh step of the architectural design process: *Specify transitions* (Section 4.2.7). Whereas the transitions of the base statechart are identified during derivation, on this tab the user specifies the events and conditions for each transition.

Fig. 5.8 shows the transitions specification for the Meeting Scheduler system on the GATO tool. On this example, only two transitions are specified: the transition from *i6* to *dt77* (*Login*), and the transition from *i6* to *dt79* (*Reset Password*). These transitions are triggered by the *Login requested* and *Reset Password requested* events, respectively.

The next section presents in details the derivation of statecharts, describing not only its user interface but also the statechart derivation algorithm we have developed.

Figure 5.8: Screenshot of the supporting tool showing the transitions specification



Transition	Triggers and Conditions
start(start)-->i6(i6)	Empty
start(start)-->i2(i2)	Empty
start(start)-->i3(i3)	Empty
dt79(Reset Password - dt79)-->dt77(Login - dt77)	Empty
i6(i6)-->dt77(Login - dt77)	Login requested
i6(i6)-->dt79(Reset Password - dt79)	Reset Password requested
dt39(Get List of Users - dt39)-->dt40(Select Participants - dt40)	Empty
t7(Define Topics - t7)-->dt39(Get List of Users - dt39)	Empty
dt40(Select Participants - dt40)-->t8(Define Date Range - t8)	Empty
dt41(Get List of Available Equipments - dt41)-->dt42(Input Required Equipments - dt42)	Empty
t8(Define Date Range - t8)-->dt41(Get List of Available Equipments - dt41)	Empty
dt37(Get List of Available Rooms - dt37)-->dt38(Display List of Available Rooms - dt38)	Empty
t8(Define Date Range - t8)-->dt37(Get List of Available Rooms - dt37)	Empty
dt42(Input Required Equipments - dt42)-->dt37(Get List of Available Rooms - dt37)	Empty
t8(Define Date Range - t8)-->dt58(Process Characterization - dt58)	Empty
dt42(Input Required Equipments - dt42)-->dt58(Process Characterization - dt58)	Empty
dt38(Display List of Available Rooms - dt38)-->dt58(Process Characterization - dt58)	Empty

5.2 Statechart Derivation

The module for deriving statecharts from flow expressions was built using the SableCC tool[4], which automatically generates Java code for parsing an input text, as well as for creating and traversing an abstract syntax tree of the parsed text. The code is generated by SableCC taking as input a custom-defined grammar, which specifies the tokens and production rules of the language that will be parsed. An excerpt of the grammar for our flavor of flow expressions is presented in Listing 5.1, showing the tokens used in the expressions (lines 2-12) and how they can be used to define the terms of the expressions (lines 16-28).

Listing 5.1: Grammar for flow expressions

```
 1  Tokens
 2     lparen  = '(';
 3     rparen  = ')';
 4     or      = '|';
 5     shuffle = '-';
 6
 7     plus   = '+';
 8     times  = '*';
 9     questionmark  = '?';
10
11     whitespace = (space | newline)+;
12     id = letter (letter | digit)*;
13
14
15  Productions
16     exp    = {term} term
17            | {sequence} exp whitespace term
18            ;
19     term   = {basicterm} basicterm
20            | {alternative} term or basicterm
21            | {orthogonal} term shuffle basicterm
22            | {zeroormore} term times
23            | {oneormore} term plus
24            | {optional} term questionmark
25            ;
26     basicterm = {state} id
27            | {nested} lparen exp rparen
28            ;
```

The code for actually identifying the states, transitions and concurrent states that should be derived from a given flow expression was developed on top of the depth-first tree traverser generated with SableCC. This mapper was wrapped up as a restful service using Jersey[5], allowing it to be invoked from our web-based client. The result of the derivation is then displayed back in the web client.

[4] Available at http://sablecc.org.

[5] Java library, available at http://jersey.java.net.

The code generated by SableCC takes as input a string of characters (the flow expression) and generates its lexical tree, according to the provided grammar. It also provides an implementation of the Visitor design pattern in order to traverse the tree. According to the operator in each segment of the tree, it calls the appropriate custom-defined methods.

In order to generate the statechart, the first thing we do is to identify the initial states. The pseudocode for this algorithm is presented in Algorithm 1. The general idea is to find the elements to the left hand of the expression, ignoring what is in the right hand, in the case of a Sequence (lines 2-4). If the expression is an Alternative, both sides of the expression need to be evaluated (lines 5-8). In the case of a Optional expression, we need to consider both the term itself and what is in its right hand (lines 9-14). These functions are called recursively until a Id is reached, in which case the Id is included in the set of initial states (lines 15-17).

Algorithm 1 EvalFirstStates

```
 1: initialStates ← $\phi$

 2: function caseASequenceExp($e$)
 3:     e.leftHand.apply()
 4: end function

 5: function caseAAlternativeTerm($e$)
 6:     e.leftHand.apply()
 7:     e.rightHand.apply()
 8: end function

 9: function caseAOptionalTerm($e$)
10:     e.getTerm.apply()
11:     if e.isPartOfASequence() then
12:         e.parent.rightHand.apply()
13:     end if
14: end function

15: function caseTId($e$)
16:     initialStates.add(e)
17: end function
```

After identifying the initial states, we can identify the remaining states and transitions, as described in Algorithm 2. When a Sequence expression is found, we identify the final states of the expression in the left side, and also the initial states of the expression in the right side (lines 4-5). The code for finding the initial states was already presented in Algorithm 1. The identification of the final states is similar to the identification of initial states, but traversing the tree to the right (Algorithm 3). Then transitions are created from the final states of the left side to the initial states of the right side (lines 6-10). For the case of One Or More terms, we identify the initial and final states of the term itself, and create the appropriate transitions (lines 12-20). The recursion ends when an Id is reached (lines 21-23).

Algorithm 2 EvalMapping

```
1:  states ← $\phi$
2:  transitions ← $\phi$

3:  function CASEASEQUENCEEXP($e$)
4:      previousStates gets e.leftHand.evalLastStates.apply()
5:      nextStates gets e.rightHand.evalFirstStates.apply()
6:      for all ps $\in$ previousStates do
7:          for all ns $\in$ nextStates do
8:              transitions.add(ps, ns)
9:          end for
10:     end for
11: end function

12: function CASEONEORMORETERM($e$)
13:     lastStates gets e.leftHand.evalLastStates.apply()
14:     firstStates gets e.rightHand.evalFirstStates.apply()
15:     for all ls $\in$ lastStates do
16:         for all fs $\in$ firstStates do
17:             transitions.add(ls, fs)
18:         end for
19:     end for
20: end function

21: function CASETID($e$)
22:     states.add(e)
23: end function
```

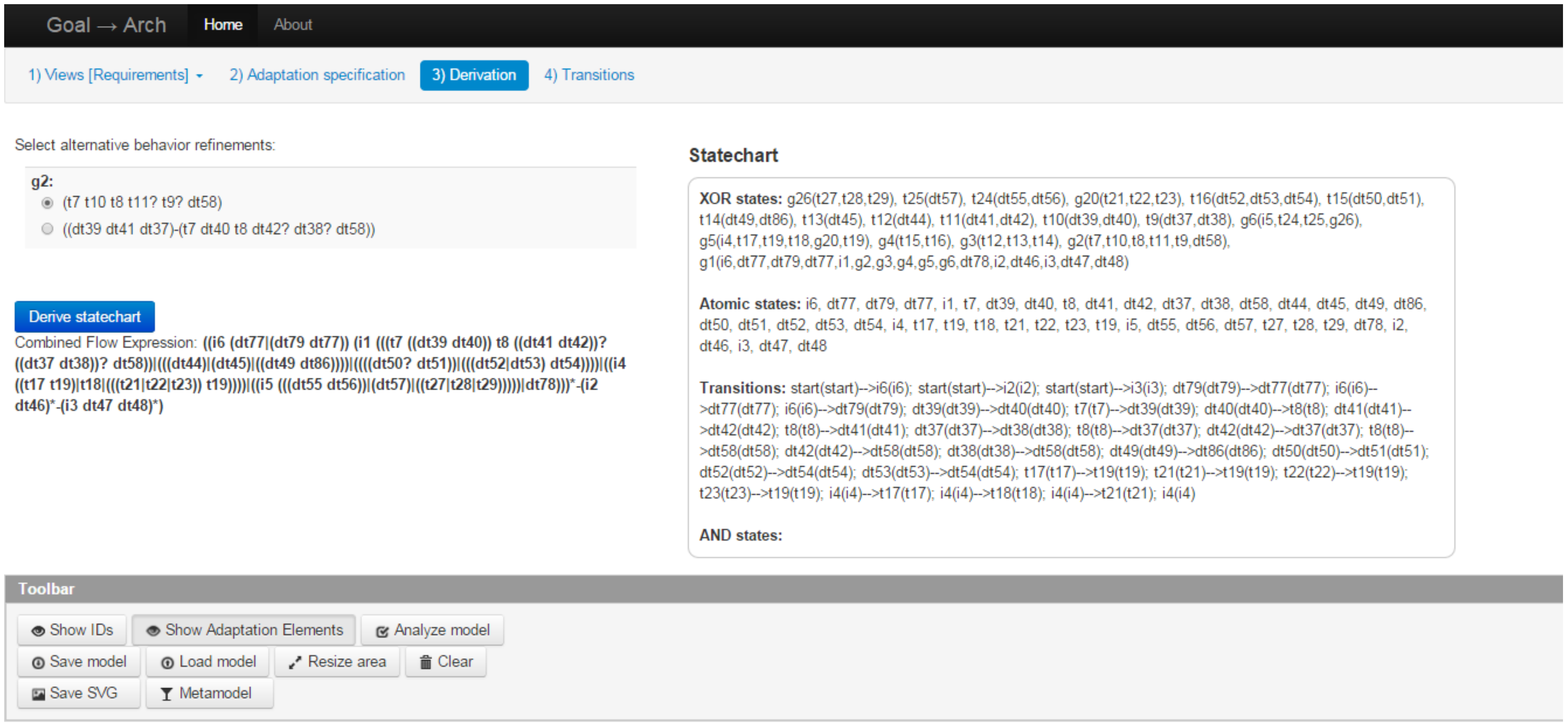
Algorithm 3 EvalLastStates

```
 1: lastStates ← $\phi$

 2: function CASEASEQUENCEEXP($e$)
 3:     e.rightHand.apply()
 4: end function

 5: function CASEAALTERNATIVETERM($e$)
 6:     e.leftHand.apply()
 7:     e.rightHand.apply()
 8: end function

 9: function CASEAOPTIONALTERM($e$)
10:     e.getTerm.apply()
11:     if e.isPartOfASequence() then
12:         e.parent.lefttHand.apply()
13:     end if
14: end function

15: function CASETID($e$)
16:     lastStates.add(e)
17: end function
```

The use of a grammar-based approach for performing the goal model - statechart transformation was selected in detriment of model transformation languages such as ATL and QVT, since the most relevant information for this transformation is the flow expression, which is textual. In order to verify the implementation of this algorithm we developed automatic tests comparing the results for given expressions to their expected results.

The result of the derivation is displayed back in two ways: as a list of states, their hierarchy, and transitions, depicted in Fig. 5.9; as a visual diagram, shown on Fig. 5.10. The creation of visual diagrams, from this output, is still in early development. Currently, the tool generates a statechart only with the atomic states; in future developments we expect to be able to include the super-states on the diagram as well.

Figure 5.9: Screenshot of output of the statechart derivation



Select alternative behavior refinements:

g2:
- [✓ (t7 t10 t8 t11? t9? dt58)]
- ((dt39 dt41 dt37)-(t7 dt40 t8 dt42? dt38? dt58))

Combined Flow Expression: **((i6 (dt77|(dt79 dt77)) (i1 (((t7 ((dt39 dt40)) t8 ((dt41 dt42))? ((dt37 dt38))? dt58))|(((dt44)|(dt45)|((dt49 dt86))))|((((dt50? dt51))|(((dt52|dt53) dt54))))|((i4 ((t17 t19)|t18|(((t21|t22|t23)) t19))))|((i5 (((dt55 dt56))|(dt57)|((t27|t28|t29)))))|dt78))*-(i2 dt46)*-(i3 dt47 dt48)*)**

Statechart

XOR states: g26(t27,t28,t29), t25(dt57), t24(dt55,dt56), g20(t21,t22,t23), t16(dt52,dt53,dt54), t15(dt50,dt51), t14(dt49,dt86), t13(dt45), t12(dt44), t11(dt41,dt42), t10(dt39,dt40), t9(dt37,dt38), g6(i5,t24,t25,g26), g5(i4,t17,t19,t18,g20,t19), g4(t15,t16), g3(t12,t13,t14), g2(t7,t10,t8,t11,t9,dt58), g1(i6,dt77,dt79,dt77,i1,g2,g3,g4,g5,g6,dt78,i2,dt46,i3,dt47,dt48)

Atomic states: i6, dt77, dt79, dt77, i1, t7, dt39, dt40, t8, dt41, dt42, dt37, dt38, dt58, dt44, dt45, dt49, dt86, dt50, dt51, dt52, dt53, dt54, i4, t17, t19, t18, t21, t22, t23, t19, i5, dt55, dt56, dt57, t27, t28, t29, dt78, i2, dt46, i3, dt47, dt48

Transitions: start(start)-->i6(i6); start(start)-->i2(i2); start(start)-->i3(i3); dt79(dt79)-->dt77(dt77); i6(i6)-->dt77(dt77); i6(i6)-->dt79(dt79); dt39(dt39)-->dt40(dt40); t7(t7)-->dt39(dt39); dt40(dt40)-->t8(t8); dt41(dt41)-->dt42(dt42); t8(t8)-->dt41(dt41); dt37(dt37)-->dt38(dt38); t8(t8)-->dt37(dt37); dt42(dt42)-->dt37(dt37); t8(t8)-->dt58(dt58); dt42(dt42)-->dt58(dt58); dt38(dt38)-->dt58(dt58); dt49(dt49)-->dt86(dt86); dt50(dt50)-->dt51(dt51); dt52(dt52)-->dt54(dt54); dt53(dt53)-->dt54(dt54); t17(t17)-->t19(t19); t21(t21)-->t19(t19); t22(t22)-->t19(t19); t23(t23)-->t19(t19); i4(i4)-->t17(t17); i4(i4)-->t18(t18); i4(i4)-->t21(t21); i4(i4)

AND states:

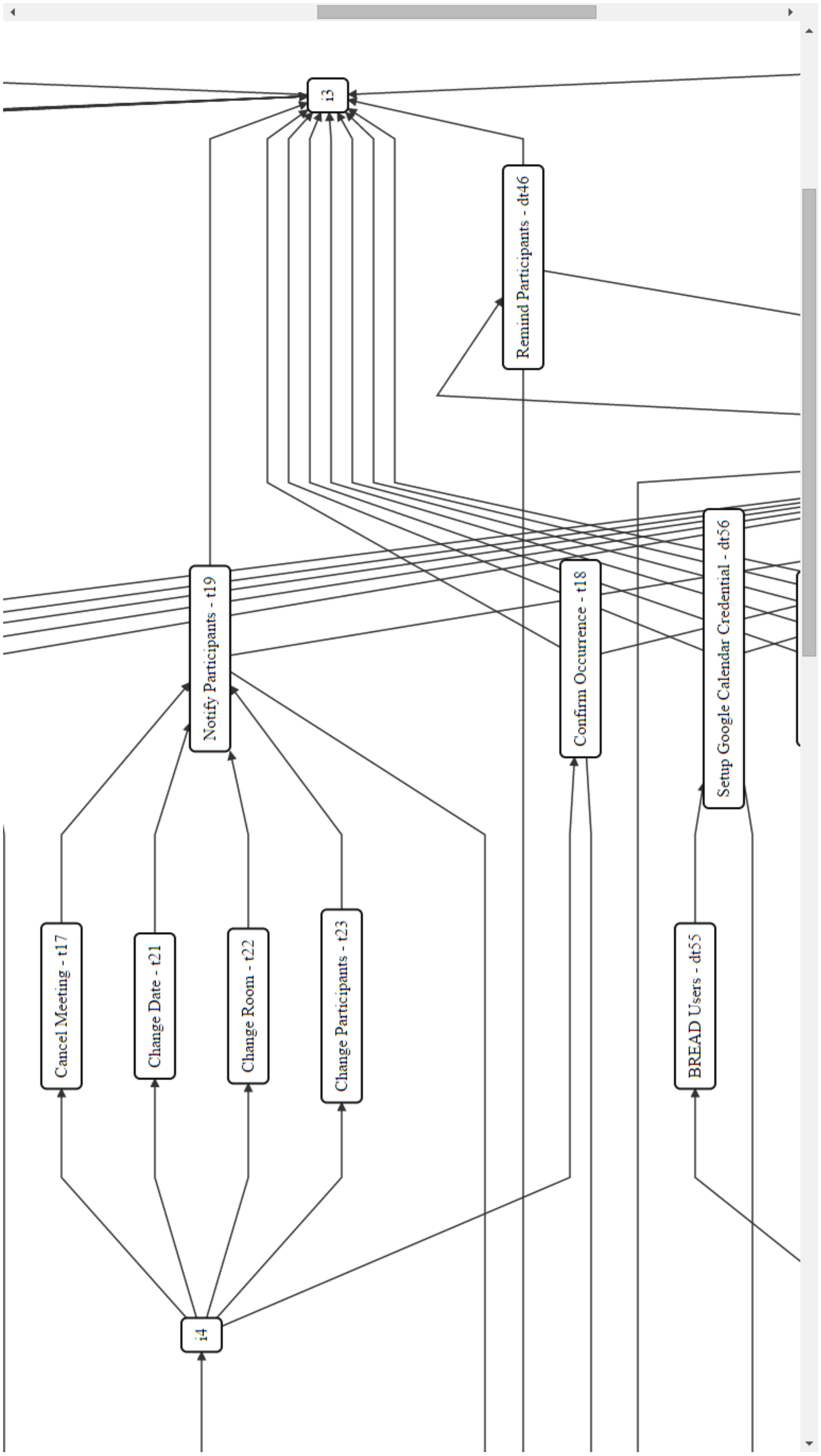

i3
Remind Participants - dt46
Notify Participants - t19
Confirm Occurrence - t18
Setup Google Calendar Credential - dt56
Cancel Meeting - t17
Change Date - t21
Change Room - t22
Change Participants - t23
BREAD Users - dt55
i4

Figure 5.10: Screenshot of the resulting statechart diagram

5.3 Integration with Zanshin

In this section we present the integration of the GATO tool with the Zanshin framework. This integration consists of exporting Zanshin models based on design goal models. By exporting such models, we expect to significantly reduce the effort required to enact the adaptation cycle using the generic adaptation component which is a part of the Zanshin framework. The use of that component is described on an Automatic Teller Machine (ATM) case study (Section 6.1).

At the time of the ATM case study, the integration of the GATO tool with the Zanshin framework was not yet performed. Throughout that study, we identified opportunities for automating the creation of Zanshin models, by mapping the constructs from MULAS design goal models onto constructs of Zanshin models.

Fig. 5.11 shows the mapping between the design goal model (DGM) metamodel and Zanshin's metamodel. DGM goals are mapped onto Zanshin goals. DGM tasks and design tasks are mapped onto Zanshin tasks. DGM quality constraints and design constraints are mapped onto Zanshin quality constraints. DGM domain assumptions and design assumptions are mapped onto Zanshin domain assumptions. DGM awareness requirements and parameters are mapped onto Zanshin awareness requirements and parameters, respectively. Lastly, DGM links are mapped onto the self-association of the *Requirement* class in Zanshin's metamodel.

With this mapping it is possible to export Zanshin models directly from the GATO tool, which allows to delegate part of the adaptation cycle to the component provided by the Zanshin framework. This functionality is accessible through the *Metamodel* button on the bottom-left region of the GATO tool (Fig. 5.2).

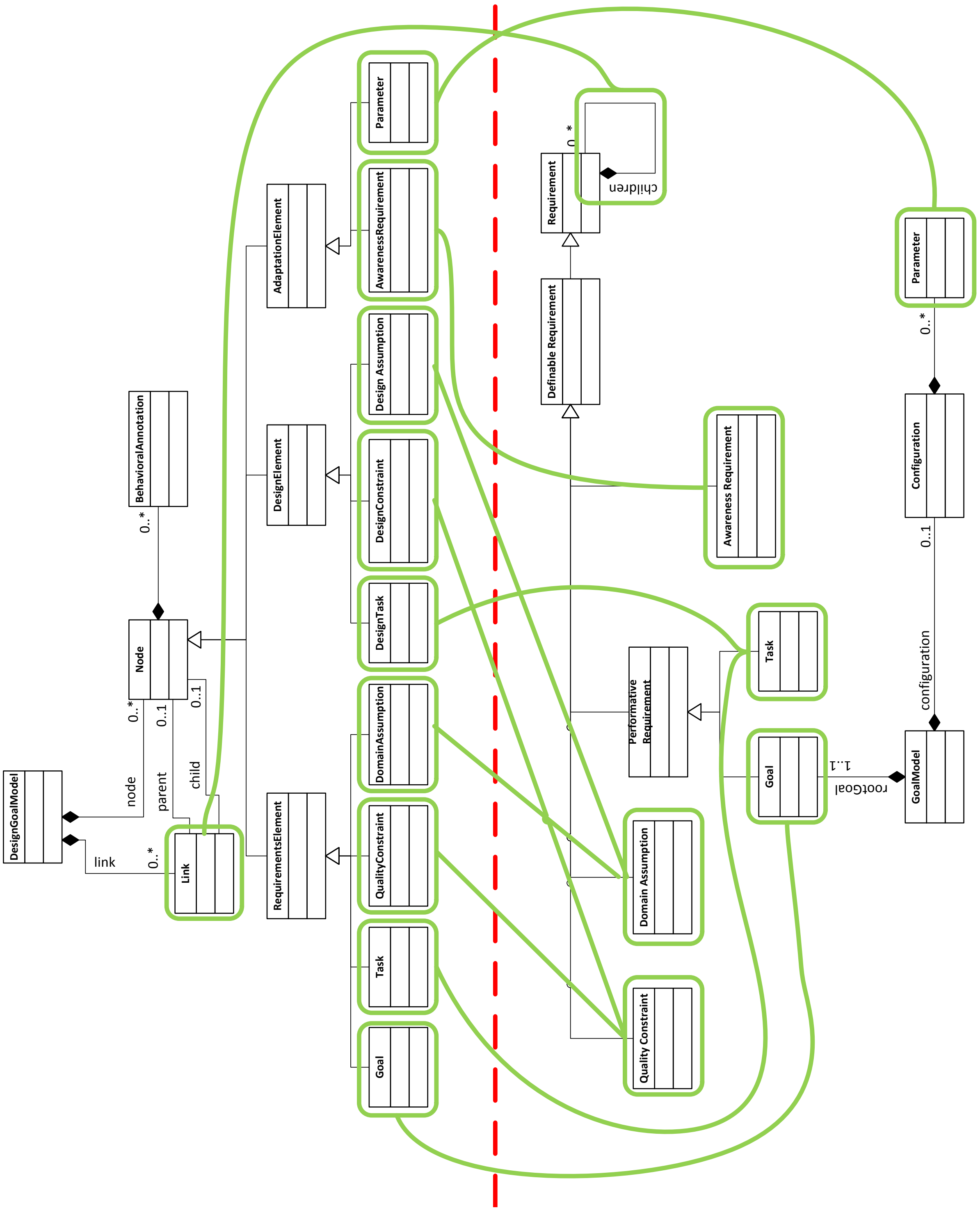


Figure 5.11: Mapping between the DGM metamodel (top) and Zanshin's metamodel (bottom)

5.4 Summary

The Goal to Architecture tool (GATO) developed to support the MULAS framework was presented in this chapter. It allows to create a requirements goal model and to include the refinements described in Chapter 4 which result in a Design Goal Model. In order to improve the visualization of the model, the tool provides different views: requirements, design, assignment and behavior. Moreover, it supports adaptation specification, the specification of transitions, and the automatic generation of statecharts from design goal models.

This tool was used to assist in the creation of the case studies presented in the next chapter, as well as for creating the design goal model diagrams depicted on this thesis.

6 Concept Proof

This chapter presents the application of the MULAS framework to the development of two different systems. These studies not only illustrate the use of the framework, but also provide early validation by showing that it is feasible to adopt this framework on different classes of system.

The first concept proof example adopts a system as described on the software engineering literature, namely the Automatic Teller Machine (ATM) system. A design goal model and a statechart of the ATM system were created with the MULAS architectural design process, along with its adaptation specification. Based on these models, *simulations* were executed in order to assess the adaptation mechanisms of the proposed framework.

The second example is a robotic system designed to monitor a specific environment. Unlike other systems presented on this thesis, this is a real system, developed through an R&D (research and development) initiative involving real academic and industrial partners, without participation of the author of this thesis. Through document analysis and interviews, we retroactively created artifacts from the MULAS framework, representing the system's requirements and design. Using a small scale robot it was possible to assess the execution of the system's adaptation cycle on a real scenario.

6.1 The ATM system

Aiming to better illustrate and evaluate the MULAS framework, its architectural design process was applied resulting in the creation of a statechart for the well-studied Automatic Teller Machine (ATM) system (WANG; MYLOPOULOS, 2009; WANG et al., 2010; BALSER et al., 2004; ROLLAND; ACHOUR, 1998; KOTONYA; SOMMERVILLE, 1996; CHOI; YEOM, 2002; GURP; BOSCH, 2002). In previous work, an implementation of this system was integrated with the Zanshin framework (TALLABACI; SOUZA, 2013), allowing us to concentrate, for this concept proof, on the generation of a behavioral model with support for adaptation mechanisms.

Fig. 6.1 shows a goal model with the requirements for an ATM system. This model

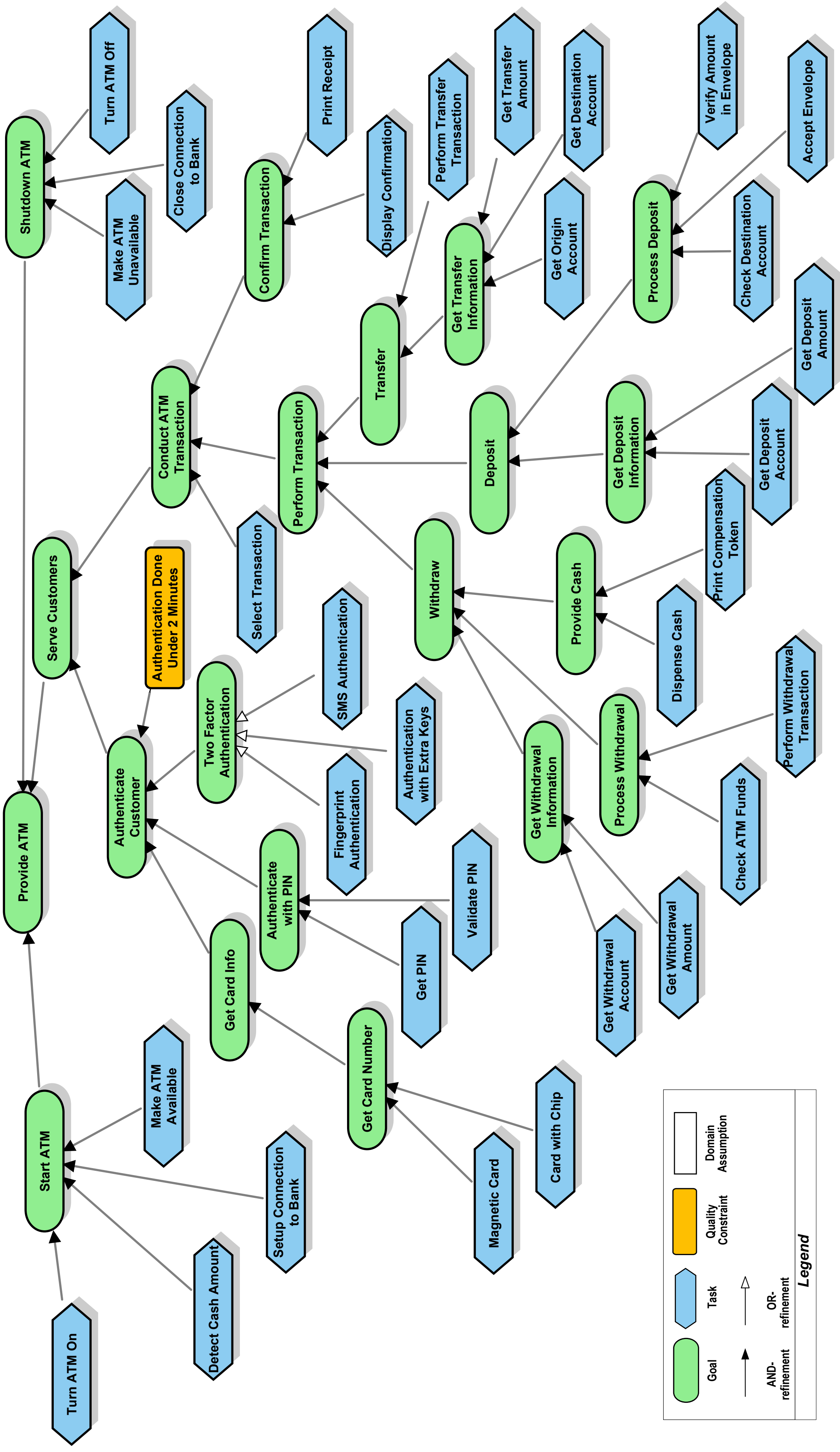


Figure 6.1: ATM Requirements Model

is based on a model from WANG; MYLOPOULOS (2009), which in its turn is based on the documentation and implementation from BJORK (2004). The ATM software system is embedded in an ATM machine, which is used by bank costumers one at a time in order to perform transactions such as withdrawal and deposit. The system is also used by operational staff from the bank, responsible for starting the ATM and for shutting it down. This operator also needs to insert bank notes in the machine, which are required for dispensing cash in withdrawal transactions. In order to provide this and other transactions, the system needs to know how much cash is available, as well as to setup its connection with the main banking system and then become available for use. At a latter point, the system can be shut down by an operator, which consists of the following tasks: *Make ATM Unavailable*, *Close Connection to Bank*, and *Turn ATM Off*.

The core of the system operation is expressed in the *Serve Customers* goal, for which the system needs to *Authenticate Customers* and *Conduct ATM Transactions*. The authentication relies on *Get Card Info*, *Authenticate with PIN* and *Two Factor Authentication*, where PIN stands for Personal Identification Number. Since the bank operates with different kinds of cards, the system needs to support both *Magnetic Cards* and *Cards with Chip*. The PIN is an input provided by bank customers, and then validated by the system. *Two Factor Authentication* provides an additional layer of security, since instead of only providing card and PIN users will also have to go through another identification step. For instance, through *Fingerprint Authentication*, *Authentication with Extra Keycodes*, or through electronic messages sent to customers' cell phones with *SMS Authentication*. d

In order to *Conduct ATM Transactions*, it is required to *Select Transaction*, *Perform Transaction* and *Confirm Transaction*. The transactions available to be performed are *Withdraw*, *Deposit* and *Transfer*. In order to perform these transactions, the system needs to receive information from the customer about the transaction, perform any required checking and validation, and then actually perform the transaction. The customer receives confirmation of the transaction through *Display Confirmation* or through *Print Receipt*.

While real ATM systems include many more additional transactions, this reduced set of functionalities based on TALLABACI; SOUZA (2013) represents the core of ATM systems, which includes: receiving and validating input from users, checking and processing requested transactions, performing requested transactions, and providing confirmation of performed transactions.

Building on the work described in TALLABACI (2012) and TALLABACI; SOUZA (2013), we present next the adaptation elements identified at the requirements level (Fig. 6.2): awareness requirements, (Table 6.1), parameters, and the relationships between the latter and the former. The *Detect Cash Amount* task is critical in order to enable withdrawal operations, thus it was established that it should never fail (AR1). Even more critical is to *Setup Connection to Bank*, since without it the transactions will not become available. Thus, it was also established that this task should never fail (AR2). Similarly, *Serve Customers* should fail at most twice in a month (AR3), while *Confirm Transaction* should never fail (AR4). Regarding the

Table 6.1: Description of the awareness requirements for the ATM System

Id	Awareness Requirement	Description
AR1	NeverFail	*Detect Cash Amount* task should never fail.
AR2	NeverFail	*Setup Connection to Bank* task should never fail.
AR3	MaxFailure(2,30d)	*Serve Customers* goal should fail at most twice on a 30 days' interval.
AR4	NeverFail	*Confirm Transaction* goal should never fail.
AR5	SuccessRate(90%,1d)	*Authentication Done Under 2 Minutes* quality constraint should have a success rate of at least 90% per day.
AR6	StateDelta(Undecided,*,30s)	*Select Transaction* task should be decided within 30 seconds.

Authentication Done in Under 2 Minutes quality constraint, it should succeed at least 90% of the time, measured daily (AR5). Lastly, there is a time limit to perform the *Select Transaction* task: 30 seconds (AR6).

While awareness requirements define the indicators of the system, i.e., what needs to be monitored, parameters define what can be changed in the system. For instance, consider the *Provide Cash* goal. The requirements model shows that the system is expected to be able to both *Dispense Cash* and *Print Compensation Token*; hence the AND-refinement. This compensation token is a receipt that the customer will receive when the transaction is not successful, allowing him to receive the requested amount from a bank clerk instead. Thus, at a single time, only one of these options will be selected. This selection is one of the parameters of the ATM system: **Cash or Task (CoT)**. The other parameters of this system are **Number of Operators Available (NOA)** and **Value of Daily Limit (VDL)**

The **NOA** parameter identifies how many operators are available for assisting bank costumers, answering doubts and thus potentially improving the success rate of the *Serve Customers* goal. This relationship is expressed by the following differential relation, using the shorthand notation described in Section 2.7:

$$\Delta\left(successRateOfServeCustomers/NOA\right)[0,10] > 0 \tag{6.1}$$

Bank customers have a limit for how much money they can withdraw in any given day. Low limits are good for security and operational purposes. However, low limits may reduce the successful rate of the *Withdraw* goal, since customers may not be able to withdraw as much as they want. This relationship between **Value of Daily Limit (VDL)** and *Withdraw* is expressed as

$$\Delta\left(successRateOfWithdraw/VDL\right)[0,+\infty) > 0 \tag{6.2}$$

Fig. 6.2 presents the complete requirements model for the ATM system, including its

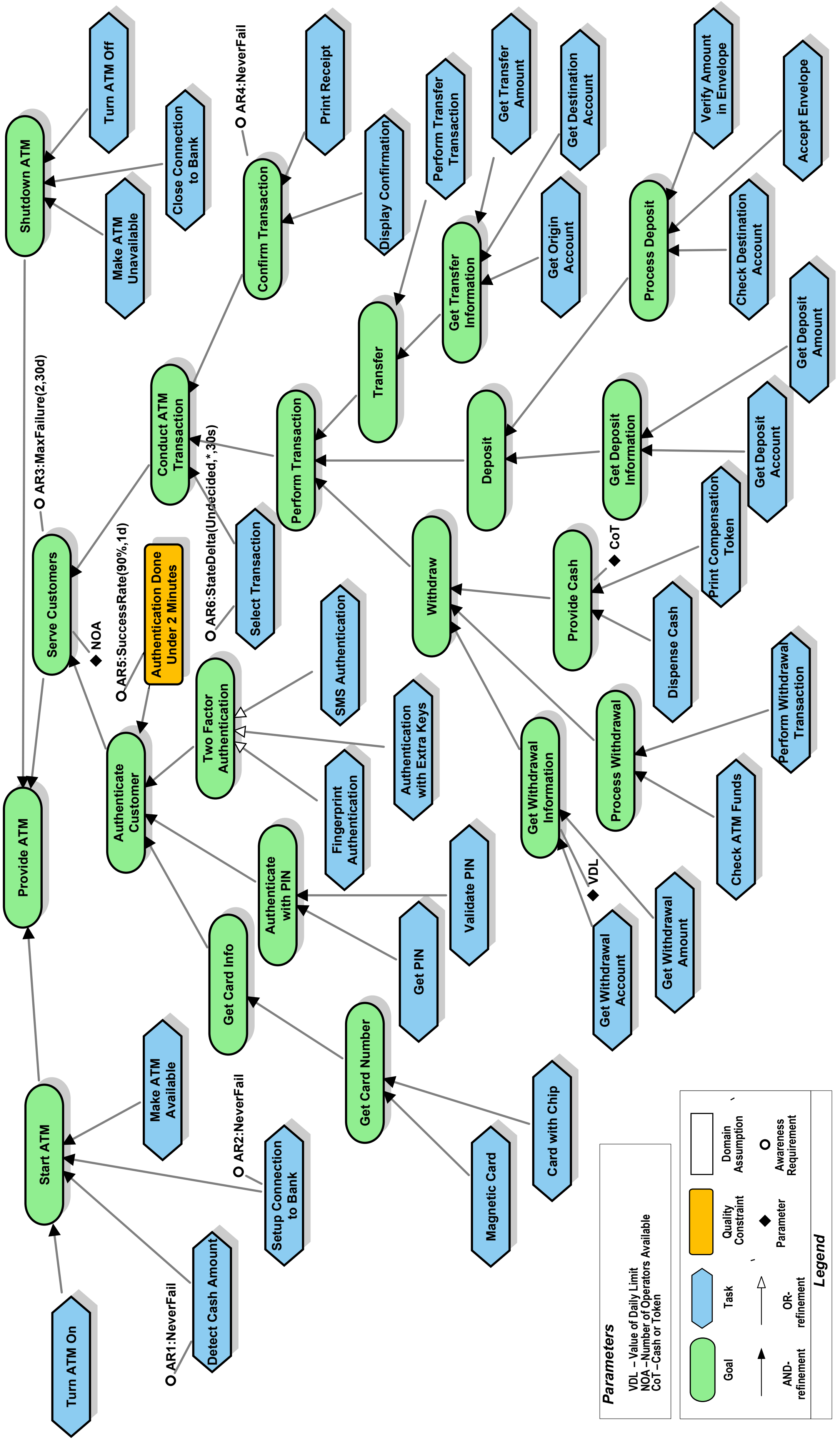


Figure 6.2: ATM Requirements Model with awareness requirements and parameters

awareness requirements and parameters. Based on this model, we performed an empirical evaluation of the MULAS framework, as described in the next subsection.

6.1.1 Evaluation using the ATM system

In this subsection we present an early evaluation of the MULAS framework using the ATM system. This evaluation aimed at identifying the suitability of the framework, verifying if its resulting artifacts were actually capable of enacting adaptation strategies on a simulated environment. This system was chosen considering that its goal model, awareness requirements, parameters, and adaptation actions were previously created by other authors (TALLABACI, 2012; TALLABACI; SOUZA, 2013). Ergo, instead of creating a requirements goal model from scratch, it was possible to focus solely on the architectural design process. Moreover, the profusion of scientific work exploring this class of systems not only suggests its relevance but also provided inspiration for additional adaptation scenarios.

This evaluation consisted of the following steps, which will be further described in the ensuing subsections.:

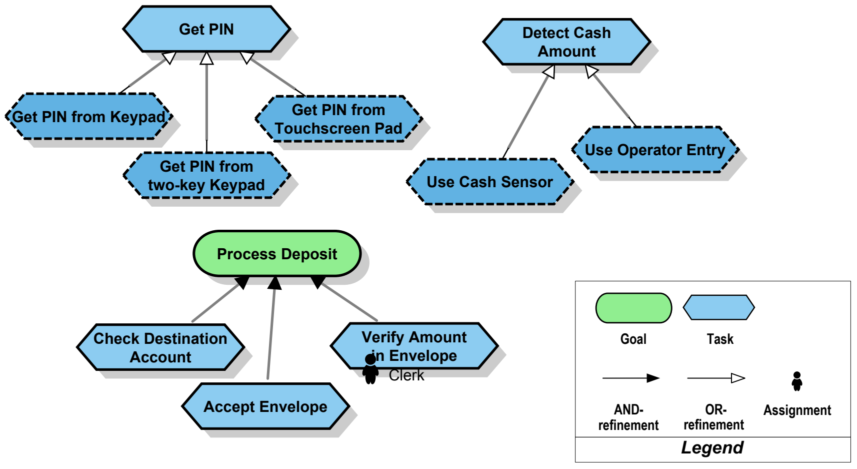
1. Apply the process described in Chapter 4 to the ATM requirements model, resulting in a Design Goal Model and a statechart;
2. Use a statechart modeling and simulation tool to enact the run-time states of the ATM based on the statechart obtained in step 1;
3. Integrate the statechart created in step 2 with the adaptation component of the Zanshin framework, thanks to the ability of the simulation tool to plug customized Java code to different parts of the model;
4. Execute adaptation scenarios, verifying that the proper adaptations are performed during simulation; i.e., verifying whether the adaptation strategies that were specified were actually enacted.

6.1.1.1 Step 1 - Following the Architectural Design process

Based on the requirements model of the ATM system (Fig. 6.2), a concept proof was performed following the eight steps of the process to move from requirements to statecharts for adaptive systems: *Identify design tasks, constraints and assumptions*; *Assign tasks*; *Define basic flows*; *Identify indicators, parameters and relations*; *Specify adaptation strategies*; *Generate base statechart*; *Specify transitions*; and *Include adaptation elements*.

The *Identify design tasks, constraints and assumptions* step is, in fact, a continuation of the AND-OR refinements of the requirements goal model. The difference is that instead of refining according to what is required by stakeholders (i.e., refining the problem), here the refinement is based on identifying how to make it possible to build a system that satisfies the requirements (i.e., define/refine the solution). This can take the form of AND refinements —

Figure 6.3: Excerpts of the ATM Design Model, with Design Tasks and Assignments



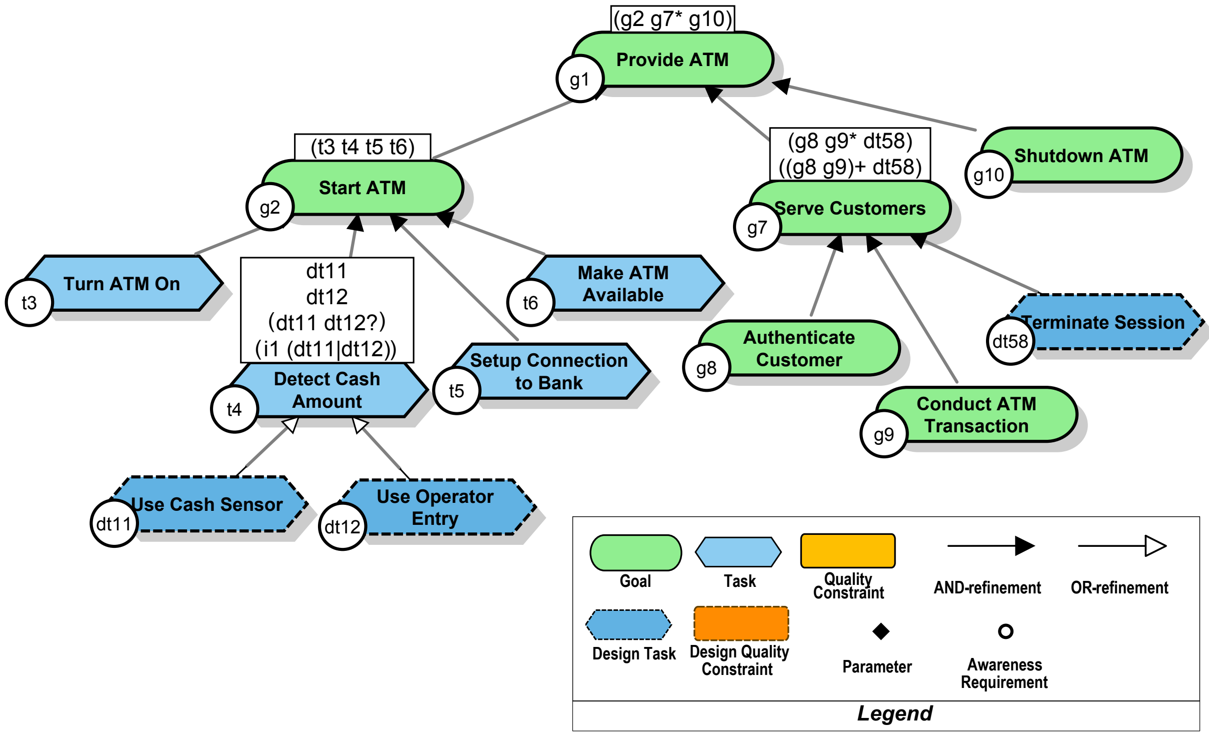
e.g., in order to provide *X*, the system needs to provide *Y* and *Z*. It can also take the form of OR refinements, specifying alternative ways of satisfying a requirement — e.g., *X* can be provided by providing *Y* or by providing *Z*. Concretely, these refinements will result in the inclusion of *design tasks*, *design constraints* and *design assumptions* in the goal model, which then becomes a *design* goal model.

By analyzing the requirements model of the ATM system and the existing literature on the subject, some elements of the goal model were further refined, such as *Get PIN*, *Detect Cash Amount*, and *Verify Amount in Envelope*. The *Get PIN* task was refined with three different alternatives for getting the PIN code: through a regular keypad, through a 2-key keypad, or through a touchscreen keypad. For the *Detect Cash Amount* task, two alternatives were identified: *Use Cash Sensor* and *Use Operator Entry*. In the latter option, the amount of money available in the ATM is tallied up by a bank employee; this information is then provided to the system. Fig. 6.3 presents excerpts of design refinements for the ATM system.

In the Brazilian banking system, the usual way of making deposits in an ATM is as follows: the customer gets a bank envelope, inserts bank notes on it and seal the envelope; the customer, interacting with the ATM and requesting a deposit transaction, inputs data such as target bank account and amount of money to deposit; during the operation, the ATM prompts the customer to insert the envelope and conclude the transaction; at a later moment, bank clerks gather the envelopes, check the amount of money in the envelopes is correct, and then confirm the transactions. That is why the *Verify Amount in Envelope* task was assigned to a bank clerk, as depicted in Fig. 6.3.

After identifying design elements and assigning tasks from the system, the next step is to *Define basic flows*. Fig. 6.4 shows the flow expressions for an excerpt of the ATM system. The flow expression from the root goal, *Provide ATM (g1)*, indicates that its behavior is defined as starting the ATM *(g2)*, then serving customers zero, one or more times *(g7)*, and finally shutting down the ATM *(g10)*. Similarly, the behavior of *Start ATM (g2)* is defined by the sequential execution of its children tasks, from *t3* to *t6*: *Turn ATM On (t3)*, *Detect Cash Amount (t4)*, *Setup*

Figure 6.4: Behavioral annotations for an excerpt of the ATM Design Model



Connection to Bank (t5), and *Make ATM Available (t6)*.

In the same way that OR refinements of goals can express alternative ways for achieving a goal, it is possible to define alternative behavioral refinements, which specify different flows for the system. This is the case of *Serve Customers (g7)*, with two alternative flow expressions, and *Detect Cash Amount (t4)*, with four alternative flow expressions, totaling eight different possible flows for the system execution. The requirements model dictates that to be able to *Serve Customers* the system must provide functionalities to *Authenticate Customer* and *Conduct ATM Transaction*. There are different behaviors for this goal, as shown in Fig. 6.4. The first option is to authenticate customers *(g8)* only once, and then conduct ATM transactions *(g9)* any number of times. The second option is to authenticate customers *(g8)* before conducting each transaction *(g9)*. Both options conclude with terminate session *(dt58)*.

This is also the case for *Detect Cash Amount (t4)*, which has four alternative behavioral refinements. The first option is to use only a cash sensor *(dt11)*, while the second option is to use only the entry of an operator *(dt12)*. The third option is to use the cash sensor and then use the operator entry only if it is necessary (e.g., if there is a malfunction on the cash sensor). The last alternative includes the use of an intermediate state *(i1)* where the operator can select whether to use the cash sensor or to entry the value manually.

In the next step of the architectural design process, *Identify indicators, parameters and relations*, two new parameters were identified (see Fig. 6.5):

- **CaD - Cash Detection —** This parameter defines which of the alternative behaviors of the *Detect Cash Amount* task (*t4*) will be executed: the first one, with only *Use Cash Sensor* (*dt11*) or the second one, with *Use Operator Entry* (*dt12*).

- **PMe - PIN Mechanism —** This parameter refers to the selection between three alternatives for *Get PIN*: *Get PIN from Keypad*, *Get PIN from two-key Keypad*, and *Get PIN from Touchscreen Pad*.

Continuing with the process, it is necessary to *Specify adaptation strategies* — i.e., to define how the system must react in case of failures. One of the adaptation scenarios that will be adopted in the concept proof here described concerns the *Detect Cash Amount* task (*t4*), which is OR-refined onto *Use Cash Sensor* (*dt11*) and *Use Operator Entry* (*dt12*), where *dt11* is the default option. Detecting how much money is available in the ATM terminal when it is turned on is essential for its proper operation, which is indicated by the awareness requirement AR1:NeverFail (Table 6.1). This selection is defined by the **CaD** parameter. The adaptation strategy for this awareness requirement specifies what to do in case of a failure of *t4*: to make sure it is not a temporary glitch, the ATM should *retry* with the cash sensor twice; if it still fails, *reconfigure* to use operator entry; finally, if manual entry also cannot satisfy the goal, the system should *abort*.

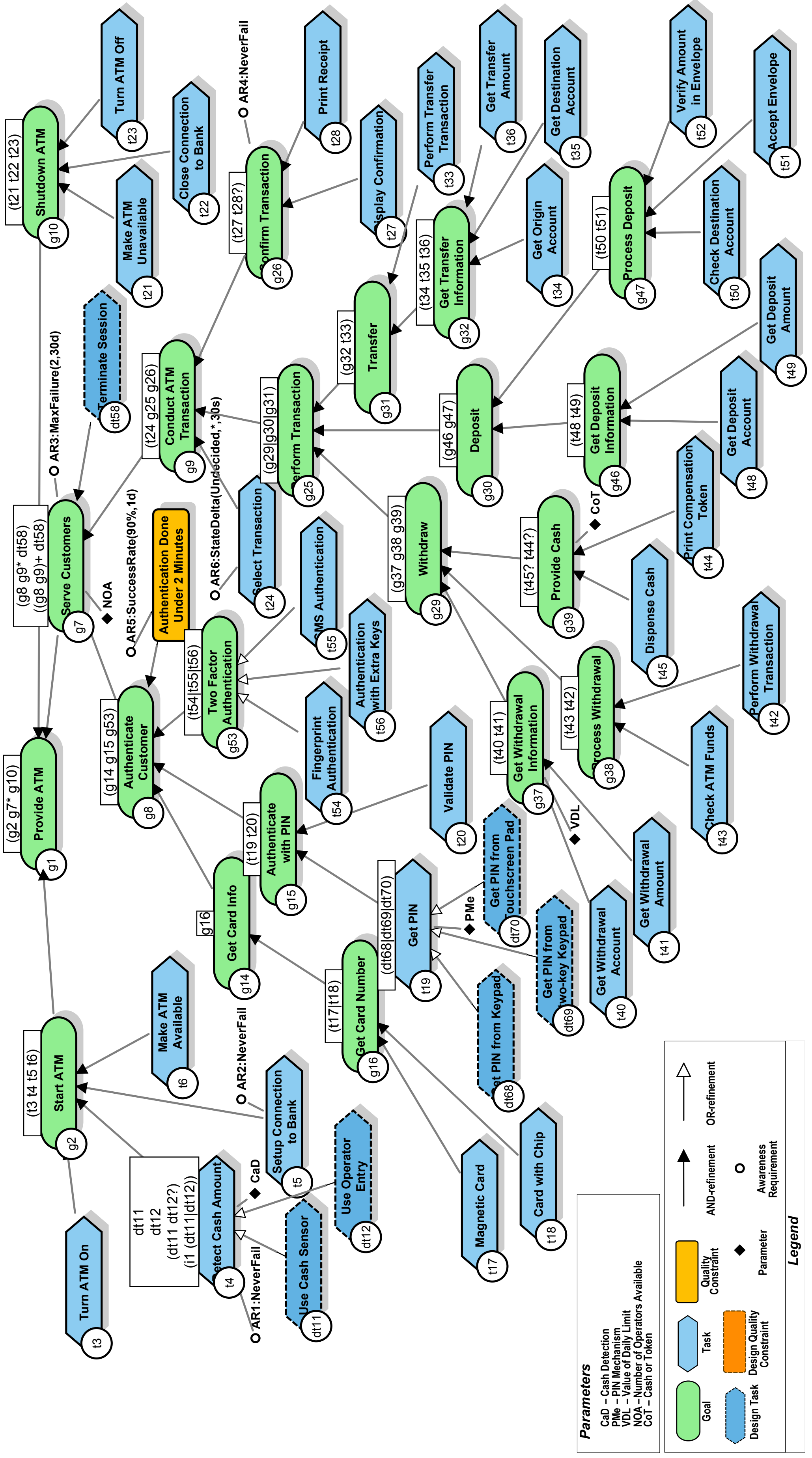
The *Setup Connection to Bank* task (*t5*) is also critical, which is represented by its *NeverFail* requirement (*AR2*) — if it is not successful, the ATM will not be able to perform any transaction. Thus, the following adaptation strategy was devised: in case of failure, the system must retry the execution of that task at most five times; if after 5 tries the connection has not been set up, the system must abort.

The adaptation strategy for *AR3*, which is related to *Serve Customers (g7)* consists of performing a reconfiguration and a delegation. The parameter related to AR3 is *NOA — Number of Operators*. Since the system itself is not able to increase the number of employees attending customers on an bank agency, the actual reconfiguration must be performed by a bank manager, hence the delegation.

For *AR4*, related to *Confirm Transaction (g26)*, and *AR5*, related to *Authentication Done Under 2 Minutes*, the adaptation strategies are *retry* and *abort*, respectively. Lastly, the adaptation strategy for *AR6*, which is related to *Select Transaction (t24)*, is to *abort*.

In order to verify the correct derivation of statecharts from the Design Goal Model, we used the *Goal to Arch* tool presented in Chapter 5, so as to perform the *Generate base statechart* step with the ATM system model.

When starting a statechart generation, the tool allows us to select between one or more flow expressions for those cases where alternative behaviors were defined. Once these behaviors are selected, the tool traverses the design goal model generating a combined flow expression. For instance, consider the *Deposit* goal (*g30*) shown in Fig. 6.5. Its flow expression is `g46 g47`. In its turn, the flow expression for *g46* (*Get Deposit Information*) is `t48 t49`, whereas the flow expression for *g47* (*Process Deposit*) is `t50 t51`. Thus, the combined flow expres-



g1 Provide ATM (g2 g7* g10)
g2 Start ATM (t3 t4 t5 t6)
t3 Turn ATM On
t4 Detect Cash Amount
dt11 dt12 (dt11 dt12?) (i1 (dt11|dt12))
dt11 Use Cash Sensor
dt12 Use Operator Entry
t5 Setup Connection to Bank
t6 Make ATM Available
AR1:NeverFail
AR2:NeverFail
CaD
g7 Serve Customers (g8 g9* dt58) ((g8 g9)+ dt58)
NOA
AR3:MaxFailure(2,30d)
dt58 Terminate Session
g8 Authenticate Customer (g14 g15 g53)
g14 Get Card Info g16
g16 Get Card Number (t17|t18)
t17 Magnetic Card
t18 Card with Chip
g15 Authenticate with PIN (t19 t20)
t19 Get PIN (dt68|dt69|dt70)
dt68 Get PIN from Keypad
dt69 Get PIN from Two-key Keypad
dt70 Get PIN from Touchscreen Pad
PMe
t20 Validate PIN
g53 Two Factor Authentication (t54|t55|t56)
t54 Fingerprint Authentication
t55 SMS Authentication
t56 Authentication with Extra Keys
Authentication Done Under 2 Minutes
AR5:SuccessRate(90%,1d)
g9 Conduct ATM Transaction (t24 g25 g26)
AR6:StateDelta(Undecided,*,30s)
t24 Select Transaction
g25 Perform Transaction (g29|g30|g31)
g26 Confirm Transaction (t27 t28?)
AR4:NeverFail
t27 Display Confirmation
t28 Print Receipt
g29 Withdraw (g37 g38 g39)
g37 Get Withdrawal Information (t40 t41)
VDL
t40 Get Withdrawal Account
t41 Get Withdrawal Amount
g38 Process Withdrawal (t43 t42)
t42 Perform Withdrawal Transaction
t43 Check ATM Funds
g39 Provide Cash (t45? t44?)
CoT
t44 Print Compensation Token
t45 Dispense Cash
g30 Deposit (g46 g47)
g46 Get Deposit Information (t48 t49)
t48 Get Deposit Account
t49 Get Deposit Amount
g47 Process Deposit (t50 t51)
t50 Check Destination Account
t51 Accept Envelope
t52 Verify Amount in Envelope
g31 Transfer (g32 t33)
g32 Get Transfer Information (t34 t35 t36)
t33 Perform Transfer Transaction
t34 Get Origin Account
t35 Get Destination Account
t36 Get Transfer Amount
g10 Shutdown ATM (t21 t22 t23)
t21 Make ATM Unavailable
t22 Close Connection to Bank
t23 Turn ATM Off
Parameters
CaD – Cash Detection
PMe – PIN Mechanism
VDL – Value of Daily Limit
NOA – Number of Operators Available
CoT – Cash or Token
Legend
Goal
Task
Quality Constraint
AND-refinement
OR-refinement
Design Task
Design Quality Constraint
Parameter
Awareness Requirement

Figure 6.5: ATM Design Goal Model with flow expressions

sion considering only the subtree of *g30* (*Deposit*) would be `t48 t49 t50 t51`.

While traversing the model, the tool also identifies the XOR-states and its sub-states resulting from the combination, which in this excerpt would be *g30(g46,g47), g46(t48,t49)* and *g47(t50,t51)* — i.e., *g30* is a XOR-state containing *g46* and *g47*; *g46* is a XOR-state containing *t48* and *t49*; and *g47* is a XOR-state containing *t50* and *t51*.

Listing 6.1: Output of the statechart derivation for the ATM system

```
XOR-states: g1(g2,g7,g10), g2(t3,t4,t5,t6), t4(dt11,dt12), g7(g8,g9,dt58),
   g8(g14,g15,g53), g14(g16), g16(t17,t18), g15(t19,t20), t19(dt68,dt69,
   dt70), g53(t54,t55,t56), g9(t24,g25,g26), g25(g29,g30,g31), g29(g37,g38,
   g39), g37(t40,t41), g38(t42,t43), g39(t44,t45), g30(g46,g47), g46(t48,
   t49), g47(t50,t51), g31(g32,t33), g32(t34,t35,t36), g26(t27,t28), g10(
   t21,t22,t23)
Atomic states: t3, dt11, dt12, t5, t6, t17, t18, dt68, dt69, dt70, t20, t54
   , t55, t56, t24, t40, t41, t43, t42, t45, t44, t48, t49, t50, t51, t34,
   t35, t36, t33, t27, t28, dt58, t21, t22, t23
Transitions: ~->t3, t3->dt11, t3->dt12, dt11->t5, dt12->t5, t5->t6, dt68->
   t20, dt69->t20, dt70->t20, t17->dt68, t17->dt69, t17->dt70, t18->dt68,
   t18->dt69, t18->dt70, t20->t54, t20->t55, t20->t56, t40->t41, t43->t42,
   t41->t43, t45->t44, t42->t45, t42->t44, t48->t49, t50->t51, t49->t50,
   t34->t35, t35->t36, t36->t33, t24->t40, t24->t48, t24->t34, t27->t28,
   t42->t27, t45->t27, t44->t27, t51->t27, t33->t27, t27->t24, t28->t24,
   t54->t24, t55->t24, t56->t24, t54->dt58, t55->dt58, t56->dt58, t27->dt58
   , t28->dt58, dt58->t17, dt58->t18, t6->t17, t6->t18, t21->t22, t22->t23,
    t6->t21, dt58->t21
Concurrent states: none
```

The complete Design Goal Model with the behavioral refinements of the ATM system is shown in Fig. 6.5. After selecting the first two behavioral alternatives for *Detect Cash Amount* (*t4*) and the first behavioral alternative for *Serve Customers* (*g7*), the resulting combined expression of the ATM system is

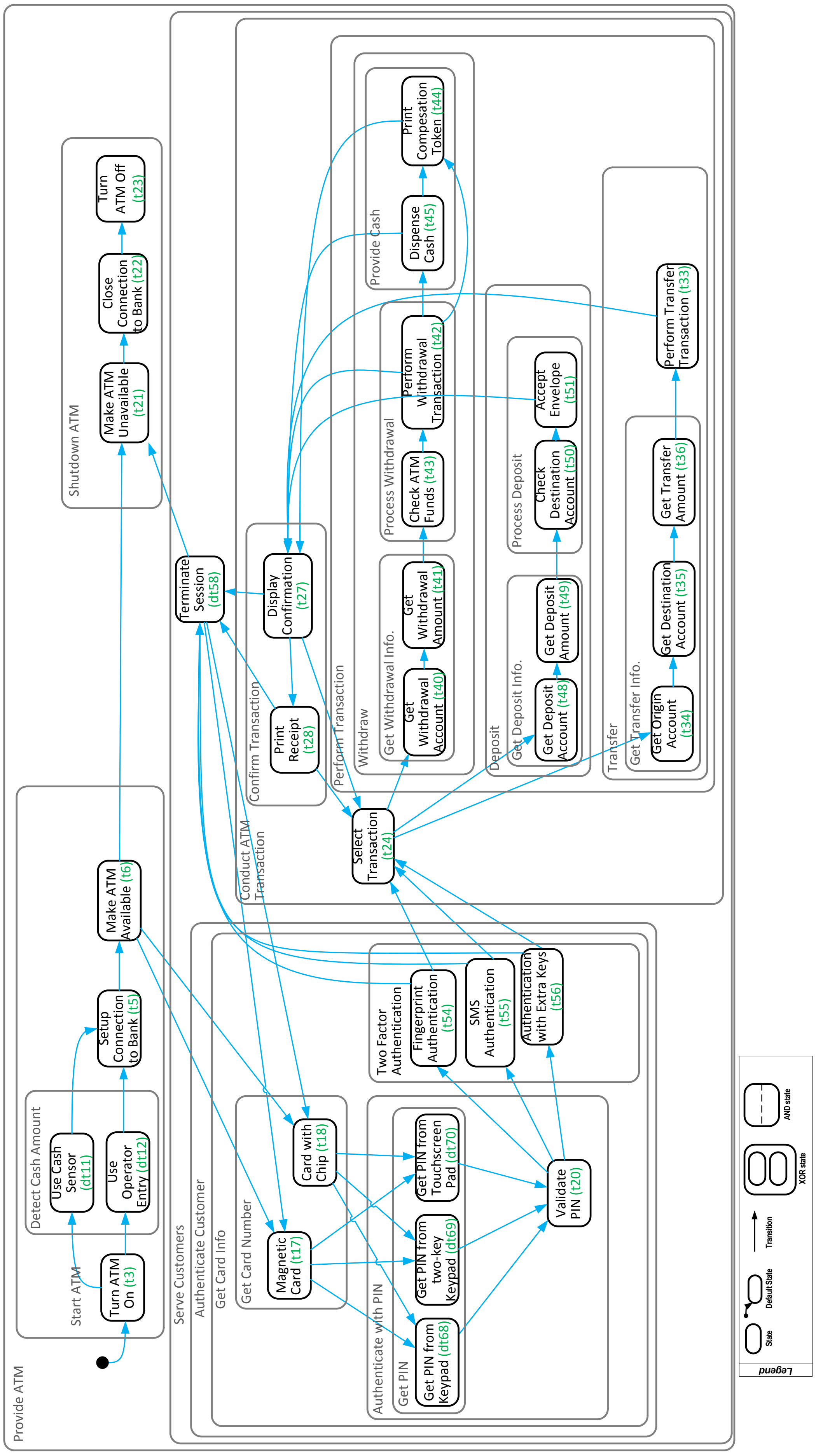
```
(t3 (dt11.dt12) t5 t6) ((((t17|t18)) ((dt68|dt69|dt70) t20)
(t54|t55|t56)) (t24 (((t40 t41) (t43 t42) (t45? t44?))|((t48
t49) (t50 t51))|((t34 t35 t36) t33)) (t27 t28?))* dt58)* (t21 t22
t23)
```

where a dot separates alternative behaviors.

This combined expression is not meant to be human-readable, inasmuch as it is somewhat large and unstructured. Instead, it is an intermediary artifact produced by the supporting tool when generating a statechart from the design goal model.

Once a combined flow expression is generated, it is then sent by the tool client to the tool server, where it will identify the states, transitions and AND-states (concurrent states). The output generated by the *Goal to Arch* tool, including the XOR-states, is presented in Listing 6.1, where a tilde (~) represents the initial state of the system. From this output, our statechart is defined as the tuple $\langle S,T,R\rangle$, where S is the union of the atomic states, the XOR states and the AND (concurrent) states — i.e., every state of the system; T is the set of transitions between states; and R is the union of the set of XOR state refinements and the set of AND state refinements — i.e., the hierarchical relationships between states. A manually created diagram for this statechart is depicted in Fig. 6.6.

Figure 6.6: Base statechart for the ATM System



This subsection described the enactment of the MULAS architectural design process for an ATM system, starting from a requirements model and moving towards statecharts, including its design goal model and the specification of adaptation strategies. Even though the process is described sequentially, step by step, in actuality the process was enacted iteratively and incrementally.

In the following subsections additional steps specific to the evaluation being presented on this section are presented, which includes a simulation of the system execution using a statechart tool.

6.1.1.2 Step 2 - Using a statechart tool

The second step of this evaluation was the creation of a model of the ATM statechart in a statechart tool, in order to be able to simulate the system execution at later steps. The selected tool was the Yakindu Statechart Tools (YST)[1], since it not only provides simulation capabilities but also provides syntactic validation, including the detection of unreachable states, as well as the generation of source code in Java, C, and C++. Moreover, it supports the utilization of Java code as part of the simulation, which is essential in order to integrate with the standard component from the Zanshin framework.

Since the *Goal to Arch* tool does not support any kind of integration with statechart tools, the model was created manually through Yakindu's user interface, based on the data presented on 6.1. After the statechart was modeled, additional procedures were still required in order to integrate it with Zanshin's component, as described in the following subsection.

6.1.1.3 Step 3 - Integration with Zanshin

The Zanshin framework includes a prototypal implementation of its reasoning/communication component. Its input includes information provided both offline (such as goal model and adaptation strategies) and online (execution data providing the activation and satisfaction status of goals, tasks, quality constraints and domain assumptions). With such information, it is able to identify the failure of awareness requirement and then select an adaptation strategy to execute, enacting the adaptation cycle described in Section 2.7.

The setup of Zanshin's component requires the execution of the following steps, as documented on its wiki[2]: install a required Integrated Development Environment (IDE) — namely, Eclipse[3]; create a metamodel based on the target system goal model; create an editor tool based on that metamodel; instantiate the metamodel, using the editor tool that was created; customize the Java communication code available in Zanshin's repository[4], which will be a part

[1]Yakindu Statechart Tools: http://statecharts.org/

[2]Zanshin's wiki: https://github.com/sefms-disi-unitn/Zanshin/wiki

[3]Eclipse is an integrated development environment with facilities to support, among others, programming and modeling activities.

[4]Zanshin's source-code repository: https://github.com/sefms-disi-unitn/Zanshin

of the target system; install and configure Zanshin's component.

Zanshin is based on the Eclipse Modeling Framework (EMF)[5]. As such, the goal model of the system, including Zanshin's specificities (awareness requirements, parameters, differential relations, etc.), is an XML file based on a metamodel of the system, which in turn is based on Zanshin's metamodel. The XML file representing the ATM goal model is shown in Listing 6.2.

Line **3** shows the root goal of this system: *Provide ATM*. Most other elements in the model will be children of the root goal or of its children (lines **4** to **114**), with the exception of parameters (lines **116** to **122**) and relations (line **123**). Goals, tasks, quality constraints, and domain assumptions are simple elements, structured accordingly to XML's tag hierarchy. For instance, the element for *Start ATM* starts on line **4** and only ends on line **12**. Thus, *Start ATM* is parent of *Turn ATM On* (line **5**) and *Detect Cash Amount* (line **6**), among others. By default, this parent-children relationship is an AND-refinement. OR-refinements must be explicitly stated through a *refinementType* attribute, as is the case of *Use Cash Sensor* and *Use Operator Entry* (lines **7** and **8**).

Awareness requirements are a little peculiar, since their relationship with another model element is defined by a *target* attribute. Awareness requirements may have two kinds of children: condition and strategies. For instance, AR4 (lines **106** to **114**) has a *SimpleResolutionCondition* (line **107**). With this kind of condition a failure is said to be solved when the next evaluation of that awareness requirement is successful, considering the adaptation cycle. Moreover, it contains a *RetryStrategy*, meaning that in case of failure, the adaptation strategy to be selected is *Retry*.

The version of Zanshin's component available during the preparation of this thesis only supports the definition of awareness requirements of the *NeverFail* type. For this reason, only the awareness requirements AR1, AR2, and AR4 are included on Listing 6.2 (lines **85** to **114**). Additionally, this version of Zanshin does not support the definition of enumerated parameters. That is why the parameters CaD (Cash Detection: *Cash Sensor* or *Operator Entry*), PMe (PIN Mechanism: *Regular Keypad*, *Two-Key Keypad*, or *Touchpad*), and CoT (Cash or Token: *Dispense Cash* or *Print Token*) are defined as *integer* instead of *enumerated*.

Lastly, it was also necessary to integrate the statechart simulation in YST with Zanshin's adaptation component. The integration with Zanshin was accomplished by defining Java methods that, when invoked by actions defined in the statechart, provide the current value of parameters, while also initiates the action of sending to Zanshin information about the system execution (monitoring points), which is required in order to determine the success or failure of awareness requirements. Additionally, the Java code monitors Zanshin's responses (such as *RetryAR1* and *AbortAR1*), firing the events required to perform the adaptation actions suggested by Zanshin, as defined in the *Include adaptation elements* sub-process of the architectural design process (Section 4.2.8). These methods are a part of the *AtmZanshinWrapper* class,

[5]Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/

Listing 6.2: ATM model as required by Zanshin's component

```
<?xml version="1.0" encoding="UTF-8"?>
<atm:AtmGoalModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.
    w3.org/2001/XMLSchema-instance" xmlns:atm="http://disi.unitn.it/zanshin/1.0/atm" xmlns:it.
    unitn.disi.zanshin.model="http://disi.unitn.it/zanshin/1.0/eca">
  <rootGoal xsi:type="atm:GProvideATM">
    <children xsi:type="atm:GStartATM">
      <children xsi:type="atm:TTurnOnATM"/>
      <children xsi:type="atm:TDetectCashAm">
        <children xsi:type="atm:TUseCashSens" refinementType="or"/>
        <children xsi:type="atm:TUseOperEntry" refinementType="or"/>
      </children>
      <children xsi:type="atm:TSetUpConnect"/>
      <children xsi:type="atm:TMakeATMAvail"/>
    </children>
    <children xsi:type="atm:GServeCust">
      <children xsi:type="atm:GAuthentCust">
        <children xsi:type="atm:GGetCardInfo">
          <children xsi:type="atm:GGetCardNumber">
            <children xsi:type="atm:TMagntCard"/>
            <children xsi:type="atm:TChipCard"/>
          </children>
        </children>
        <children xsi:type="atm:GAuthentWPin">
          <children xsi:type="atm:TGetPin">
            <children xsi:type="atm:TEnterPinFromK" refinementType="or"/>
            <children xsi:type="atm:TEnterPinFrom2K" refinementType="or"/>
            <children xsi:type="atm:TEnterPinFromTouch" refinementType="or"/>
          </children>
          <children xsi:type="atm:TValidatePin"/>
        </children>
        <children xsi:type="atm:GAuthentTwoFact">
          <children xsi:type="atm:TTwoFactFinger" refinementType="or"/>
          <children xsi:type="atm:TTwoFactExtra" refinementType="or"/>
          <children xsi:type="atm:TTwoFactSMS" refinementType="or"/>
        </children>
        <children xsi:type="atm:QAuthUnder2Min"/>
      </children>
      <children xsi:type="atm:GConductTrans">
        <children xsi:type="atm:TSelectTrans"/>
        <children xsi:type="atm:GPerfTrans">
          <children xsi:type="atm:GWithdraw">
            <children xsi:type="atm:GGetWithdInfo">
              <children xsi:type="atm:TGetWithdrAcc"/>
              <children xsi:type="atm:TGetWithdrAmo"/>
            </children>
            <children xsi:type="atm:GProcWithdraw">
              <children xsi:type="atm:TCheckAtmFunds"/>
              <children xsi:type="atm:TPerfWithdTrans"/>
            </children>
            <children xsi:type="atm:GProvideCash">
              <children xsi:type="atm:TDispCash"/>
              <children xsi:type="atm:TPrintCompTok"/>
            </children>
          </children>
          <children xsi:type="atm:GDeposit">
            <children xsi:type="atm:GGetDepInfo">
              <children xsi:type="atm:TGetDepAcc"/>
              <children xsi:type="atm:TGetDepAmo"/>
            </children>
            <children xsi:type="atm:GProcDep">
              <children xsi:type="atm:TCheckDestAcc"/>
              <children xsi:type="atm:TAcceptEnv"/>
              <children xsi:type="atm:TVerifyEnvelope"/>
            </children>
          </children>
          <children xsi:type="atm:GTransfer">
            <children xsi:type="atm:GGetTransInfo">
```

```
              <children xsi:type="atm:TGetFromAcc"/>
              <children xsi:type="atm:TGetToAcc"/>
              <children xsi:type="atm:TGetTransAmo"/>
            </children>
            <children xsi:type="atm:TPerfTransTrans"/>
          </children>
        </children>
        <children xsi:type="atm:GConfirmTrans">
          <children xsi:type="atm:TDisplay"/>
          <children xsi:type="atm:TPrintReceipt"/>
        </children>
      </children>
      <children xsi:type="atm:TTerminateSess"/>
    </children>
    <children xsi:type="atm:GShutAtm">
      <children xsi:type="atm:TMakeATMUnavail"/>
      <children xsi:type="atm:TCloseConnect"/>
      <children xsi:type="atm:TTurnOffATM"/>
    </children>
    <children xsi:type="atm:AR1" target="//@rootGoal/@children.0/@children.1">
      <condition xsi:type="it.unitn.disi.zanshin.model:SimpleResolutionCondition"/>
      <strategies xsi:type="it.unitn.disi.zanshin.model:RetryStrategy" time="5000">
        <condition xsi:type="it.unitn.disi.zanshin.model:
             MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="2"/>
      </strategies>
      <strategies xsi:type="it.unitn.disi.zanshin.model:ReconfigurationStrategy" algorithmId="
           qualia">
        <condition xsi:type="it.unitn.disi.zanshin.model:
             MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="1"/>
      </strategies>
      <strategies xsi:type="it.unitn.disi.zanshin.model:AbortStrategy">
        <condition xsi:type="it.unitn.disi.zanshin.model:
             MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="1"/>
      </strategies>
    </children>
    <children xsi:type="atm:AR2" target="//@rootGoal/@children.0/@children.2">
      <condition xsi:type="it.unitn.disi.zanshin.model:SimpleResolutionCondition"/>
      <strategies xsi:type="it.unitn.disi.zanshin.model:RetryStrategy" time="10000">
        <condition xsi:type="it.unitn.disi.zanshin.model:
             MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="3"/>
      </strategies>
      <strategies xsi:type="it.unitn.disi.zanshin.model:AbortStrategy">
        <condition xsi:type="it.unitn.disi.zanshin.model:
             MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="1"/>
      </strategies>
    </children>
    <children xsi:type="atm:AR4" target="//@rootGoal/@children.1/@children.1/@children.2">
      <condition xsi:type="it.unitn.disi.zanshin.model:SimpleResolutionCondition"/>
      <strategies xsi:type="it.unitn.disi.zanshin.model:RetryStrategy" time="2000">
        <condition xsi:type="it.unitn.disi.zanshin.model:
             MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="5"/>
      </strategies>
      <strategies xsi:type="it.unitn.disi.zanshin.model:AbortStrategy">
        <condition xsi:type="it.unitn.disi.zanshin.model:
             MaxExecutionsPerSessionApplicabilityCondition" maxExecutions="1"/>
      </strategies>
    </children>
  </rootGoal>
  <configuration>
    <parameters xsi:type="atm:CaD" unit="1" value="0" metric="integer"/>
    <parameters xsi:type="atm:PMe" unit="1" value="0" metric="integer"/>
    <parameters xsi:type="atm:VDL" unit="100" value="300" metric="real"/>
    <parameters xsi:type="atm:NOA" unit="1" value="1" metric="integer"/>
    <parameters xsi:type="atm:CoT" unit="1" value="0" metric="integer"/>
  </configuration>
  <relations indicator="//@rootGoal/@children.3" parameter="//@configuration/@parameters.0"
       lowerBound="0" upperBound="1"/>
</atm:AtmGoalModel>
```

presented on Listing 6.3.

Lines **4** to **17** are attributes whose values are modified according to instructions received from Zanshin's component. For instance, if an reconfiguration is performed by Zanshin, changing the *Value of Daily Limit* (*VDL*) parameter from *300* to *400*, the value of the *VDL* attribute will be modified accordingly (line **7**). Those attributes related to adaptation actions are likewise modified based on Zanshin's instructions. For example, supposing that Zanshin identified that the *AR2* awareness requirement has failed, triggering the *Retry* adaptation strategy: on this scenario, the value of the *retryAR2* attribute would be changed from *false* to *true* (line **14**).

Due to a limitation on Yakindu Statechart Tools (YST), the statechart simulation is not able to access these attributes directly. For this reason, this class also contains *getter* methods for every attribute — these methods simply return the value of the respective parameter (lines **60** to **95**). These methods are invoked during simulation as statechart actions, as shown on Fig. 6.7. For instance, the *CaD* method is invoked to ascertain whether the system should *Use Cash Sensor* or *Use Operator Entry* (Fig. 6.7-A). Similarly, the *abortAR1* method is invoked to verify if that specific adaptation strategy is active (Fig. 6.7-B). If, during a simulation, the system is in the *Setup Connection to Bank* state and that adaptation strategy is activated, then the respective transition will be executed.

The other kinds of methods shown on Fig. 6.7 are also part of the *AtmZanshinWrapper* class. The *startZanshin* method is a one-off method which must be invoked before interacting with Zanshin, hence its inclusion as an entry action on the first state of the system (Fig. 6.7-C). Its implementation (Listing 6.3, lines **20** to **26**) setups the communication with Zanshin and informs that a new session has been started. Lastly, monitoring methods are used to let Zanshin know the status of different awareness requirements. For instance, *informStartAR1* (Fig. 6.7-D) informs Zanshin that *AR1* is now active. The *informResultAR1* method, on the other hand, informs Zanshin that the goals, tasks, or design tasks related to that awareness requirement were concluded, either successfully or with failure. Information about success or failure must be included in the implementation of the method according to the simulation scenario to be executed (Listing 6.3, lines **33** to **38**).

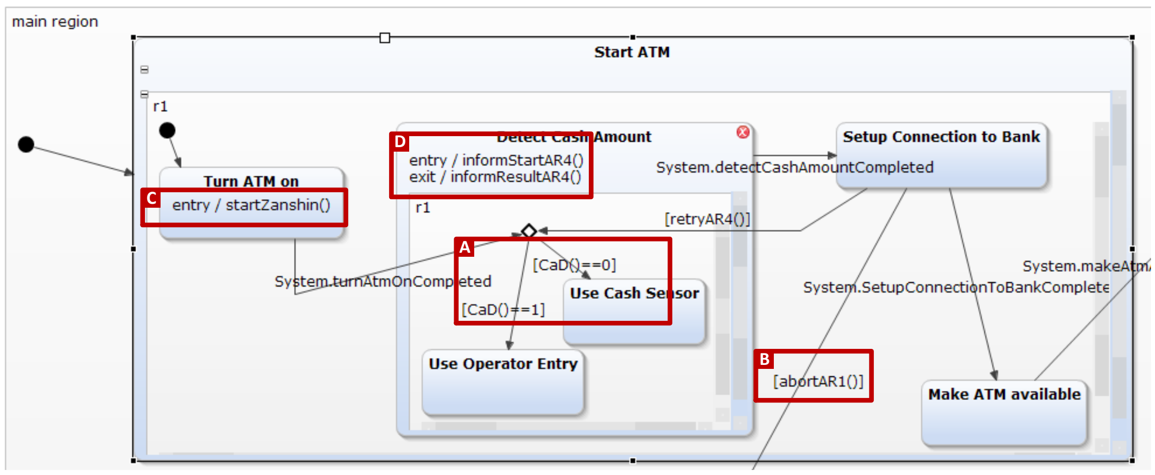
The adaptation cycle with Yakindu and Zanshin starts when Zanshin receives notifications about the execution of a task. If the notification indicates the failure of an awareness requirement, Zanshin then analyzes the goal model and the adaptation strategies of the system to identify the appropriate adaptation strategy to execute. The proposed adaptation strategy sent by Zanshin (if any) will be read by a Java method that process the received instruction and updates the proper attributes (Listing 6.4).

This method from Listing 6.4 is able to process three kinds of instructions: *INITIATE*, representing a *Retry* strategy; *APPLY_CONFIG*, which represents a *Reconfigure* strategy; and *ABORT*, representing its namesake. Lines **5** to **16** handle *Retry* strategies, setting the respective attributes from an *AtmZanshinWrapper* object. *Reconfigure* strategies are handle on lines **18** to **31**. On this case, the only parameter that may be reconfigured is *CaD* — *Cash Detection*, where

Listing 6.3: Source code of the *AtmZanshinWrapper* class

```java
public class AtmZanshinWrapper {
  private TargetSystemController controller;

  //parameters
  public int CaD = 0; //Cash Detection
  public int PMe = 0; //PIN Mechanism
  public double VDL = 300; //Value of Daily Limit
  public int NOA = 1; //Number of Operators Available
  public int CoT = 0; //Cash or Token

  //adaptation actions
  public boolean retryAR1 = false;
  public boolean abortAR1 = false;
  public boolean retryAR2 = false;
  public boolean abortAR2 = false;
  public boolean retryAR4 = false;
  public boolean abortAR4 = false;


  //method to start a new session with Zanshin's component (server)
  //this method is required in order to start the interaction with Zanshin
  public void startZanshin()
  {
    this.controller = TargetSystemController.getInstance(this);
    this.controller.startSession();
  }

  //Methods related to Monitoring
  public void informStartAR1()
  {
    this.controller.logRequirementStart(AtmRequirement.T_DETECT_CASH_AM);
  }
  public void informResultAR1()
  {
    //use logRequirementFailure if simulating a failure scenario
    //use logRequirementSuccess if simulating a success scenario
    this.controller.logRequirementFailure(AtmRequirement.T_DETECT_CASH_AM);
  }
  public void informStartAR2()
  {
    this.controller.logRequirementStart(AtmRequirement.T_SET_UP_CONNECT);
  }
  public void informResultAR2()
  {
    //use logRequirementFailure if simulating a failure scenario
    //use logRequirementSuccess if simulating a success scenario
    this.controller.logRequirementFailure(AtmRequirement.T_SET_UP_CONNECT);
  }
  public void informStartAR4()
  {
    this.controller.logRequirementStart(AtmRequirement.G_CONFIRM_TRANS);
  }
  public void informResultAR4()
  {
    //use logRequirementFailure if simulating a failure scenario
    //use logRequirementSuccess if simulating a success scenario
    this.controller.logRequirementFailure(AtmRequirement.G_CONFIRM_TRANS);
  }
  //Methods that check if its respective adaptation action was triggered
  public boolean retryAR1() {
    return retryAR1;
  }
  public boolean abortAR1() {
    return abortAR1;
  }
  public boolean retryAR2() {
    return retryAR2;
  }
  public boolean abortAR2() {
    return abortAR2;
  }
  public boolean retryAR4() {
    return retryAR4;
  }
  public boolean abortAR4() {
    return abortAR4;
  }

  //Methods to get the updated value of their respective parameters
  public int CaD() {
    return CaD;
  }
  public int PMe() {
    return PMe;
  }
  public double VDL() {
    return VDL;
  }
  public int NOA() {
    return NOA;
  }
  public int CoT() {
    return CoT;
  }
}
```

Figure 6.7: Screenshot of the simulation tool, with different kinds of actions highlighted: A) parameter-related actions; B) statechart actions related to adaptation action; C) initialization action; D) monitoring actions



'0' represents the option for *Use Cash Sensor* and '1' represents the selection of *Use Operator Entry*.

With this setup, it is possible to execute simulations with Zanshin using a statechart simulation tool, which allows to test feedback loops on adaptive systems before developing the target system itself. Details on the simulation executed with the ATM system are presented on the next subsection.

6.1.1.4 Step 4 - Simulation

With the preparation described on the previous subsection it was possible to execute simulations and verify appropriate responses to failure scenarios that should lead to adaptation. Given the limitations on the implementation of Zanshin's component, it was only possible to perform three of the adaptation scenarios identified on the first step of this concept proof (Section 6.1.1.1): those with *NeverFail* awareness requirements. The three scenarios are as follows.

The **first adaptation scenario** is related to *AR1*, which specifies that the *Detect Cash Amount* task should *NeverFail* (Fig. 6.5). The adaptation strategy for this awareness requirement specifies what to do in case of failure: to make sure it is not a temporary glitch, the ATM should *retry* with the cash sensor twice; if it still fails, *reconfigure* to use operator entry; finally, if manual entry also cannot satisfy the goal, the system should *abort* (Listing 6.2, lines **85** to **96**). Since the goal of this simulation is to verify if the system adapts correctly — i.e., if the correct adaptation strategies are selected and enacted —, we are assuming that all attempts of detecting the amount of cash available are going to fail. Thus, the expected behavior for this scenario is:

1. *Failure (Use Cash Sensor);*
2. *Retry;*
3. *Failure (Use Cash Sensor);*
4. *Retry;*
5. *Failure (Use Cash Sensor);*
6. *Reconfigure - Retry;*
7. *Failure (Use Operator Entry);*
8. *Abort;*

The **second adaptation scenario** refers *AR2*, which states that the *Setup Connection to Bank* task must *NeverFail* (Fig. 6.5). If the connection setup is not successful, the ATM will not be able to perform any transaction. Thus, the following adaptation strategy was specified: in case of failure, the system must retry the execution of that task at most three times; if after three tries the connection has not been set up, the system must abort (Listing 6.2, lines **97** to **105**). Assuming successive failures of *Setup Connection to Bank*, the expected behavior is:

1. *Failure (Setup Connection to Bank);*

Listing 6.4: Java method that receive instructions from Zanshin and modify attributes from *AtmZanshinWrapper* accordingly

```java
private void processAdaptationAction(AdaptationAction action) {
    log.info("Adaptation Thread processing action: {0}", action);
    String reqName = "";
    switch (action.getInstruction()) {
    case INITIATE:
      reqName = action.getParams()[1].toString();
      if (AtmRequirement.T_DETECT_CASH_AM.matches(reqName)) {
        wrapper.retryAR1 = true;
      }
      else if (AtmRequirement.T_SET_UP_CONNECT.matches(reqName)) {
        wrapper.retryAR2 = true;
      }
      else if (AtmRequirement.G_CONFIRM_TRANS.matches(reqName)) {
        wrapper.retryAR4 = true;
      }
      break;

    case APPLY_CONFIG:
      @SuppressWarnings("unchecked")
      Map<String, String> newConfig = (Map<String, String>)action.getParams()[0];
      if (newConfig.containsKey(AtmRequirement.CaD.getName())) {
        String value = newConfig.get(AtmRequirement.CaD.getName());
        try {
          float temp = Float.parseFloat(value);
          wrapper.CaD = Math.round(temp);
        } catch (NumberFormatException e) {
          log.error("Parameter value cannot be parsed: " + value);
        }
        wrapper.retryAR1 = true;
      }
      break;

    case ABORT:
      reqName = action.getParams()[1].toString();
      if (AtmRequirement.AR1.matches(reqName)) {
        wrapper.retryAR1 = false;
        wrapper.abortAR1 = true;
      }
      else if (AtmRequirement.AR2.matches(reqName)) {
        wrapper.retryAR2 = false;
        wrapper.abortAR2 = true;
      }
      else if (AtmRequirement.AR4.matches(reqName)) {
        wrapper.retryAR4 = false;
        wrapper.abortAR4 = true;
      }
      break;

    default:
      break;
    }
  }
```

2. *Retry;*
3. *Failure (Setup Connection to Bank);*
4. *Retry;*
5. *Failure (Setup Connection to Bank);*
6. *Retry;*
7. *Failure (Setup Connection to Bank);*
8. *Abort;*

Lastly, *AR4* specifies that the *Confirm Transaction* goal should *NeverFail* (Fig. 6.5). The adaptation strategy for this awareness requirement is to *retry*, with a limit of five retries. If the confirmation is not successful after these retries, the system should *abort* (Listing 6.2, lines **106** to **114**). Thus, the expected behavior for this **third adaptation scenario** is as follows:

1. *Failure (Confirm Transaction);*
2. *Retry;*
3. *Failure (Confirm Transaction);*
4. *Retry;*
5. *Failure (Confirm Transaction);*
6. *Retry;*
7. *Failure (Confirm Transaction);*
8. *Retry;*
9. *Failure (Confirm Transaction);*
10. *Retry;*
11. *Failure (Confirm Transaction);*
12. *Abort;*

These adaptation scenarios were simulated through the Yakindu tool. Fig. 6.8 shows a screenshot of the tool running a simulation of the first adaptation scenario. The larger part of the tool display a statechart. During simulation, the current state the system is in is highlighted — for instance, on this screenshot the active state is *Use Cash Sensor*, highlighted in red[6].

The tool also provides a list of events from the statechart, shown in the top-right corner of the screenshot. The occurrence of an event is simulated by clicking on the event name. The tool will process the event and react accordingly, by triggering a transition. For instance, the current state of the system is *Use Cash Sensor*, which is a sub-state of *Detect Cash Amount*. There is a transition from *Detect Cash Amount* to *Setup Connection to Bank*, where the event that trigger this transition is *System.detectCashAmountCompleted*. Thus, if a user clicks on the *detectCashAmountCompleted* event positioned at the top-right area of the screen, the system will indeed transition from one state to another.

[6]In grayscale versions of the thesis the active state presents a darker tone.

Lastly, the bottom-right section of the screen (Fig. 6.8) shows a log provided by Zanshin's component. This log presents the messages received at and sent from Zanshin, as well as its internal reasoning. Through this log it is possible to analyze the adaptations taking place on the system.

Listings 6.5, 6.6, and 6.7 present excerpts of Zanshin's log that describe the execution of the first, second, and third adaptation scenarios, respectively. The full log for each scenario is available on appendices B.1, B.2, and B.3, respectively. All of these logs start from the beginning of a new session — i.e., when Zanshin's component is ready to handle adaptation for that particular system.

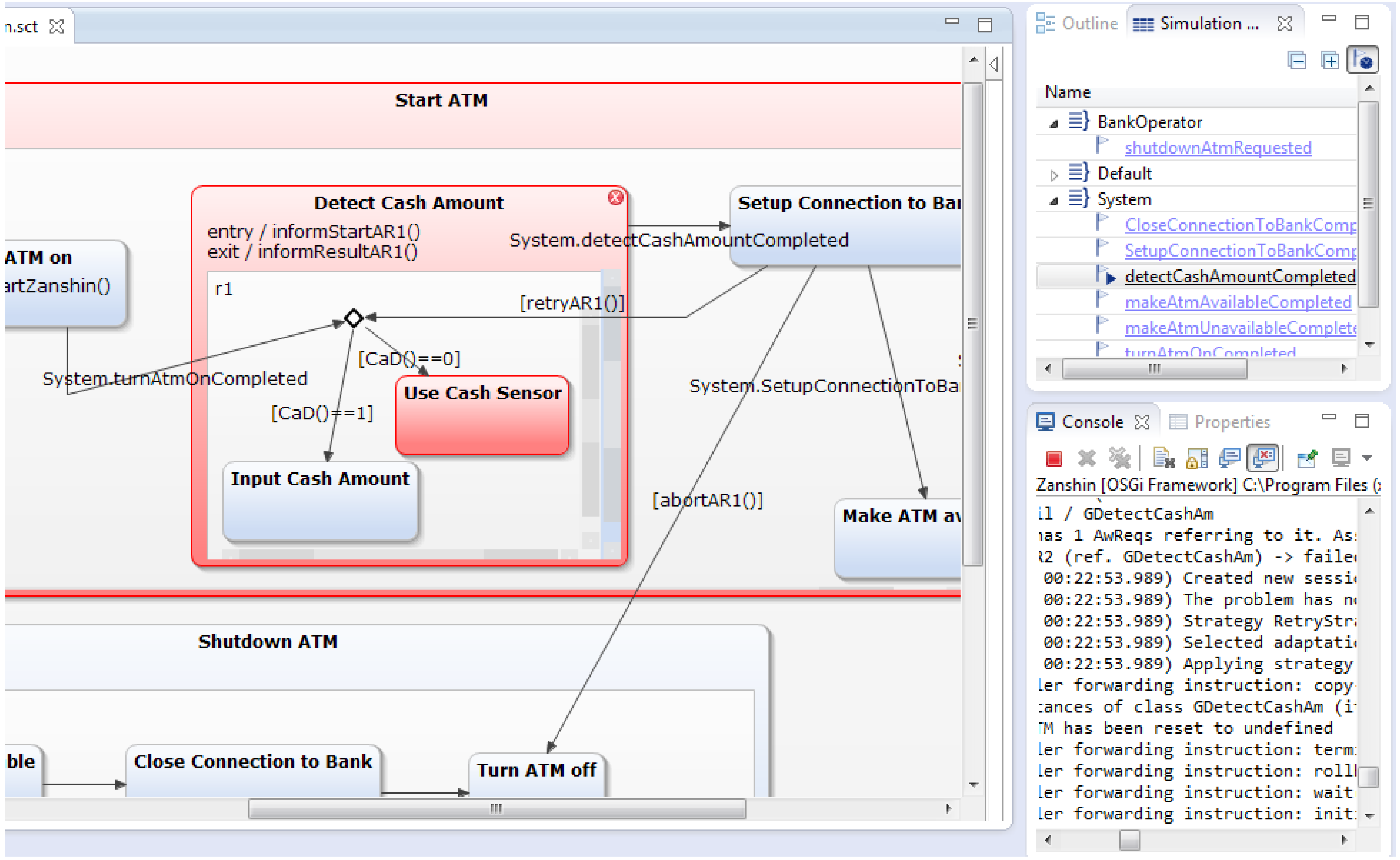
Listing 6.5 shows the log for the first adaptation scenario, which includes *AR1*, *CaD - Cash Detection*, and the following tasks: *Detect Cash Amount*, *Use Cash Sensor*, and *Use Operator Entry*. Lines **2** and **3** shows that Zanshin received the information that *Detect Cash Amount* has failed, and then it identifies that this failure is related to *AR1* (lines **4** and **5**). On lines **6** and **7** Zanshin selects to handle this failure with a *Retry* strategy. The system behavior described on lines **9** to **14** is exactly the same: *Detect Cash Amount* fails, Zanshin recognizes that the failure is related to *AR1*, and then selects a *Retry* strategy.

The beginning of the next set of lines is also the same: on lines **16** to **19** *Detect Cash Amount* fails and Zanshin recognizes that the failure is related to *AR1*. However, this time Zanshin informs that "RetryStrategy is not applicable because it has been applied at least 2 time(s) this session" (line **20**). Thus, a *Reconfiguration* strategy is selected (lines **21** and **22**), leading to a change on the *CaD* parameter (lines **23** to **25**). Please note that the value of *CaD* determines which task is executed between *Use Cash Sensor* and *Use Operator Entry*, as shown on Fig. 6.7-A. Thus, while before the default task for *Detect Cash Amount* was being performed (*Use Cash Sensor*), with the change of *CaD* the *Use Operator Entry* will be performed instead.

In the following lines, it is shown that *Detect Cash Amount* has failed again, which is recognized as being related to *AR1* (lines **27** to **30**). Neither *Retry* nor *Reconfiguration* is applicable now, as per condition rules defined on the lines **88** and **91** of Listing 6.2. Thus, Zanshin selects an *Abort* strategy (Listing 6.5, lines **31** to **34**). Since there is no more possible adaptation action for *AR1*, Zanshin terminates its session.

On the second adaptation scenario, Listing 6.6, the task under focus is *Setup Connection to Bank*, which is related to *AR2*. On lines **2** and **3** Zanshin is informed of the failed execution of *Setup Connection to Bank*. It then recognizes this failure as related to *AR2* (lines **4** and **5**), and then selected a *Retry* strategy. This behavior repeats on lines **9** to **14**, as well as on lines **16** to **21**. After the third retry this strategy is abandoned (line **27**). Instead, the *Abort* strategy is selected.

Figure 6.8: Screenshot of the simulation tool, displaying the current states (left), the list of events to trigger (top-right) and Zanshin's execution log (bottom-right)



```
il / GDetectCashAm
has 1 AwReqs referring to it. As
R2 (ref. GDetectCashAm) -> faile
 00:22:53.989) Created new sessi
 00:22:53.989) The problem has n
 00:22:53.989) Strategy RetryStr
 00:22:53.989) Selected adaptati
 00:22:53.989) Applying strategy
ler forwarding instruction: copy
ances of class GDetectCashAm (i
TM has been reset to undefined
ler forwarding instruction: term
ler forwarding instruction: roll
ler forwarding instruction: wait
ler forwarding instruction: init
```

Listing 6.5: Log excerpt of the execution of the first adaptation scenario (*AR1*)

```
INFO: Successfully created a new user session for target system atm: 1.422.716.236.486
DEBUG: Received log for life-cycle method call in session atm/1.422.716.236.486: TDetectCashAm.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.716.236.486: TDetectCashAm.FAIL()
INFO: Requirement TDetectCashAm has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR1 (ref. TDetectCashAm) -> failed
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Strategy RetryStrategy is applicable.
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.716.236.486: TDetectCashAm.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.716.236.486: TDetectCashAm.FAIL()
INFO: Requirement TDetectCashAm has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR1 (ref. TDetectCashAm) -> failed
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Strategy RetryStrategy is applicable.
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.716.236.486: TDetectCashAm.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.716.236.486: TDetectCashAm.FAIL()
INFO: Requirement TDetectCashAm has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR1 (ref. TDetectCashAm) -> failed
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Strategy RetryStrategy is not applicable because it has been applied at least
      2 time(s) this session.
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Strategy ReconfigurationStrategy is applicable.
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Selected adaptation strategy: ReconfigurationStrategy
INFO: Parameters chosen: [CaD]
INFO: Values to inc/decrement in the chosen parameters: [1.00000]
INFO: Produced new configuration with 1 changed parameter(s)

DEBUG: Received log for life-cycle method call in session atm/1.422.716.236.486: TDetectCashAm.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.716.236.486: TDetectCashAm.FAIL()
INFO: Requirement TDetectCashAm has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR1 (ref. TDetectCashAm) -> failed
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Strategy RetryStrategy is not applicable because it has been applied at least
      2 time(s) this session.
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Strategy ReconfigurationStrategy is not applicable because it has been applied
       at least 1 time(s) this session.
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Strategy AbortStrategy is applicable.
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) Selected adaptation strategy: AbortStrategy
INFO: (Session: AR1 / 2015-01-31 11:57:16.523) The problem has been solved or there is nothing else to try. Adaptation
      session will be terminated.
```

Listing 6.6: Log excerpt of the execution of the second adaptation scenario (*AR2*)

```
INFO: Successfully created a new user session for target system atm: 1.422.718.157.453
DEBUG: Received log for life-cycle method call in session atm/1.422.718.157.453: TSetUpConnect.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.157.453: TSetUpConnect.FAIL()
INFO: Requirement TSetUpConnect has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR2 (ref. TSetUpConnect) -> failed
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) Strategy RetryStrategy is applicable.
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.718.157.453: TSetUpConnect.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.157.453: TSetUpConnect.FAIL()
INFO: Requirement TSetUpConnect has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR2 (ref. TSetUpConnect) -> failed
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) Strategy RetryStrategy is applicable.
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.718.157.453: TSetUpConnect.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.157.453: TSetUpConnect.FAIL()
INFO: Requirement TSetUpConnect has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR2 (ref. TSetUpConnect) -> failed
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) Strategy RetryStrategy is applicable.
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.718.157.453: TSetUpConnect.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.157.453: TSetUpConnect.FAIL()
INFO: Requirement TSetUpConnect has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR2 (ref. TSetUpConnect) -> failed
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) Strategy RetryStrategy is not applicable because it has been applied at least
      3 time(s) this session.
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) Strategy AbortStrategy is applicable.
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) Selected adaptation strategy: AbortStrategy
INFO: (Session: AR2 / 2015-01-31 12:29:17.490) The problem has been solved or there is nothing else to try. Adaptation
      session will be terminated.
```

The third adaptation scenario, on Listing 6.7 is very similar to the second scenario. The main difference is that the task involved is *Confirm Transaction*, related to *AR4*. Moreover, instead of three retries, now five retries are allowed. In fact, after five retries (lines **7**, **14**, **21**, **28**, and **35**) Zanshin informs that "RetryStrategy is not applicable because it has been applied at least 5 time(s) this session" (line **41**), selecting to *Abort* (lines **42** and **43**).

Constrasting the actual execution of these three adaptation scenarios with the expected behavior, it is possible to conclude that the simulation was performed successfully. Based on the MULAS artifacts, specially on its statechart, it was possible to simulate the execution of an adaptive system, which displayed a correct behavior — i.e., it performed all adaptation actions according to what was specified.

The preparation for this simulation took between three and four hours, due to the need to manually prepare three different models: Zanshin requires the goal model to be expressed as a metamodel (i), which describe the elements of the goal model, as well as a model instance (ii) describing the relationships between elements; also, it was necessary to create a statechart diagram (iii) with the Yakindu tool, based on the output of the *Goal to Arch* tool (Chapter 5). Moreover, the integration with Zanshin's component requires additional programming, which also requires some effort. This time span of three to four hours prevents quick modeling-simulation iterations. However, based on this experience, we believe it to be possible to automate most of these steps, which may drastically reduce the time required to perform this kind of simulation.

Listing 6.7: Log excerpt of the execution of the third adaptation scenario (*AR4*)

```
INFO: Successfully created a new user session for target system atm: 1.422.718.724.215
DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.FAIL()
INFO: Requirement GConfirmTrans has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR4 (ref. GConfirmTrans) -> failed
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Strategy RetryStrategy is applicable.
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.FAIL()
INFO: Requirement GConfirmTrans has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR4 (ref. GConfirmTrans) -> failed
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Strategy RetryStrategy is applicable.
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.FAIL()
INFO: Requirement GConfirmTrans has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR4 (ref. GConfirmTrans) -> failed
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Strategy RetryStrategy is applicable.
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.FAIL()
INFO: Requirement GConfirmTrans has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR4 (ref. GConfirmTrans) -> failed
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Strategy RetryStrategy is applicable.
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.FAIL()
INFO: Requirement GConfirmTrans has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR4 (ref. GConfirmTrans) -> failed
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Strategy RetryStrategy is applicable.
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Selected adaptation strategy: RetryStrategy

DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.START()
DEBUG: Received log for life-cycle method call in session atm/1.422.718.724.215: GConfirmTrans.FAIL()
INFO: Requirement GConfirmTrans has 1 AwReqs referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state
      change: fail
INFO: Processing state change: AR4 (ref. GConfirmTrans) -> failed
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Strategy RetryStrategy is not applicable because it has been applied at least
      5 time(s) this session.
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Strategy AbortStrategy is applicable.
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) Selected adaptation strategy: AbortStrategy
INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The problem has been solved or there is nothing else to try. Adaptation
      session will be terminated.
```

6.2 The Environment Monitoring robot

In the remainder of this thesis we focused on the design of adaptive *software* systems. Here, we focus on an adaptive system on which hardware takes a central role: a robot for monitoring equipment temperature. For confidentiality reasons, we cannot provide further details about the context of this project.

The benefit of performing this concept proof is twofold. On one hand, it allows us to analyze the use of the MULAS framework within the context of hardware-intensive systems. On the other hand, it allows to analyze potential limitations of the framework when adopted for documenting real systems.

6.2.1 Requirements and Design

The main goal of this Environment Monitoring (EM) system is to *Monitor Environment* (Fig. 6.9). Since the environment it is supposed to monitor is dangerous, the client defined that the system should request *Minimal Intervention on the Site*, if any. After considering the possible options, it was decided to develop a robotic system (*Use a Robot* quality constraint), to be used under *Good Weather*.

The *Monitor Environment* goal is refined onto three sub-goals: *Provide History Data*, *Measure Temperature*, and *Handle Alerts*. The temperature measurement can be achieved through *Capture Thermal Image*, but first the robot needs to *Get in Position for Image Capture*. This captured image can then be used to *Calculate Temperature*.

Information captured by the robot can be accessed by end users, as part of the *Provide History Data* goal. In order to provide this capability, the system needs to *Record Captured Image* and *Record Temperature Data*. This information can then be viewed by users, as expressed by the *Display History Data* goal. The system may *Display Temperature per Equipment*, as well as *Display Images per Date*. For the latter, besides *Display Image* it is also possible to *Set Zoom*, *Set Color Contrast*, and *Set Color Palette*.

Based on the gathered information, the system must also *Handle Alerts*. After performing the *Analyze Temperature* and *Analyze Temperature Records* tasks, the system must *Display Alerts* in case the temperature of an equipment reaches a dangerous zone. Moreover, users may *Add Alert Manually*.

During design, the research team considered different options for the conception of this robotic system, such as using a robot that moves over a rail (*Rail Robot*) or an *Aerial Robot*. These alternatives refine the *Use a Robot* quality constraint (Fig. 6.10). In light of safety concerns and the *Minimal Intervention on the Site* constraint, a third alternative was selected: a *Wheeled Robot*.

Other design elements are included on Fig. 6.10. With respect to the *Generate Alert Automatically*, it was observed that it is necessary to persistently store the generated alert, which is

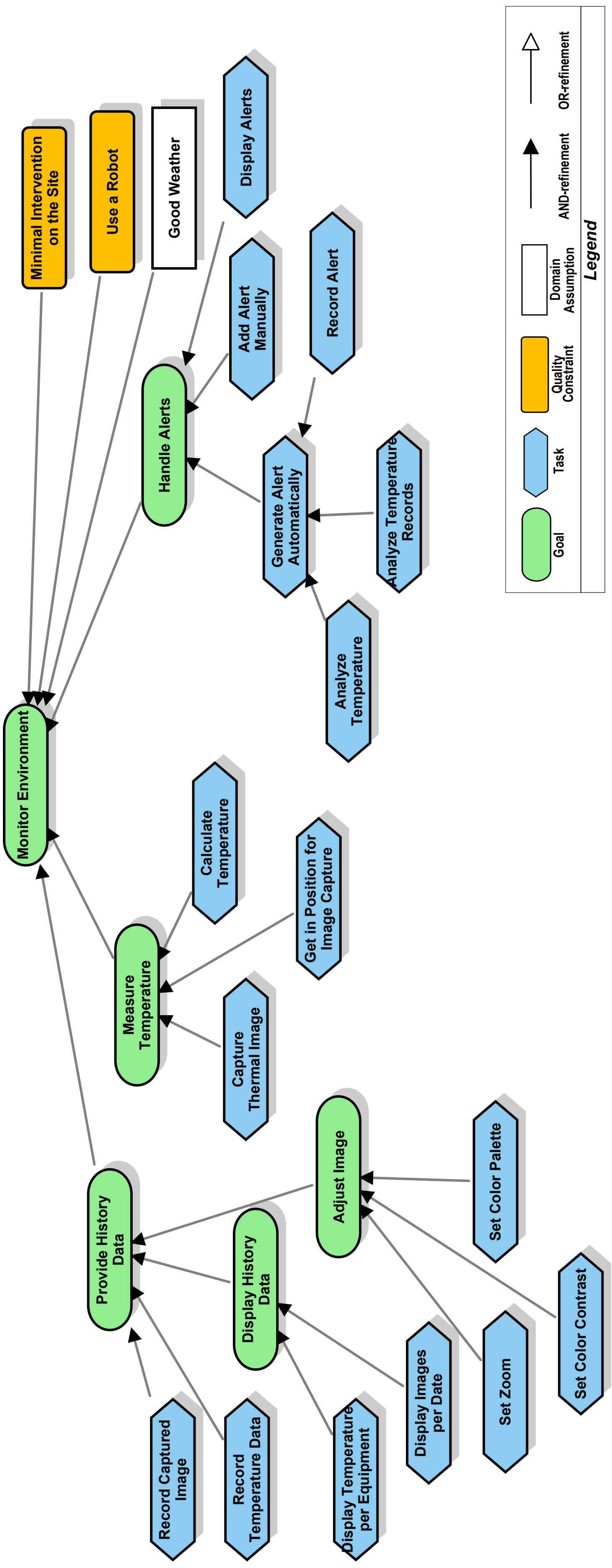


Figure 6.9: Requirements Model of the EM system

represented by the *Record Alert* design task. Furthermore, it was decided to allow the creation, modification, and removal of rules that define when an alert should be issued. This is represented by the *BREAD Alert Rules* design task, where BREAD stands for Browse, Read, Edit, Add, Delete.

A requirement that was extensively refined during the design process is the *Get in Position for Image Capture* task, which defines the robot's locomotion. The most basic (design) task is to *Move Robot*. It is also possible to *Orientate Camera*, which may either *Turn* or *Tilt Vertically*. This orientation is necessary for pointing the camera to the equipment that will be monitored. It is also necessary to *Navigate Robot*, which refers to controlling the robot's position.

The robot's navigation may be performed manually (*Navigate Manually*): through a *Control pad*, users may control the movement of the robot. Moreover, the system's *Graphical User Interface* also provides specific buttons for controlling the robot manually.

Automatic navigation allows the robot to move autonomously through a substation (*Navigate Automatically*). For this, an inspection route must be defined, through the *Edit Inspection Route* design task. One of the elements of this route is a set of operations points, which are positions over a pre-defined path (line) on a substation (*BREAD Operation Point*). These positions are detected (*Detect Position*) through RFID markers on the ground (*Detect RFID Markers*), as well as through the distance traversed by the robot (*Count Steps*).

The inspection route also includes equipments that need to be monitored (*BREAD Equipments*), specific parts of these equipments (*BREAD Equipment Part*), and regions of interest (*BREAD Region of Interest*). These regions of interest are rectangular areas on a picture defining which section of a picture represents an equipment part.

Lastly, it is relevant that the robot does not move towards forbidden zones. For this reason, the *Do not Deviate from Route* constraint was defined. The mechanism devised to satisfy this constraint is to make the robot follow a line marked on the floor (*Follow Line* design task).

Fig. 6.11 shows the flow expressions of the EM system. The resulting transitions can be seen on Listing 6.8, which represents the output provided by *Goal to Arch* tool (Chapter 5). The flow expression of *g1*, albeit reasonably large, can be analyzed by considering its three concurrent segments, which are separated with hyphens. Each one of these segments corresponds to a concurrent region on the system's statechart, shown at Fig. 6.12:

- **(i1 (t21 i1)* (dt32 i1)* (dt47 i1)* (g2 i1)* (dt34 i1)*)** — corresponds to the bottom region of the statechart. This region represents user-interaction with the system. From *Idle 1 (i1)*, a user may access different functionalities of the system: the system may *Display Alerts (t21)*; the user may control the robot through the system's *Graphical User Interface (dt32)*, Browse, Read, Add, Edit or Delete *(BREAD) Alert Rules (dt47)*; the system may *Provide History Data (g2)*; lastly, users may *Edit Inspection Route (dt34)*.

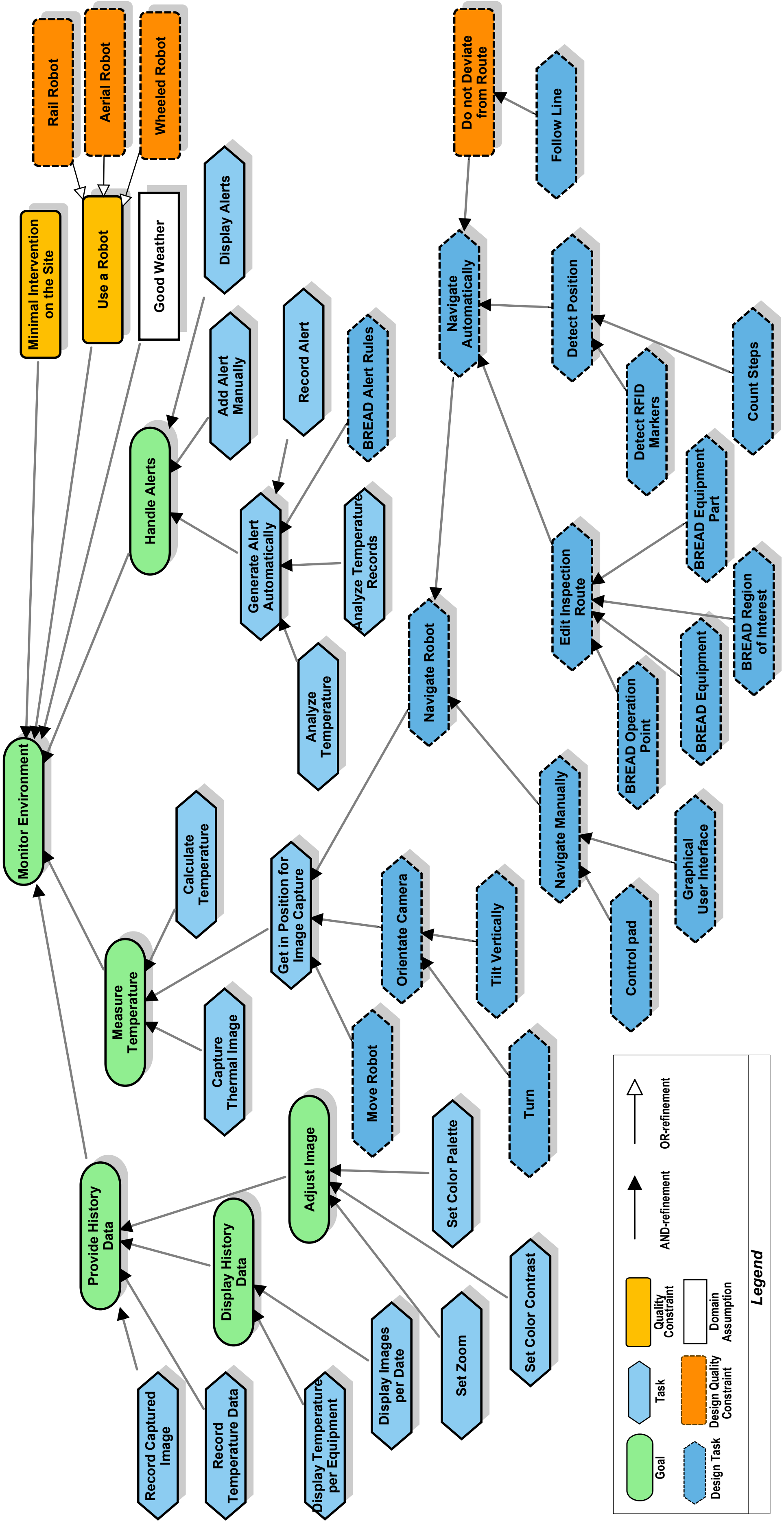


Figure 6.10: Design Goal Model of the EM system

- **(i2 t14 t4 t19) —** corresponds to the top left region of the statechart. This segment represents a cycle that is performed whenever a new thermal image is received: first *Calculate Temperature (t14)*, then *Record Temperature Data (t4)*, and lastly *Generate Alert Automatically (t19)*.
- **(i3 (dt25 i3)? (dt26 i3)? (dt33 ((dt26 t13 t3 dt25)|(dt26 dt41))?)*) —** corresponds to the top-right region of the statechart. This flow represents the main loop of the actual robot. If being controlled manually, the system may *Move Robot (dt25)* or *Orientate Camera (dt26)*. If on automatic, i.e., when executing an inspection, the robot will *Navigate Automatically (dt33)*. When it is on the right position, it will *Orientate Camera (dt26)*, *Capture Thermal Image (t13)*, *Record Captured Image (t3)*, and continue to move (*dt25*).

Whereas this section presents the general requirements and design elements of the EM robotic system, the following section describes its specific adaptation elements, such as awareness requirements, parameters, and adaptation strategies.

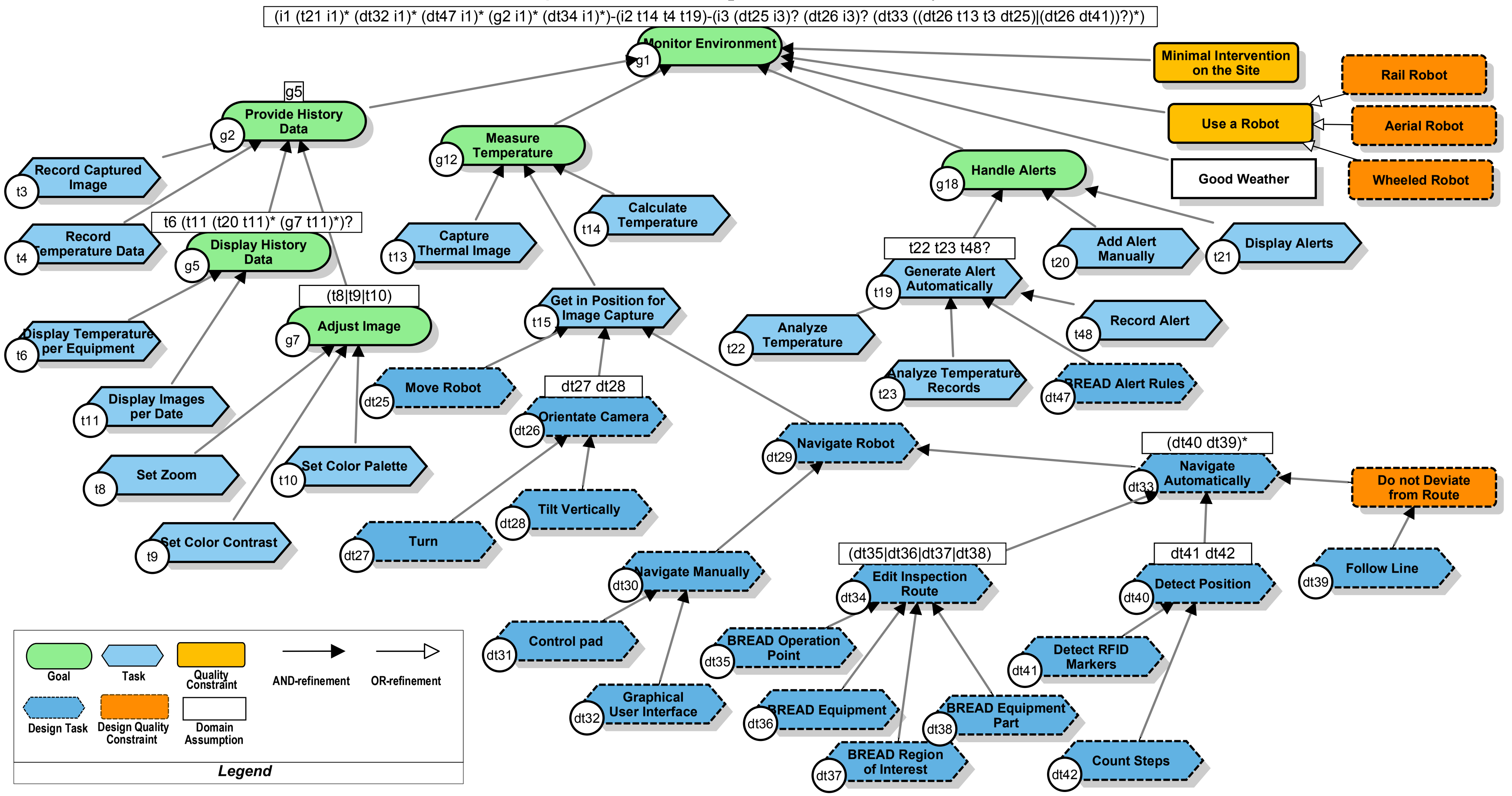
6.2.2 Adaptation

For this project, it is critical that the proposed robot does not go beyond the area designated for its movement. In the design goal model (Fig. 6.10), this constraint is expressed as the *Do not Deviate from Route* constraint. In order to satisfy this constraint, it was decided that the robot must follow a line marked on the floor (*Follow Line* design task). The act of following a line can be enacted by moving forward and then performing corrective actions — moving to the left or to the right — whenever the robot goes astray. In other words, the robot will adapt its behavior according to its positioning relative to a path (line).

It is possible to implement this adaptation using the component that is a part of the Zanshin framework. That component has the peculiarity of not being able to handle any information regarding a failure. For instance, if we were to define a *NeverFail* awareness requirement to the *Follow Line* task, it would not be possible to perform the correct adaptation, since we would not know if the robot should move to the right or to the left. For this reason, three additional design constraints were included in the model: *Do Not Deviate to the Right*, *Do Not Deviate to the Left*, and *Do Not Move if Away from the Line* (Fig. 6.13). A *NeverFail* awareness requirement was defined for each one of these constraints — respectively, *AR1*, *AR2*, and *AR3*.

Once what needs to be monitored is defined (deviation), it is also necessary to specify what happens if some correction is necessary (adaptation actions). For this scenario, it is possible to define a parameter that represents which corrective action must be performed: *MTP — Movement To Perform*. It is also necessary to define a *Change Parameter* action for each awareness requirement: if *Do Not Deviate to the Right* (*AR1*) fails, the value of *MTP* changes to *MoveToTheLeft*; if *Do Not Deviate to the Left* (*AR2*) fails, the value of *MTP* changes to *MoveToTheRight*; lastly, if *Do Not Move if Away from the Line* (*AR3*) fails, i.e., if the robot is moving

Figure 6.11: Flow Expressions of the EM system



Listing 6.8: Output of the statechart derivation for the EM system

```
Transitions: ~->i1, ~->i2, ~->i3, t21->i1, i1->t21, i1->t21, dt32->i1, i1->
   dt32, i1->dt32, i1->dt32, dt47->i1, i1->dt47, i1->dt47, i1->dt47, i1->
   dt47, t20->t11, t11->t20, t11->t20, t8->t11, t9->t11, t10->t11, t11->t8,
    t11->t9, t11->t10, t11->t8, t11->t9, t11->t10, t11->t8, t11->t9, t11->
   t10, t6->t11, t6->i1, t11->i1, t11->i1, t11->i1, i1->t6, i1->t6, i1->t6,
    i1->t6, i1->t6, dt35->i1, dt36->i1, dt37->i1, dt38->i1, i1->dt35, i1->
   dt36, i1->dt37, i1->dt38, i1->dt35, i1->dt36, i1->dt37, i1->dt38, i1->
   dt35, i1->dt36, i1->dt37, i1->dt38, i1->dt35, i1->dt36, i1->dt37, i1->
   dt38, i1->dt35, i1->dt36, i1->dt37, i1->dt38, i1->dt35, i1->dt36, i1->
   dt37, i1->dt38, i2->t14, t14->t4, t22->t23, t23->t48, t4->t22, dt25->i3,
    i3->dt25, dt27->dt28, dt28->i3, i3->dt27, i3->dt27, dt41->dt42, dt42->
   dt39, dt39->dt41, dt27->dt28, dt28->t13, t13->t3, t3->dt25, dt27->dt28,
   dt28->dt41, dt39->dt27, dt39->dt27, dt39->dt41, dt25->dt41, dt39->dt41,
   i3->dt41, i3->dt41, i3->dt41
```

far from the designated path, the value of *MTP* changes to *Stop*. If there is no failure, the robot is allowed to move forward.

The following subsection describes the enactment of this adaptation on a real, small scale robot, which provided positive results.

6.2.3 Experimentation

In order to evaluate whether it is feasible to use the MULAS framework on a hardware-intensive system, as well as on the context of a real project, this experiment consisted of enacting the adaptation cycle described in Section 6.2.2 on a real robot, using the adaptation component included in the Zanshin framework.

Partnering with the research team that developed the EM robot, we assembled a smaller scale robot for experimentation. On this robot we adopted the design and the operating system of an early prototype that was used as a concept proof for the EM robot. Since this smaller robot does not have the same capabilities as its larger counterpart, the experimentation was performed using only a subset of the EM system functionalities, related to mobility and navigation. This robot is able to move around the environment through manual input or through automatic control, with the latter being achieved by following a specific line on the floor.

Fig. 6.14 shows a picture of the robot, highlighting its main components. A webcam mounted towards the floor provides pictures that are used for line detection (Fig. 6.14-A). This webcam is connected to an ordinary computer, which performs the image processing required for line detection. Based on the results of image processing, the computer will select one of the following instructions: turn left, turn right, move forward, or stop. These instructions are sent from the computer to the robot through a USB port (Fig. 6.14-B). This port is attached to a microcontroller board (Fig. 6.14-C), model Arduino Uno R2[7], which translates instructions

[7]http://arduino.cc/en/main/arduinoBoardUno

Figure 6.12: Base statechart of the EM system

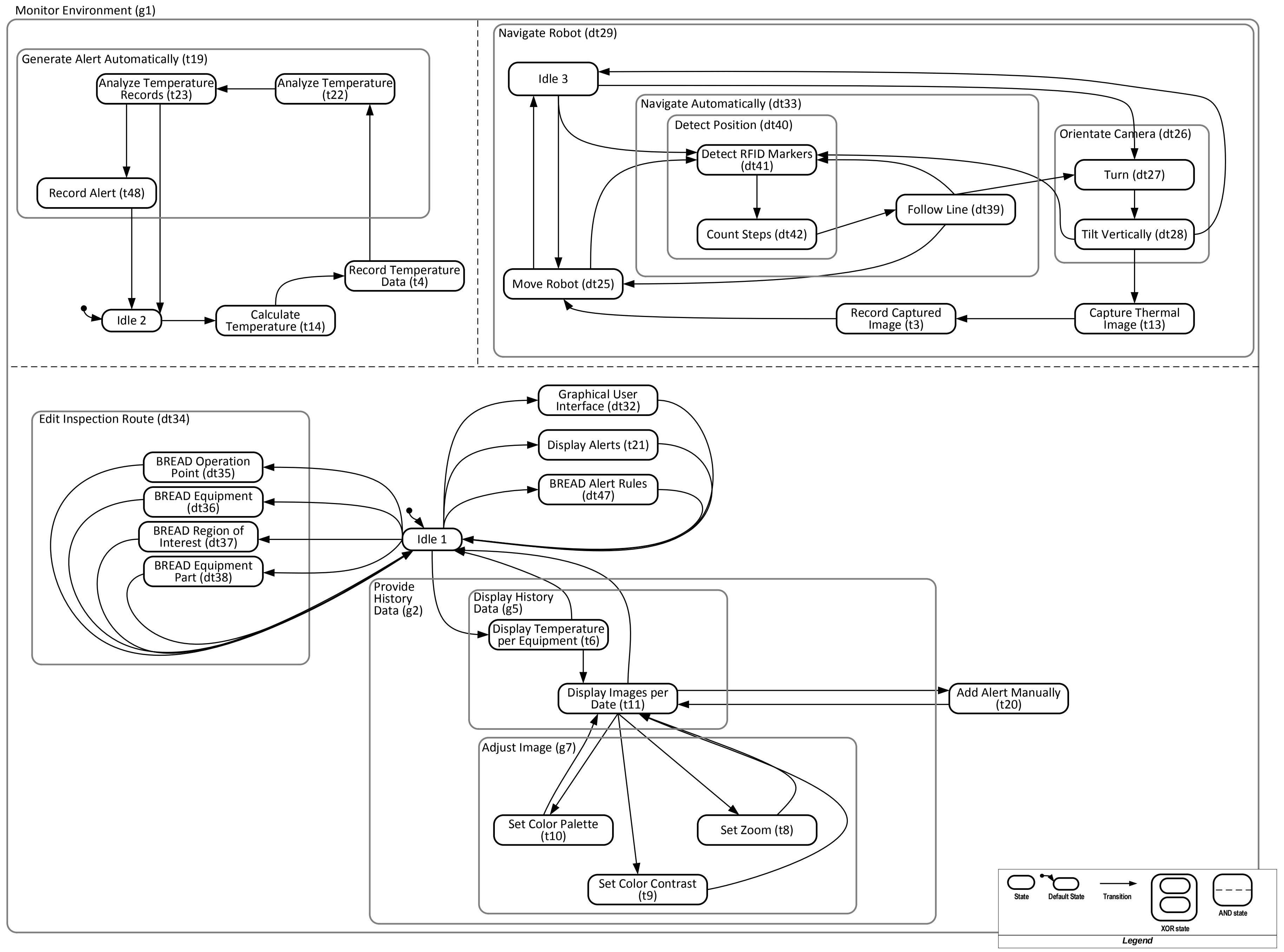

Monitor Environment (g1)
Generate Alert Automatically (t19)
Analyze Temperature Records (t23)
Analyze Temperature (t22)
Record Alert (t48)
Record Temperature Data (t4)
Idle 2
Calculate Temperature (t14)
Navigate Robot (dt29)
Idle 3
Navigate Automatically (dt33)
Detect Position (dt40)
Detect RFID Markers (dt41)
Count Steps (dt42)
Follow Line (dt39)
Orientate Camera (dt26)
Turn (dt27)
Tilt Vertically (dt28)
Move Robot (dt25)
Record Captured Image (t3)
Capture Thermal Image (t13)
Edit Inspection Route (dt34)
BREAD Operation Point (dt35)
BREAD Equipment (dt36)
BREAD Region of Interest (dt37)
BREAD Equipment Part (dt38)
Idle 1
Graphical User Interface (dt32)
Display Alerts (t21)
BREAD Alert Rules (dt47)
Provide History Data (g2)
Display History Data (g5)
Display Temperature per Equipment (t6)
Display Images per Date (t11)
Add Alert Manually (t20)
Adjust Image (g7)
Set Color Palette (t10)
Set Zoom (t8)
Set Color Contrast (t9)
State
Default State
Transition
XOR state
AND state
Legend

Figure 6.13: Awareness requirements and parameters on the EHM system's design goal model

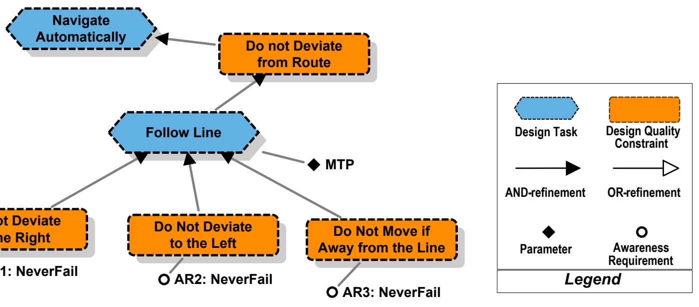
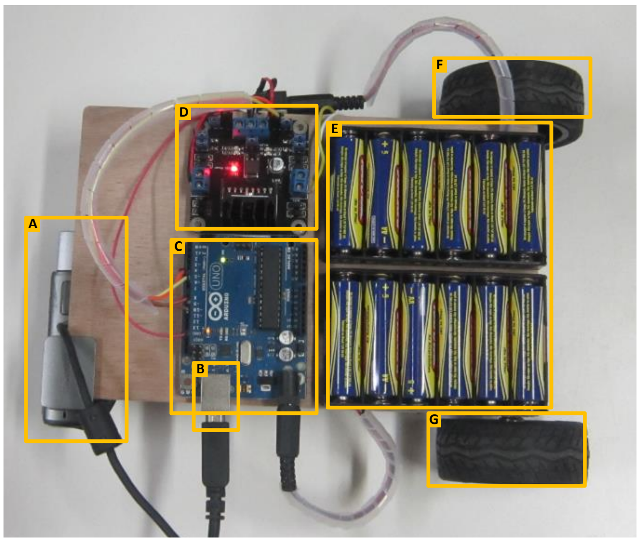


Figure 6.14: Picture of the robot equipment. A) Webcam; B) USB port; C) Microcontroller board; D) Motor driver; E) Power supply; F) Right wheel; G) Left wheel



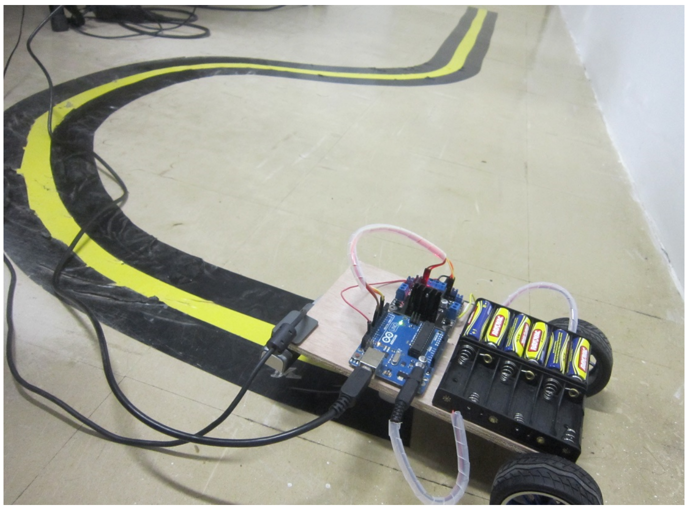
received from the USB port onto I/O signals that control the motor driver (Fig. 6.14-D). Based on these signals, the motor driver provides electricity to the motors that turn the wheels (Fig. 6.14-F and Fig. 6.14-G). The power supply consists of six 1.5V batteries for the motor driver, as well as six 1.5V batteries for the microcontroller board (Fig. 6.14-E).

Navigation of the robot is achieved by marking a line on the floor, over which the robot must move. Fig. 6.15 shows the robot positioned over the line that will be followed, which has a light part on the middle and a dark part on the sides. Based on pictures captured from a webcam mounted on the robot, it is possible to apply traditional image processing techniques and identify the center of the line. Comparing the center of the line with the center of the image, it is possible to know if the robot needs to move either to the left or to the right. In order to perform the required processing we developed a Java program using the OpenCV[8] library, which includes an implementation of different image processing algorithms.

Fig. 6.16 shows pictures of the line shown on Fig. 6.15, taken at different processing

[8]Open Source Computer Vision library, available at http://opencv.org/

Figure 6.15: Picture of the robot positioned over a line to be followed



steps. Fig. 6.16-A shows the image captured through webcam, after being transformed to grayscale. The dark areas on the sides of this image represent the dark tape that is used to delimit the line, whereas the light area on the center of the image represents the light tape on the middle of the path. The white blob on the top of the image is a reflex of light, whereas the somewhat darker area to the bottom of the blob is due to shadow cast by the webcam.

After thresholding, the result is a pure black-and-white image, shown in Fig. 6.16-B. Erosion and dilation are then applied to reduce noise (Fig. 6.16-C). With this image, it is possible to identify the area of the line by identifying the larger white area on the image, using algorithms from SUZUKI; ABE (1985). The extremities of this area are delimited by thin straight lines on Fig. 6.16-D. Based on this area, it is possible to find the horizontal center of the line, depicted as a small black circle on the top of Fig. 6.16-D. On this case, the perceived center of the line is close to the center of the image, meaning that the robot is correctly positioned over the line.

In contrast, Fig. 6.17 shows the processed images when the robot is over a curve. As it moves forward on a curve to the right, the center of the white area (representing the middle of the line) moves to the right. This indicates that the robot should turn right in order to keep following the line.

Besides the programming specific to this domain, it was also necessary to perform the integration with Zanshin, following the procedure described on Section 6.1.1.3 (Step 3): create a metamodel based on the target system goal model; create an editor tool based on that metamodel; instantiate the metamodel, using the editor tool that was created; and customize the Java communication code available in Zanshin's repository. With this setup the robot was able to successfully follow the path depicted on Fig. 6.15. The distance from the center of the robot to the perceived center of the line, throughout a traversal of the path, is shown on Fig. 6.18.

Fig. 6.18 shows that the robot was able to stay reasonably close to the center of the line, only twice going slightly beyond one centimeter. Positive values indicate that the robot

Figure 6.16: Image captured via a webcam mounted on the robot, when over a straight line, after different processing steps. a) captured image, in grayscale; b) image after thresholding; c) image after erosion and dilation; d) image with rectangular contour and horizontal center

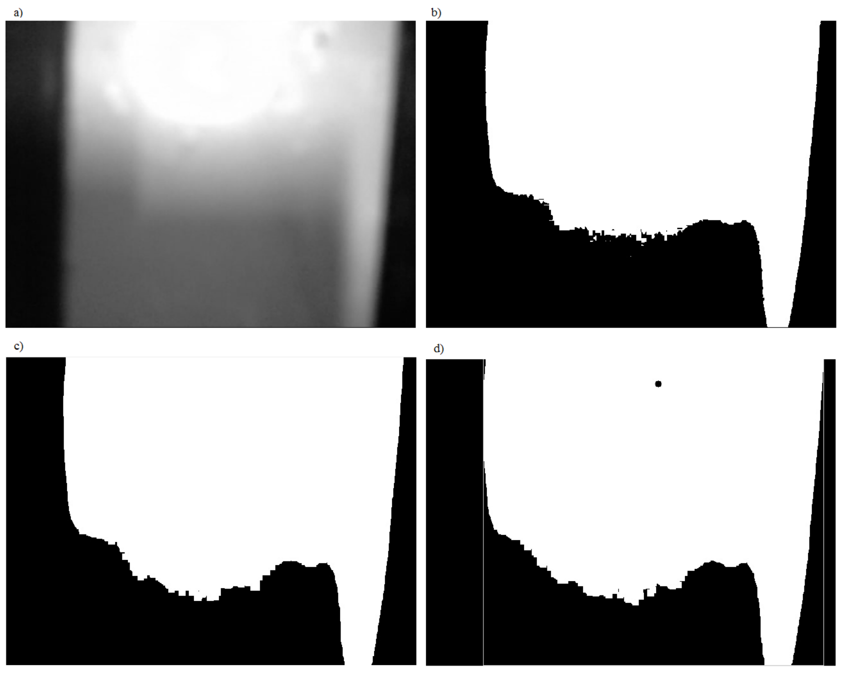


Figure 6.17: Image captured via a webcam mounted on the robot, when over a curve, after different processing steps. a) captured image, in grayscale; b) image after thresholding; c) image after erosion and dilation; d) image with rectangular contour and horizontal center

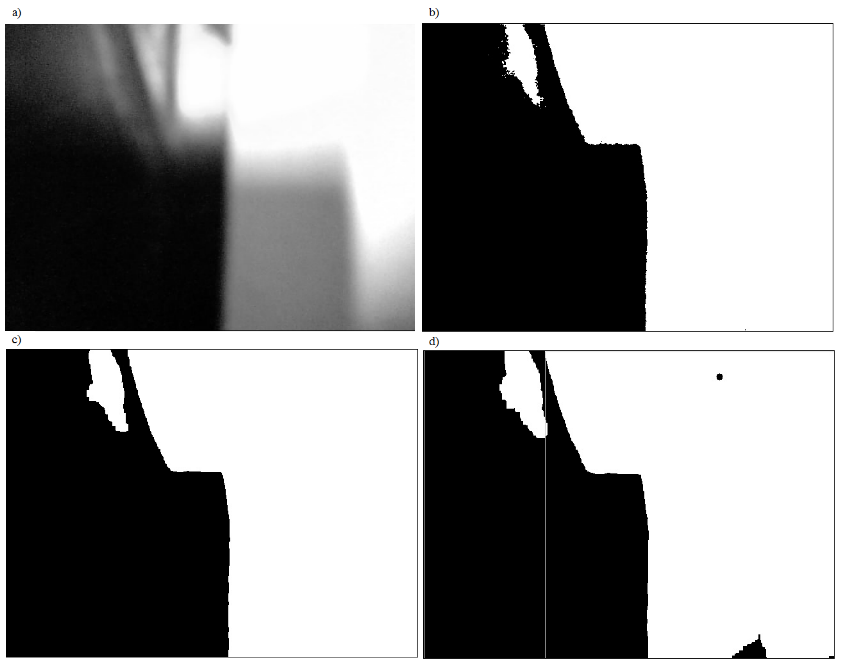
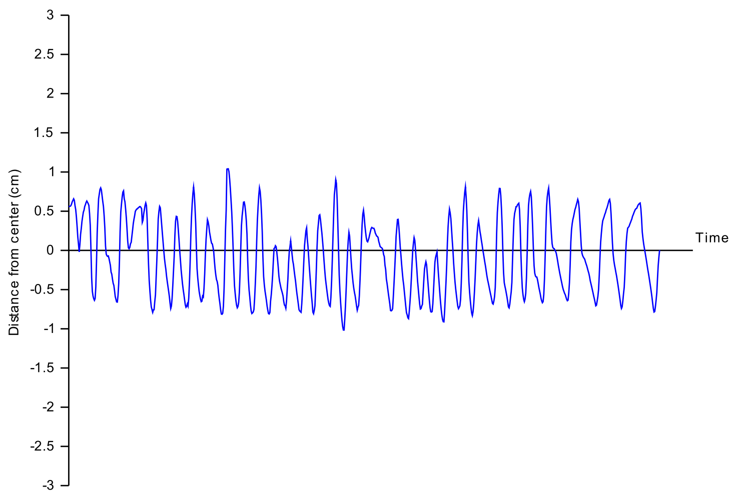


Figure 6.18: Distance from the center of the camera to the center of the visible part of the line being followed, over time



is to the right of the center of the line, whereas negative values indicate that it is to the left. The average distance from center on this run was -0.100 centimeters, with standard deviation of 0.481. However, it is important to notice that the distance plotted on Fig. 6.18 is based on the *perceived* center of the line — the distance from the perceived center to the actual center may vary from zero to ± 1 centimeter. For comparison, the width of the line varies between 12 and 13 centimeters.

6.3 Summary

This chapter presented our initial efforts to evaluate the MULAS framework. First, a concept proof on the development of an ATM system, which shows the evolution of its initial requirements goal model towards a Design Goal Model and the statechart generated from the latter, using the GATO tool that was described in the previous chapter. For the sake of clarity, the ATM concept proof was presented as if the architectural design process was completely followed sequentially. However, the enactment of the process was actually incremental.

The resulting models from the design process are then used to simulate the system behavior with a statechart simulation tool, which showed the correct execution of the adaptation scenario described in the concept proof. As the simulation on Section. 6.1.1.4 (Step 4) illustrates, the use of statecharts in combination with Zanshin's adaptation component helped defining the behavior of an adaptive system, bridging the gap between the system requirements and the system reification in terms of statecharts.

The second concept proof goes a step further and use the MULAS framework and Zanshin's component to actually develop and run a robotic system. This concept proof was based on a real project resulting from an R&D project involving academic and industrial partners. On the experimentation section of this concept proof it was shown that the developed system was able to successfully enact the adaptation segment of the Environment Monitoring system.

While the evidence presented on this chapter is anecdotal, the following chapter presents a set of empirical experiments that evaluate different aspects of the MULAS framework: the scalability of the algorithms for automatically generating statecharts from a design goal model (from Section 5.2), and the design process that is described on Chapter 4.

7
Experiments

In order to evaluate different characteristics of the proposed framework, a mix of empirical methods was utilized. Two controlled experiments were conducted: the first one had the goal of evaluating the scalability of the automatic generation of statecharts from Design Goal Models; the second one aimed at analyzing the use of the architectural design process by non-experts.

7.1 Scalability Evaluation

Besides analyzing if the statechart generated with the MULAS Architectural Design process is able to enact the correct adaptation strategies when coupled with Zanshin's adaptation component, a controlled experiment was performed to assess the scalability of the statechart generation algorithm. The goal of this empirical study is to analyze whether the algorithm is able to handle the generation of statecharts for large systems within a reasonable time.

If the generation takes too long (in the range of hours), the architectural design process would be deemed practically infeasible, since it would not allow for the incremental generation of statecharts. Ideally, the algorithm duration should be within the range of seconds, allowing to provide a quick feedback for the software architect that is designing the flow of the system.

The input for this experiment is a set of ten flow expressions corresponding to models with different number of elements (100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000). This evaluation could have included larger models, but this number is sufficient to provide a notion of the performance of the algorithm. Furthermore, when analyzing goal models used in industrial projects (MAIDEN et al., 2011), we could not find any model with more than 1000 elements[1].

A first flow expression, with 100 elements (see Listing 7.1), was randomly generated and then composed to create the larger expressions. These expressions use all possible opera-

[1]The largest model in this thesis has 102 elements. The largest model presented in a collection of goal modeling showcases (MAIDEN et al., 2011) has approximately 493 elements. The larger example in a recent study that analyzed the complexity of goal models contains approximately 200 elements (GRALHA; GOULÃO; ARAúJO, 2014).

tors: sequential, alternative, concurrent, optional, zero or more times, and one or more times. The derivation of each expression was executed 1000 times, cycling between the expressions, in order to reduce interference from the operating system.

Listing 7.1: Randomly generated flow expression with 100 elements

```
t1* t2? t3 (t4|t5|t6|t7) t8 (t9|t10)? t11+ t12 (t13-(t14 t15 (t16|t17)))
    t18? (t19+)? i20 t21? t22+ t23 (t24|t25|t26|t27) t28 (t29|t30)? t31* t32
     (t33-(t34 t35 (t36|t37))) t38? t39+ i40 t41* t42? t43 (t44|t45|t46|t47)
     t48 (t49|t50)? t51+ t52 (t53-(t54 t55 (t56|t57))) t58? (t59+)? i60 t61?
     t62+ t63 (t64|t65|t66|t67) t68 (t69|t70)? t71* t72 (t73-(t74 t75 (t76|
    t77))) t78? t79+ i80 t81* t82? t83 (t84|t85|t86|t87) t88 (t89|t90)? t91+
     t92 (t93-(t94 t95 (t96|t97))) t98? t99* i100
```

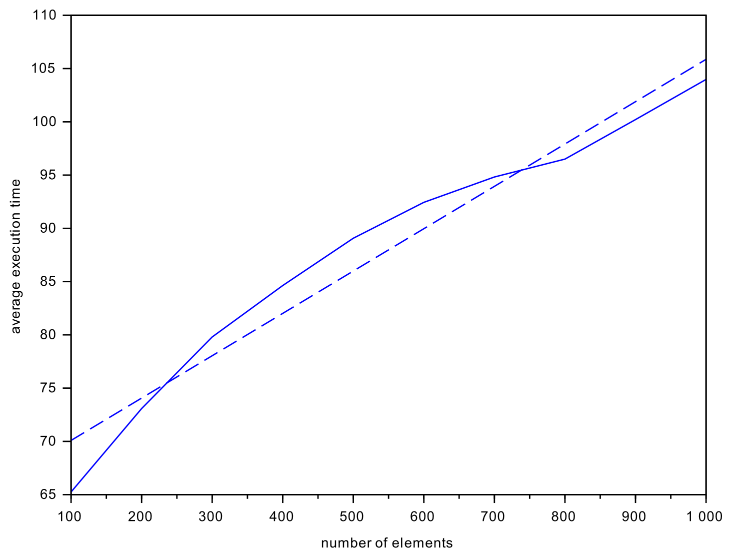
Table 7.1: Average time for deriving statecharts from flow expressions, with standard deviation σ

Size	Average time (ms)	σ
100	65.25	6.32
200	73.08	8.28
300	79.80	6.15
400	84.63	9.59
500	89.07	9.71
600	92.44	12.60
700	94.82	6.29
800	94,68	5.35
900	100.21	8.59
1000	103.98	8.63

Table 7.1 presents the results of the experiment, with average execution times and its standard deviation. The measurement was conducted on a computer with a 64 bits Pentium Dual CPU T4300 processor with 2.1 GHz and 3 Gb of memory. These results show that the derivation of statecharts from design goal models can be performed quickly even on large models, thus indicating that the derivation algorithm is scalable.

These results shown on Table 7.1 do not include an execution of the derivation from flow expression with 800 elements, which lasted for 1919 milliseconds. That execution was clearly an outlier, 1989% higher than the average, which was bringing the standard deviation from 5.35 to 57.94.

Figure 7.1: Average Execution Time of the statechart derivation algorithm. The solid line represents the actual average execution time, while the dashed line represents its linear regression



The solid line on Fig. 7.1 presents an alternative representation of the average execution time per number of elements, whereas the dashed line represents its linear regression. This graph suggests that the execution time of the statechart derivation algorithm is linear.

Two threats to the validity of this study were identified:

- **Unrealistic input —** it is not known if the flow expressions adopted in this experiment are representative of the flow expressions to be found in real, large systems.
- **Possible lack of complexity —** Although the flow expressions used for this experiment contain all the possible operators, they do not contain all of their possible combinations. Some combinations may prove to be more complex than the ones found in this study, which may result on longer execution times.

Moreover, future development on the *Goal to Arch* tool includes the capability to automatically generate a visual representation of the statechart, instead of providing the current text-based list of states and transitions. Once this feature is developed it will be necessary to rerun this experiment, since the generation of visual diagrams will, most likely, require longer execution times.

7.2 Process assessment

When conducting research on information systems it is essential not only to develop theories and artifacts, but also to evaluate them (HEVNER; MARCH; PARK, 2004). Besides the assessment itself, the evaluation may provide additional information which can be used to further refine and improve the proposed solution. Hence, as part of this research, we have used a mix of evaluation methods.

In the previous section we have shown the evaluation of the suitability of the proposed framework for designing adaptive systems, showing that the resulting artifacts were able to reify the adaptive feedback loop.

Considering the effort required to perform the generation of base statecharts, we developed algorithms to automatically derive a statechart from the design goal model. The scalability of these algorithms was evaluated with positive results, showing that they can be applied to large, realistic systems. However, the previous evaluations were conducted by the authors, thus it was still not clear whether the process could be successfully applied by others – and, if so, how easy would it be, and how good would the resulting artifacts be?

The design and results of a controlled experiment (SJØ BERG et al., 2005) with 15 subjects on which we explore those questions is presented in the following sub-sections

7.2.1 Experiment Definition and Planning

This experiment was designed aiming to analyze the MULAS architectural design process, for the purpose of evaluation and improvement, with respect to its use by non-experts, from the point of view of software engineers, in the context of students applying the process on a toy example. This is a exploratory qualitative study, aimed at identifying early indications on the applicability of the process and on the quality of the resulting artifacts. It can be characterized as a multi-test within object study, as a single object is examined across different subjects (WOHLIN et al., 2012).

This experiment took place on a requirements engineering course within a computer science program, with a sample of 15 undergraduate and graduate (master) students. Thus, its context can classified as a specific context (results cannot be generalized), in an offline environment (i.e. it is not a part of industrial software development), with students working on a toy problem.

7.2.1.1 Hypotheses, Variables, and Measures

Albeit being a qualitative study, the definition of hypotheses, variables and measurements is helpful for guiding the planning of the experiment, as well as the results analysis. Thus, the following null hypotheses were defined:

- **$H_0$1:** The syntactical correctness of a statechart created with the MULAS treatment is the same as with the control treatment.
- **$H_0$2:** The complexity of a statechart created with the MULAS treatment is the same as with the control treatment.

- **$H_0$3:** It is not viable for non-experts to enact the MULAS architectural design process.

- **$H_0$4:** The MULAS architectural design process is complex.

If the null hypotheses can be rejected with relatively high confidence, it is then possible to consider alternative hypotheses:

- **H_a1:** The syntactical correctness of a statechart created with the MULAS treatment is higher than with the control treatment.

- **H_a2:** The complexity of a statechart created with the MULAS treatment is smaller than with the control treatment.

- **H_a3:** It is viable for non-experts to enact the MULAS architectural design process.

- **H_a4:** The MULAS architectural design process is not complex.

The independent variable of this study is the statechart generation process, which can assume one of the values in {*MULAS, ad hoc*} —- *MULAS* is the creation of statecharts using the MULAS architectural design process, whereas *ad hoc* is the creation of statecharts without any kind of guidance (control group). Each of these options correspond to a experimental treatment.

The dependent variables are: syntactical correctness of statecharts, complexity of statecharts, process viability, and process complexity. The metrics adopted to measure these variable are described next.

The syntactical correctness of statecharts can be measured by the number of syntactical errors, where more errors implies less correctness. To evaluate the complexity of statecharts, we adopted the metrics proposed and validated by GENERO; MIRANDA; PIATTINI (2002); MIRANDA; GENERO; PIATTINI (2003): *number of states* and *number of transitions*.

While the first two hypotheses can be tested by comparing the results of both treatments, the remaining two hypotheses are tested by analyzing only the results of the MULAS treatment. To assess the viability of non-experts enacting the MULAS architectural design process, we analyzed the correct use of flow expressions (based on *number of errors, coherence with the statechart*, and *coherence with the expected behavior*), as well as the correct use of adaptation strategies patterns (*number of errors*). Lastly, the perceived complexity of the process can be analyzed by means of Likert items (Appendix C.4).

The controlled variables are *time spent studying MULAS* and *time spent to perform the experiment*. The first variable was controlled by assuring that all subjects in the MULAS treatment group had the same amount of classes on the topic. Moreover, no resource on the topic was made available for them, preventing studies away from class. To prevent this lack of individual studies from harming the quality of the results, a reference guide was provided to each subject during the experiment itself. This reference guide summarized the steps, notation, and patterns from the architectural design process. The second variable was controlled by requiring all the students to remain in the experiment room for a set amount of time —- no subject was allowed to leave early or to stay late.

7.2.1.2 Subjects, Treatments, and Instrumentation

The sample of this study is a set of students from a requirements engineering course offered at the Center of Informatics in the Federal University of Pernambuco, during the second semester of 2014. Thus, this sampling can be classified as a non-probability, convenience sampling (WOHLIN et al., 2012).

In this experiment we compare (i) the use of the MULAS architectural design process for creating statecharts with (ii) the creation of statecharts without the use of any particular method or process (ad hoc). Subjects on the first treatment compose the MULAS group, whereas subjects on the second treatment compose the control group.

Subjects were assigned to the different treatments randomly. Since the same object (an e-commerce system) was used in both treatments, this can be classified as a completely randomized design. Since there was an odd number of participants, the distribution was not evenly: seven subjects were assigned to the MULAS group, while eight subjects were assigned to the control group. Therefore, there is a slight unbalance on the distribution of subjects.

The object of this study is a description of the behavior of an online bookstore system. Considering the specificities of each treatment, two slightly different instruments were created for the same object (Appendices C.2 and C.3). In the first instrument, applied to the MULAS group, adaptation was described using awareness requirements and parameters. However, since participants of the control group were not trained on these concepts, the same adaptation was described textually in their instrument (Appendix C.3).

The measurement instrument devised to collect data about the complexity of the MULAS architectural design process was a questionnaire composed of eighteen Likert items (LIKERT, 1932), presented in Appendix C.4. Since this questionnaire refers to the MULAS architectural design process, it was applied only to participants of the MULAS group. The data regarding other aspects of the experiment was collected manually upon the statecharts created by the subjects.

Besides these measurement instruments, we also designed a characterization questionnaire containing questions about the software engineering background of the subjects (Appendix C.1). This questionnaire was applied to both groups.

Lastly, according to WOHLIN et al. (2012), guidelines such as process descriptions and checklists are part of the instrumentation of a experiment. Thus, subjects in the MULAS group received a description of the MULAS architectural design process to be consulted during the experiment. Since the control group did not adopt any particular process or method, no such instrument was provided to them.

While this section describes the definition and planning of the experiment, the following section presents the preparation of the experiment, as well as the execution of the experiment itself.

7.2.2 Experiment Preparation and Execution

Before performing the experiment we needed to assure that all subjects had enough familiarity with statecharts. This was achieved through a series of activities: first, the students participated on 6 hours of classes on statecharts, including theory, practice, and tool support. Then, they were tasked with an assignment on which they needed to create a statechart based on a textual description of the requirements and of the expected behavior of a software system. In order to assure that each student created their statecharts individually, the students had to perform an oral examination where they described the statechart they created and answered questions related to their model. Since this assignment was part of their grading in the requirements engineering course, the students were motivated to perform well on this assignment. Nevertheless, students with unsatisfactory results in this phase were excluded from the experiment.

With this training and examination, the control group was ready to take part on the experiment, since they had shown that they were able to successfully perform the task of the experiment: to create a statechart based on a textual description of the behavior of a system, without adopting any particular method or process. For the students on the MULAS group we provided an additional 5.5 hour training on the MULAS framework and its architectural design process, consisting of theoretical lessons and practical activities.

The training and examination performed during the preparation of this experiment required a total of 16.5 hours of classroom interaction with subjects. On top of this time, a reasonable effort was required in order to plan the experiment, as well as to prepare the training resources and the experiment instruments. This preparation took place over the span of seven weeks.

The experiment execution was carried out on a 2-hour session, corresponding to the duration of a regular class. In the first 10 minutes the students answered a characterization questionnaire, used to obtain their background on software modelling, in particular, and software engineering, in general. The following 1 hour and 40 minutes were spent with the execution of a set of tasks, resulting on the creation of a statechart based on the goal model of a system and on a textual description of its behavior. The remaining 10 minutes was used for a feedback questionnaire composed of 18 Likert items, answered only by those subjects on the MULAS treatment. In order to reduce bias, this questionnaire was applied anonymously.

During the experiment, the following protocol was observed: only questions about the instruments could be answered by the experimenter; no question about the MULAS architectural design process or about statecharts could be answered; no interaction between subjects was allowed.

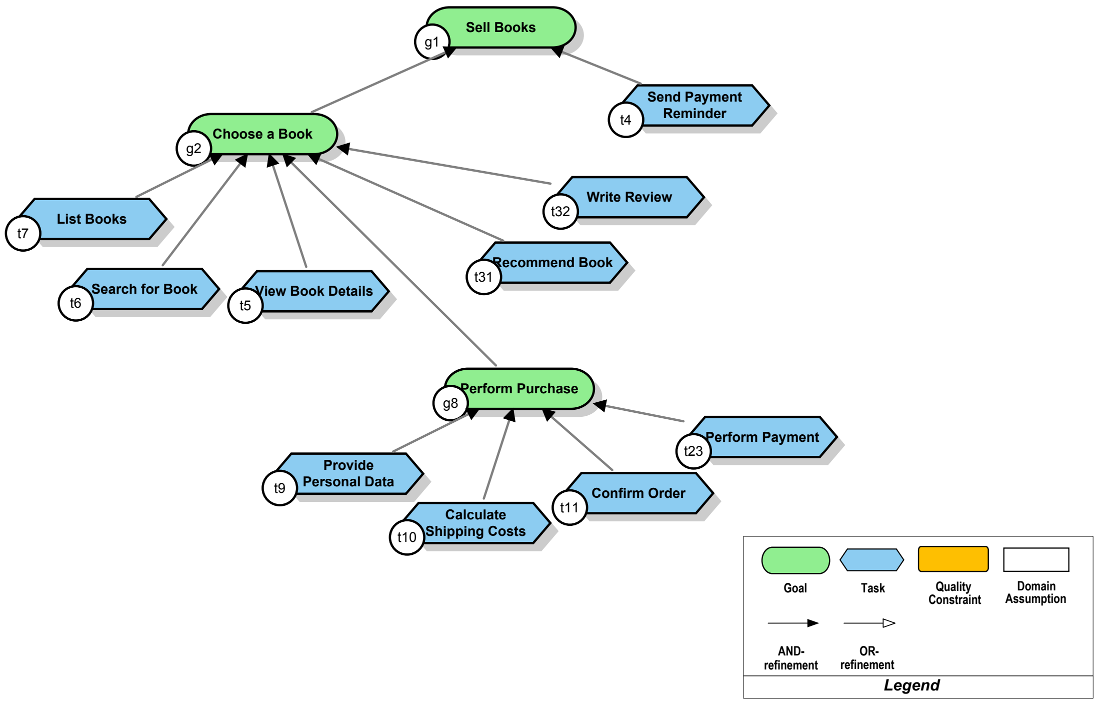
The set of tasks required from the subjects was slightly different between treatments, but they both had the same experimental object: an online bookstore. The subjects were provided with the system's requirements (expressed as a goal model), and a textual description of the expected behavior of the system. Fig. 7.2 shows the goal model, whereas the textual description of its behavior is as follows:

> *The Saravá Bookstore intends to develop a web-based e-commerce system, where it can sell the books in its inventory. Accessing the website, users will see a listing of available books. If the user finds the desired book in the listing, she will be able to view the book details directly from the listing. If the book was not found on the listing, the user can use a search mechanism, as many times as warranted, until the book is found —- then, the book details can be viewed.*
>
> *Once the desired book is found and its details are viewed, the user has 3 options: either buy the book, or send a book recommendation, or write a book review.*
>
> *In order to perform the purchase, the user must follow the following procedure: first it is required to provide personal data, then calculate shipping costs, confirm order, and lastly perform the payment. However, the shipping costs calculation is not always performed, as the user may choose to have the book delivered on a physical store. To make a purchase, no kind of registering or login is required.*
>
> *Regardless of user interaction, the system shall send e-mails at 12-hours intervals for those users whose payment have not been approved yet.*

Even though it can be considered a simple behavior, it involves non-trivial flow expressions, comprising the different constructs from the flow expression language: sequences, alternatives, optionality, repetition, and parallelism. The following flow expressions represent a possible solution: `g2 - ((i1 t4)+)` for g1, `t7 t6* t5 (g8 | t31 | t32)` for g2, and `t9 t10? t11 t23` for g8.

The tasks performed by the subjects can be divided in two sub-sets. The first sub-set was concerned with the regular behavior of the system (without adaptation), whereas the second sub-set was concerned with the adaptation behavior. The MULAS group had to create a statechart following the following steps of the process: *Define basic flows*, *Generate base statechart*, *Specify transitions*, and *Include adaptation elements*. For this group, the adaptation behavior was defined in terms of awareness requirements and adaptation strategies. The full description of the tasks are presented on Appendix C.2. Additionally, a reference guide with main elements of the MULAS framework was made available for the students during the experiment. For the control group, the adaptation behavior was described textually, since they had not been introduced to these concepts. Still, they had to perform two tasks: create a statechart that represents

Figure 7.2: Goal Model used in the empirical evaluation



the regular behavior, and modify the created statechart in order to include the adaptation behavior (C.3). No supporting tool was allowed for the experiment, thus the outcomes of the tasks were produced manually over pen and paper.

After the execution of the experiment, the collected data was validated, checking against unanswered questions, incomplete answers, questions with more than one answer, and unreadable text. No problem was found on the collected data.

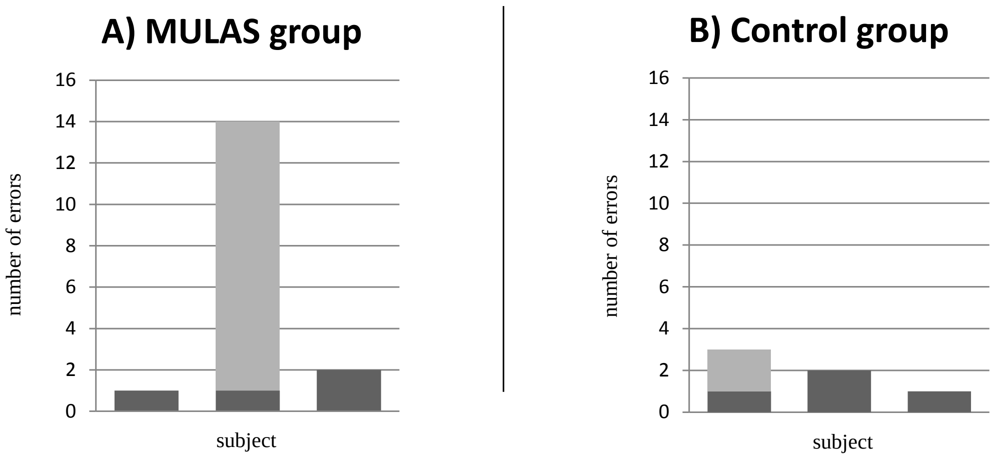
With this structured activity, where subjects on both treatments tackled the same object, we were able to compare statecharts created with and without the MULAS architectural design process, as described in the following section.

7.2.3 Results and Analysis

The first hypothesis (**H**$_0$**1**) is related to syntactical correctness of the statecharts created by the experiment participants. In order to measure correctness, we can analyze the number of errors in the statechart. Fig. 7.3 provides an overview on the number of errors observed in the statecharts created during the experiment. Different bars represent different subjects, whereas different shades on the same bar represent different kinds of errors made by the same subject.

On each group, three subjects made syntactical errors, suggesting a balance on the syntactical correctness of both treatments. When considering the number of rules violated, the results are also identical: on the MULAS group, two of the subjects violated one syntactical rule each, whereas one subject violated two syntactical rules, totalling up four kinds of mistakes; the same is the case on the control group. However, when considering the number of

Figure 7.3: Syntactical errors in the statecharts created during the execution of the experiment. Different shades in the same bar indicate different kinds of errors. A) MULAS group; B) Control group



instances of each error, there are more syntactical errors on the MULAS group (17) than in the control group (6). This disparity of 183.33% is due to a single kind of mistake repeated thirteen times by a single subject: the lack of events on statechart transitions. While the sample of this experiment is not statistically relevant, we are assuming that this subject (represented by the bar in the middle of the MULAS group, Fig. 7.3) is an outlier, as it made 3266,67% more mistakes than the average on the same group, and 700% than the second subject with most mistakes on the same group.

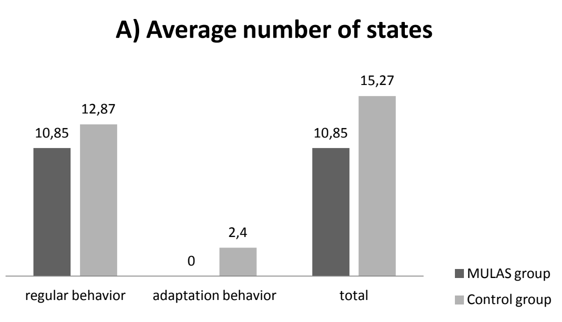
Hypothesis $\mathbf{H_0 2}$ is concerned with the complexity of the statecharts. Since all subjects covered the entire behavior requested in the textual description of the system, it was possible to directly compare the statecharts from the control group and from the MULAS group. Three metrics for measuring the structural complexity of statecharts were validated by MIRANDA; GENERO; PIATTINI (2003): Number of Activities, Number of States and Number of Transitions. Since no statechart activity[2] was defined in this experiment, we adopted only the latter two metrics.

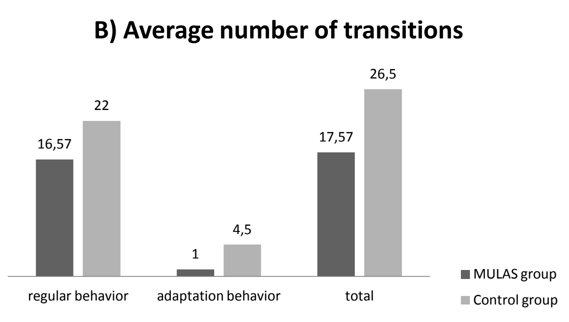
Comparing the number of states in the regular behavior, there is a slight difference between the MULAS treatment (average of 10.85 states), 15.69% less than the ad-hoc treatment (average of 12.87 states), as shown in Fig. 7.4-A. Moreover, the number of transitions in the MULAS treatment was considerably smaller than in the ad-hoc treatment: an average of 16.57 transitions against an average of 22 transitions, a difference of 24.68% (Fig. 7.4-B).

The difference is even more relevant when considering the adaptation behavior. In the MULAS treatment, considering each subject, no additional state was included in the statechart, while only one additional transition was included – in conformity with the adaptation strategies patterns. In the control group, an average of 2.4 additional states and 4.5 additional transitions

[2]Statechart activities are long-term actions, which represent continuous and interruptible work that is carried out *while* the system is in a given state (HAREL, 1987).

Figure 7.4: Number of elements in the statecharts created during the execution of the experiment. A) Average number of states; B) Average number of transitions





were included in the model.

In order to analyze the viability of the enactment of the MULAS architectural design process by non-experts (**H$_0$3**), we can consider the number of errors specifically related to the process. In order to do so, we analyzed the flow expressions created by subjects in the MULAS group with respect to *syntactical errors, coherence with its respective statechart* and *coherence with the textual description of the expected behavior*. Moreover, we also measured the number of errors related to the application of adaptation strategies patterns.

An overview of the number of errors related to flow expressions is shown on Table 7.2. None of the subjects made syntactical errors —- i.e., all of their expressions were valid expressions. Two out of the seven subjects had incoherencies between their flow expressions and their statechart —- one made one mistake, and the other one made two mistakes. In both cases, the statechart represented the expected behavior of the system, but were in discordance with the flow expressions. Two other subjects wrote correct expressions coherent with the textual description, but in the statechart included additional transitions —- one included one transition, and the other included two transitions. Thus, these expressions were also incoherent with their respective statecharts. Inversely, another subject wrote flow expressions coherent with his statechart, but incoherent with the textual description of the system behavior. Considering these different kinds of mistakes, only two subjects wrote flow expressions correct on all aspects.

Besides analyzing the use of flow expressions, we also analyzed the use of adaptation strategies patterns (Table 7.3). The experiment instrument required the use of two adaptation strategies patterns: *notify* and *retry* (Appendix C.2). All subjects successfully reified the *notify* pattern. Only two subjects made a mistake on the *retry* pattern, as follows: the additional transition had the wrong source state.

Lastly, hypothesis **H$_0$4** is concerned with the complexity of the MULAS architectural design process. Through a post-experiment questionnaire containing 18 questions (Appendix C.4) it was possible to assess the complexity of the process, as perceived by the experiment subjects. These questions are formulated as Likert items (LIKERT, 1932), i.e., they are statements to which the respondent may strongly disagree, disagree, be neutral, agree, or strongly agree.

Table 7.2: Number of errors related to flow expressions. Each row represents a subject of the MULAS group

Syntactical errors	**Incoherence between flow expression and statechart**	**Incoherence between flow expression and textual description of the system behavior**
0	1	0
0	2	0
0	1	0
0	2	0
0	0	1
0	0	0
0	0	0

Table 7.3: Number of errors related to adaptation strategy patterns. Each row represents a subject of the MULAS group

Adaptation strategy pattern	
Notify	**Retry**
0	1
0	1
0	0
0	0
0	0
0	0
0	0

Figure 7.5: Answers of the post-experiment questionnaire — questions A to H

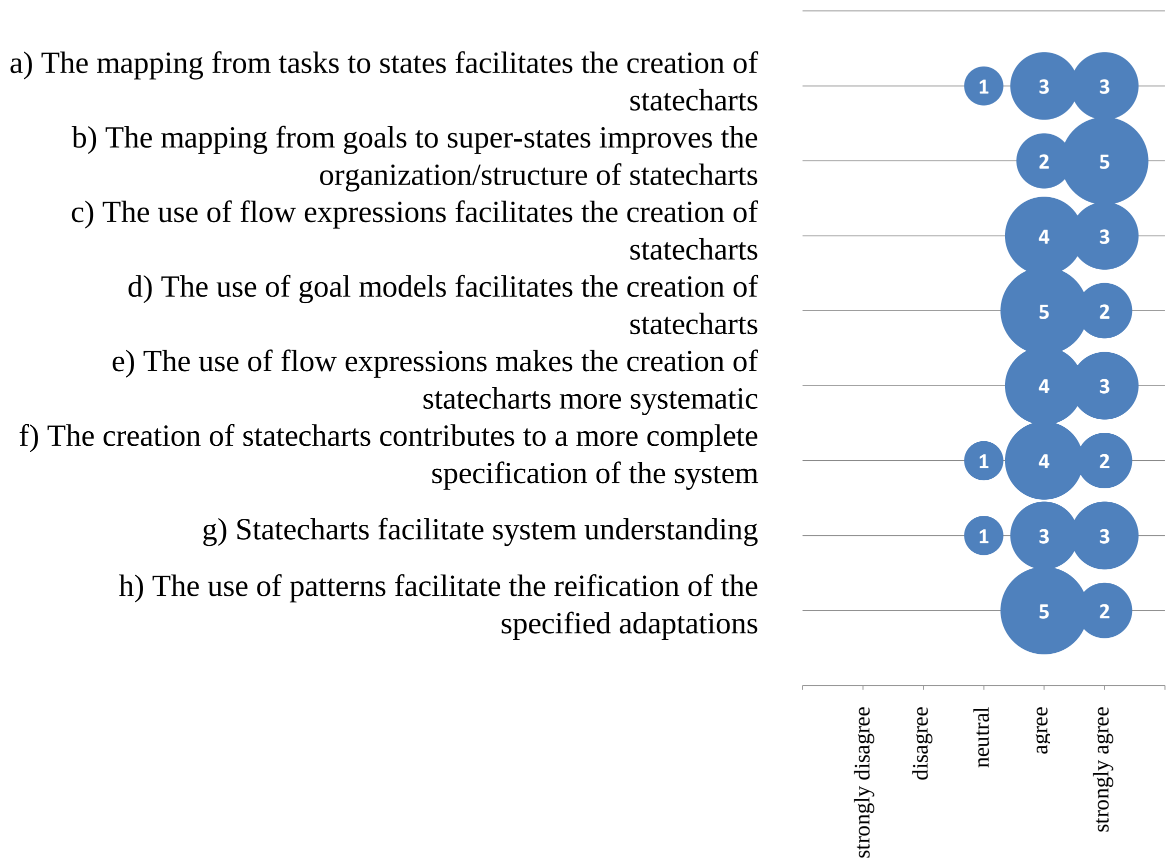


Fig. 7.5 shows the questions and answers of the first eight questions of the questionnaire. From these answers, 94.64% are positive, while the remaining 5.36% are neutral. In particular, questions *a*, *c*, *d*, and *h* ask whether different parts of the process facilitate the creation of statecharts.

Follow-up items ask about the ease of use of each kind of flow expression construct: sequence, alternative, optionality, repetition, parallelism, and idle states. The answer to those questions were mostly positive, with some disagreements (as shown in Fig. 7.6). Contrary to our intuition, the repetition and parallelism cases had the most positive results, without any disagreement. This was unexpected since, based on our own experience, those two constructs are the most complex ones.

The remaining questions are related to four adaptation strategy patterns that were covered in the pre-experiment training: *abort*, *notify*, *reconfigure*, and *retry*. The answers, while mostly positive, contained some disagreements: one for the abort pattern, two for the reconfigure pattern, and one for the retry pattern (Fig. 7.7). Comparing the four patterns, the *reconfigure* one seems to be the most problematic, as it has the highest number of disagreements (two) and the lowest number of strong agreements (zero).

As described previously in this chapter, this experiment is a qualitative study, rather than

Figure 7.6: Ease of use for different constructs, according to the subjective questionnaire

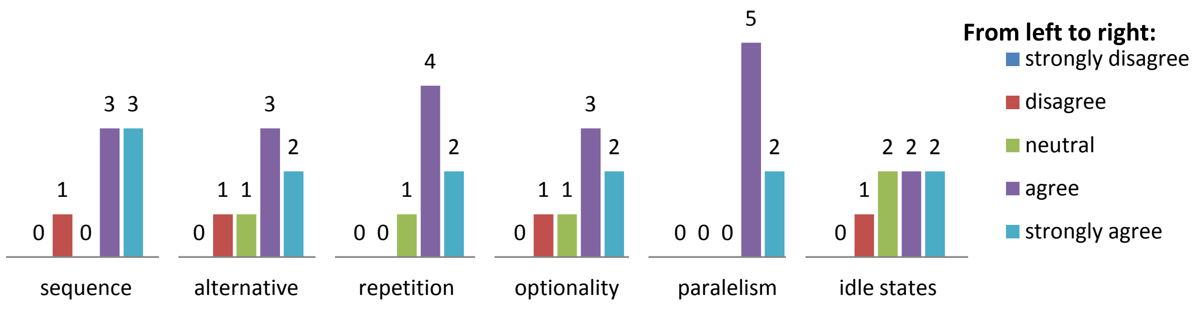
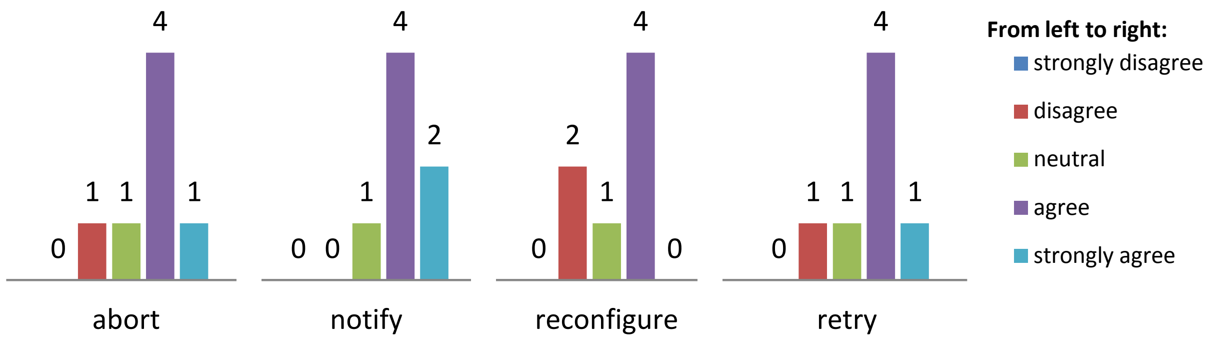


Figure 7.7: Ease of use for different adaptation strategies patterns, according to the subjective questionnaire



quantitative. Thus, its results cannot be generalized to other context. The goal of this study is to provide early evaluation of the approach, besides providing inputs for further improvements. Even though it is not possible to conclusively reject or confirm the hypotheses of this experiment, due to a lack of statistical significance, it is possible to ponder the implications of the results in this specific context:

- **$H_0$1: The syntactical correctness of a statechart created with the MULAS treatment is the same as with the control treatment —** this hypothesis cannot be rejected since the statecharts created by the MULAS group presented, in total, more errors than those created by the control group.

- **$H_0$2: The complexity of a statechart created with the MULAS treatment is the same as with the control treatment —** this hypothesis can be rejected, since the statecharts created by the MULAS group presented less complexity than those created by the control group, as measured by the number of states and the number of transitions. Thus, in this specific context, the alternative hypothesis **H_a2** must be true: the complexity of a statechart created with the MULAS treatment is smaller than with the control treatment.

- **$H_0$3: It is not viable for non-experts to enact the MULAS architectural design process —** this hypothesis can be rejected, since there are subjects that were able to enact the process successfully. Hence, in this specific context, the alternative hypothesis **H_a3** must be true: it is viable for non-experts to enact the MULAS architectural design process, given proper training

- **$H_0$4: The MULAS architectural design process is complex —** based on the results of the post-experiment questionnaire, it is possible to affirm that this hypothesis can be rejected. Therefore, in this specific context, the alternative hypothesis **H_a4** must be true: the MULAS architectural design process is not complex.

The next section presents further considerations regarding the results here presented.

7.2.4 Discussion

During the classes it was observed some difficulty on understanding the difference between the *alternative* and the *optional* constructs. Although the difference was discussed in class, one subject still mistakenly used the *alternative* symbol to represent an *optional* task. In order to prevent this kind of confusion, the supporting resource (process description) can be improved to clarify the meaning of each construct and to explicitly show the difference between them. Another recurrent doubt during classes concerned the difference between the zero-or-more and one-or-more variations of the *repetition* construct. However, this issue seems to have been resolved during classes, since there was no error related to this doubt in the flow expressions created by the subjects during the experiment.

During the experiment, most of the subjects from the MULAS treatment group made at least one mistake related to the flow expressions. However, these mistakes were localized in a specific excerpt of the expressions. Thus, even though there were mistakes, most of each expression was written correctly. Furthermore, the use of a supporting tool can prevent these errors, by enforcing the coherence between flow expressions and their respective statecharts.

In the beginning of this experiment the authors had concerns about the usability of the process, since it had never been applied by non-experts. During the classes and the experiment itself we could observe that the subjects were able to learn how to perform each step of the process — in particular, the creation of flow expressions and the application of the adaptation strategies patterns. The evidence gathered with this experiment, although not conclusive, suggests that the process is indeed usable given proper documentation and training, since: one of the seven MULAS subjects was able to correctly follow the process, and four other subjects concluded the process with only minor mistakes.

Besides the objective comparison of complexity, based on metrics validated by MIRANDA; GENERO; PIATTINI (2003), when analyzing the different statecharts we observed that statecharts created with the MULAS process are more uniform in terms of structure and nomenclature, resulting in similar models. This uniformity indicates that these statecharts seem to be easier to understand, as well as being easier to compare against each other.

With the characterization questionnaire answered during the experiment, we observed a bias in the distribution of subjects between treatments: the MULAS group contained 5 master students and 2 undergraduate students, whereas the control group had 2 master students and 6 undergraduate students. In order to prevent possible biases in future experiments it is advisable to apply the characterization questionnaire beforehand, so that some blocking technique may be adopted if needed be.

In the remainder of this sub-section we discuss other threats that may compromise the validity of this experiment.

- **Conclusion validity —** The conclusion validity is concerned with the factors that could affect the ability to draw conclusions about the experiment. As described previously, this is an initial experiment with a small sample. Thus, the results from this experiment have no statistical relevance and cannot be generalized for the population as a whole. Moreover, since the object of the experiment was based on a toy example, it may not be representative of real adaptive systems. In order to mitigate these threats it would be necessary to perform a larger experiment with more representative samples and object.

- **Internal validity —** The threats to internal validity are influences that can affect the results with respect to causality. Aiming to prevent threats in the History and Maturation categories, we applied both treatments to the same object at the same time. To reduce the impact of Selection within our sample, the assignment of subjects to different groups was randomized. Lastly, reducing the effect of Instrumentation, no supporting tool was used by the subjects during the experiment. Moreover, all measurements were taken with objective metrics, ensuring that the assessment performed by different observers would obtain the same results.

- **External validity —** Threats to external validity are conditions that limit our ability to generalize the experiment to real practice. In that respect, empirical studies with students provide some particular challenges (CARVER et al., 2003). In order to reduce the difference between results obtained with students and results obtained with professionals, CARVER et al. (2003) suggests that the tasks they execute in the experiment must be highly structured. That is the rationale for our decision of providing a complete description of the expected behavior in the object, as well as to provide a clear set of steps for the execution of the experiment (Appendices C.2 and C.3). Another concern when performing experiments with students is that of motivation. In order to provide motivation, the participation on this experiment was used as part of their grade. Nonetheless, this participation was voluntary – students who did not want to partake on it could have an alternative method of evaluation. In the end, two students decided not to participate on this experiment.

- **Construct validity —** Construct validity concerns generalizing the results of the experiment to the concept or theory behind the experiment. When analyzing the results of the feedback questionnaire, it is important to consider hypothesis guessing – i.e., that the subjects may provide positive feedback because they believe they are supposed to. Additionally, object representativeness may compromise the construct validity. In order to mitigate this threat our object contained representations of all classes of constructs, both in the regular behavior (sequences, alternatives, optionality, repetition, and parallelism) and in the adaptation behavior (action-based and transition-based, plus monitoring).

7.3 Summary

This chapter presented a scalability evaluation, showing that the statechart derivation algorithm performs well even when taking large models (1000 elements) as input. This level of performance is important in order to provide quick feedback for tool users, as they will be able to quickly see the resulting statechart after modifying the Design Goal Model.

Moreover, an empirical evaluation with software engineering students showed that it is feasible for non-expert users to enact the proposed architectural design process. The results are not conclusive, but they are promising, as some of the collected metrics are slightly positive when contrasted with an *ad hoc* approach.

The next chapter presents final considerations about this work, including its limitations and future work.

8 Conclusion

In this chapter we present a summary of our contributions and some final considerations, as well as future work to be carried on for the advancement of the MULAS framework.

8.1 Context

There are different approaches and frameworks for supporting the development of adaptive systems, but they are often restricted to a single aspect of software development. For instance, the Zanshin framework (SOUZA et al., 2013) provides support for handling adaptation at the requirements level, enacting the monitoring-diagnosis-compensation cycle. Rainbow (GARLAN et al., 2004) provides similar capabilities, but addressing architectural models – namely, components and connectors. However, with the former the developer is at a loss on how to implement the system, since the starting point is at a too high abstraction level. With the latter there is no relation between the architectural models and the system requirements, which can lead to a mismatch between the stakeholders' expectations and the delivered solution. A multi-level solution that handles these different aspects of software adaptation can improve the development of adaptive software systems by supporting a broader set of adaptations, as well as by bridging the gap between requirements and architectural design from the adaptation point of view.

However, as convincingly argued by BROOKS (1986), there is no single solution that solves all the problems in software engineering. In fact, the support of adaptation may increase the complexity and the size of software systems, thus increasing the effort required to develop adaptive systems. This kind of problem can be mitigated by means of Model Driven Engineering.

Model Driven Engineering is an approach for software development focused on models, rather than on source code. Ideally, a software developer would not need to write a single line of source code — instead, executable code would be automatically generated from the models. In practice, the majority of the current approaches still require manual coding. Nonetheless, such approaches are able to reduce the effort required for software development, providing benefits with regard to productivity, portability, and maintainability (KLEPPE; WARMER; BAST, 2003).

In this thesis we presented the MULAS framework — Multi Level Adaptation for Software Systems. On one hand, it integrates requirements-based and architecture-based adaptation within a single framework. On the other hand, it integrates adaptation frameworks with model driven engineering. In the following subsection the main contributions of this thesis are presented.

8.2 Contributions

In Chapter 3 we presented the results of our investigation on software adaptation, discussing possible adaptations at the requirements level and at the architectural level. Based on this investigation, we proposed the Design Goal Model (DGM), also described in Chapter 3 along with its metamodel and constraints. The DGM allows to express, in a single model, information about requirements, architectural design, and adaptation, hence enabling multi-level adaptation.

Considering the intertwined nature of requirements engineering and architectural design (NUSEIBEH, 2001), we proposed an architectural design process with which a system's architecture can be iteratively and incrementally designed. Throughout this process, presented in Chapter 4, the goal model is enriched with *design tasks*, *design constraints*, *design assumptions*, *assignments*, *behavioral annotations*, *additional awareness requirements* and *additional parameters*. Moreover, *adaptation strategies* are specified and a *statechart* is derived and refined. The resulting statechart is instrumented with the actions required to communicate with an external adaptation component (such as Zanshin's), and is able to enact the adaptation strategies that were specified.

Acknowledging the effort that would be required in order to create and maintain the models used throughout the process, we developed the Goal to Architecture (GATO) tool, a modeling tool that supports the creation of the process artifacts, including the design goal model with its multiple views (Chapter 5). This tool is unique in the context of goal modeling tools in the sense that its execution does not require the installation of any specific piece of software, being accessible through modern web browsers.

Moreover, derivation patterns and algorithms for the generation of statecharts from Design Goal Models are also presented in Chapter 4 and Chapter 5. These algorithms were implemented and integrated with the GATO tool. By means of a controlled experiment, described in Chapter 7, it is possible to observe that these algorithms are able to handle large models with good performance.

8.3 Related Work

In this section we discuss related work that represent the state of the art on the main topics covered within this thesis: software adaptation and the creation of architectural models from requirements.

8.3.1 Software Adaptation

The following subsections provide a brief overview of approaches that target adaptation at the requirements level and at the architectural level, respectively.

8.3.1.1 Requirements-centric

There are several approaches for developing adaptive systems based on goal models. In this subsection we discuss some of them, including those on which our process is based.

LAPOUCHNIAN; MYLOPOULOS (2009) and ALI; DALPIAZ; GIORGINI (2010) use the notion of context to express domain variability. The goal model is annotated with context expressions that define conditions on the model elements. During runtime, a system may check if a task being performed is allowed on that context and, if not, it may change its behavior. Both approaches are concerned with reasoning at requirements level, without prescribing any specific architecture.

DALPIAZ; GIORGINI; MYLOPOULOS (2009) also uses context-enriched goal models, aiming to deploy adaptive systems. Besides constraining the selection of alternatives, the context is used to define activation events and commitment conditions for goals and preconditions to tasks. Compensations are also defined to mitigate the occurrence of failures. Their approach describes the architecture of a component responsible for performing the adaptation-related reasoning. However, it does not prescribe how to define the architecture of the system that will interact with this component.

MORANDINI; PENSERINI; PERINI (2008) propose to use goal models enriched with environment and fault modeling. The goal status is expressed in terms of environment conditions, similar to the context annotations. Fault modeling is used to define situations on which recovery activities may be performed to prevent or mitigate a fault. This is similar to the concept of obstacles that is part of KAOS. Besides being a comprehensive approach, it is only suited to develop multi-agent systems.

At another level of requirements engineering for adaptive systems (BERRY; CHENG; ZHANG, 2005), there are some approaches based on the notion that requirements might change at runtime and that the system should be able to respond to these changes with minimal human intervention. However, as of today there are still too many open issues on these approaches, such as how to express the new requirements in a way that is both simple to the user to define and that can be understood by the machine.

JIAN et al. (2010) allows the insertion of goals at runtime. However, to respond to these changes new modules must be incorporated to the system as well. This approach also uses a notation for expressing environmental conditions similar to the contextual approaches above.

QURESHI et al. (2010) also allows the changing of goal models at runtime: add goal, add means-end, suspend means-end, resume means-end and relax means-end. To address these news goals it uses a service-based architecture, on which a lookup mechanism will identify services that may satisfy the new requirements. The services may either already exist on the system's pool or may be found through web service search mechanisms.

BENCOMO; WHITTLE; SAWYER (2010) also deals with the notion of changing goal models at runtime, through requirements reflection. Additionally, it uses a flexibility language to deal with uncertainty.

BARESI; PASQUALE (2010) propose the use of adaptive goals, in contrast to conventional goals. The adaptive goals specify countermeasures to be performed when a conventional goal is violated.

SOUZA (2012) presents the Zanshin framework, which is part of the baseline of the MULAS framework. It extends goal models with control theory concepts, allowing to define the requirements of adaptive systems. As described in Chapter 2, the Zanshin framework only tackles requirements-based adaptation.

More recently, CHEN et al. (2014) also proposed to integrate requirements and design concerns in the context of adaptive software systems. While their work is focused on the structural view, ours includes the behavioral view.

8.3.1.2 Architecture-centric

Most of the approaches for architecture-based adaptation is focused on components and connectors models. Rainbow (GARLAN et al., 2004) uses the Acme language (GARLAN; MONROE; WILE, 2010) to describe a software architecture, and it has its own language to define adaptation strategies (CHENG; GARLAN, 2012). A standard component enacts an external feedback loop, which can trigger the execution of custom-made scripts that are specific for the system being instrumented. When faced with different possibilities, it uses utility functions to decide which adaptation to perform.

The K-Component framework (DOWLING; CAHILL, 2001) define adaptation contracts, on which component interfaces and connectors are static whereas component instances and connector properties are dynamic. Adaptation is triggered by the violation of architectural constraints, in terms of configurations, components, connectors and their properties. Instead of relying on an Architecture Description Language (ADL), the K-Component framework extract information about the architecture of the instrumented system directly from C++ source code.

The work by ALLEN; DOUENCE; GARLAN (1998) adds control mechanisms to components described with the Wright ADL. Adaptation is defined on a Configuror program, where triggers are defined based on properties of components and connectors of the instrumented system and the adaptation itself is described as the attaching and detaching of components and connectors.

OREIZY; MEDVIDOVIC; TAYLOR (1998) classify architectural adaptation in four types: component addition, component removal, component replacement and reconfiguration (change of connector bindings). The adaptation itself can be performed manually, through an interactive model, or automatically, through its own language called ArchShell (OREIZY, 1996).

The StarMX framework (ASADOLLAHI; SALEHIE; TAHVILDARI, 2009) is able to manage the adaptation of Java systems based on custom policies containing decision-action rules. These policies can access properties of the system components, also being able of invoking the Java methods of these components. As Rainbow, this framework provides a set of standard components that facilitate the implementation of self-adaptive systems.

The work by CETINA et al. (2009) use condition rules that trigger the activation or deactivation of system features. Based on these conditions, it plugs or unplugs components and connectors from the instrumented system. Unlike other works here discussed, this framework support the definition of high-level conditions, such as *the house is empty* and *the alarm is failing* (examples from a smart home system).

8.3.2 Architectural Design and Derivation

In the following subsections we present approaches for creating architectural models based on requirements models.

8.3.2.1 Derivation of component models

The SIRA approach (BASTOS; CASTRO, 2005) focuses on a systematic way to assist the transition from requirements models in $i*$ to architecture. It describes a software system from the perspective of an organization, as stated by the Tropos methodology (CASTRO; KOLP; MYLOPOULOS, 2002). Both requirements and architecture models are described using the $i*$ language (YU et al., 2011). An organizational architectural style is chosen based on a catalogue of non-functional requirements presented in KOLP; GIORGINI; MYLOPOULOS (2006). $i*$ elements, at requirements level, are grouped, inside an actor, according to their contribution to achieve some responsibilities. Then, an architectural design model is created by considering the similarities between the requirements actors and the architectural actors present in the chosen organizational architectural style.

LAMSWEERDE (2003) defines a method to produce architectural models from KAOS requirements models. In that approach, requirements specifications are gradually refined to meet specific architectural constraints of the domain and an abstract architectural draft is generated from functional specifications. The resulting architecture is recursively refined to meet the various non-functional goals analyzed during the requirements activities. It relies on KAOS modeling language, which consists of a graphical tree and a formal language.

In SILVA et al. (2007) a set of mapping rules is proposed between the Aspectual Oriented V-graph (AOV-graph) and the AspectualACME, an ADL based in Acme. This approach does not address the adaptability softgoal. Each element (goal/softgoal/task) present in an AOV-graph is mapped to an element of AspectualACME, depending on its position in the graph hierarchy. The information about the source of each element in the AOV-graph is registered in the properties of a component or a port in AspectualACME. These properties make it possible to keep the traceability and propagation of change from AspectualACME to AOV-graph models and vice-versa.

The CBSP approach (GRüNBACHER et al., 2001) supports the derivation from textual requirements to architecture. It is based on traversing a list of requirements and identifying information relevant to the system's architecture. The identified information is then refined towards the specification of an architecture.

The STREAM approach (Strategy for Transition between Requirements and Architectural Models) takes requirements models on the i* language as input and provides components and connectors models as output (CASTRO et al., 2012). With the support of transformation rules, the architect refactors the requirements model considering modularity criteria, and derive component and connector models that reflect the structure of the refactored i* model. STREAM-A (PIMENTEL et al., 2012) is an extension of STREAM that includes support for requirements-based adaptation.

8.3.2.2 Derivation of behavioral models

There are other approaches that use goal models as a starting point for the definition of system behavior. LIASKOS et al. (2012) support the definition of systems where the order of task execution is constrained at design time through Linear Temporal Logic (LTL) expressions. Instead of architectural design, the goal of this approach is to support the customization of system behavior.

The work by YU et al. (2008) uses the hierarchy of a goal model to derive the structure of a statechart, where higher level elements are superstates and lower level elements are transitions. The order of execution of the states is defined by the visual order of the elements, from left to right, along with annotations for expressing alternatives, sequentiality, parallelism or delegation. One of the emphases of this approach is to ensure that the variability expressed in the requirements model is preserved in the architectural models.

LETIER et al. (2008) start with KAOS models and derive Labeled Transition Systems (LTS), which resemble statecharts. The goal models in KAOS present a temporal formalization in LTL. Leaf-level goals are assigned to agents and can be refined into operations, which are specified through pre-conditions, post-conditions and triggers. Thus, the derivation of the behavioral models is mostly (but not only) a translation of formalism, with little human input.

There are a number of approaches that handle the derivation of statecharts from scenarios, which describe specific interactions between users and the system. In these approaches, the focus is on how to combine different scenarios and thus identify the overall states of the system. WHITTLE; SCHUMANN (2000) use UML sequence diagrams and generate UML statecharts. HAREL; KUGLER; PNUELI (2005) use live sequence charts and an extension to traditional sequence diagrams that supports optional and mandatory interactions. A synthesis algorithm based on model checking generates a statechart. The hierarchy of these diagrams is based on UML models provided as input.

GOMAA; SHIN (2003) discuss the occurrence of variability in different architectural models. In the case of statecharts, variability is expressed through alternative transitions related to different features of the system. Thus, certain states will only be entered if a specific feature is available.

8.3.3 Summary of related work

Table 8.1 shows the major characteristics of each related work presented in this section — whether they support adaptation, and of which kind, as well as whether and which type of architectural derivation is supported. For comparison, the MULAS framework is included as the last entry in that table.

As can be seen in Table 8.1, MULAS differentiate from other proposals by not only supporting requirements- and design-related adaptation, but also by supporting the systematic design of the system behavior.

Table 8.1: Comparison of related work

Work	Adaptation			Derivation	
	Requirements	**Design (structure)**	**Design (behavior)**	**Structure**	**Behavior**
LAPOUCHNIAN; MYLOPOULOS (2009)	✓				
ALI; DALPIAZ; GIORGINI (2010)	✓				
DALPIAZ; GIORGINI; MYLOPOULOS (2009)	✓				
MORANDINI; PENSERINI; PERINI (2008)	✓				
JIAN et al. (2010)	✓	✓			
QURESHI et al. (2010)	✓	*partial*			
BENCOMO; WHITTLE; SAWYER (2010)	✓				
BARESI; PASQUALE (2010)	✓				
SOUZA (2012)	✓				
CHEN et al. (2014)	✓	✓			
GARLAN et al. (2004)		✓			
DOWLING; CAHILL (2001)		✓			
ALLEN; DOUENCE; GARLAN (1998)		✓			
OREIZY; MEDVIDOVIC; TAYLOR (1998)		✓			
ASADOLLAHI; SALEHIE; TAHVILDARI (2009)		✓			
CETINA et al. (2009)	*partial*	✓			
BASTOS; CASTRO (2005)				✓	
LAMSWEERDE (2003)				✓	
SILVA et al. (2007)				✓	
GRüNBACHER et al. (2001)				✓	
CASTRO et al. (2012)				✓	
PIMENTEL et al. (2012)	✓			✓	
LIASKOS et al. (2012)					✓
YU et al. (2008)					✓
LETIER et al. (2008)					✓
WHITTLE; SCHUMANN (2000)					✓
HAREL; KUGLER; PNUELI (2005)					✓
GOMAA; SHIN (2003)					✓
MULAS	✓	*partial*	✓		✓

8.4 Considerations

We have presented a systematic process for deriving architectural behavioral models —- namely, statecharts —- from requirements models, supporting the Twin Peaks model (NUSEIBEH, 2001) of software design. Through a series of incremental refinements the architect can move towards architecture, by (i) adding design elements (tasks and constraints), (ii) adding behavioral refinements (in the form of flow expressions), and (iii) generating statechart models from (possibly incomplete) models. Acknowledging the inherent variability of the design process, where different solutions for a single problem can be devised, the design goal model supports the documentation of alternative design elements and alternative behavior refinements, which can lead to the generation of multiple (alternative) statecharts. Since the resulting models are statecharts without any extension, it is amenable for validation, simulation and code generation using existing tools. Moreover, the properties of these models can be checked and proved using one of the several formalizations of statecharts (for instance, LEVI (1997)).

A key element of our proposal is the integration of requirements and design elements in a single model. The use of different views in the supporting tool allows to seamlessly navigate between requirements and design elements. Moreover, the integration with the Zanshin framework allows to support both requirements-based and architectural-based adaptation (ANGELOPOULOS; SOUZA; PIMENTEL, 2013).

While this work provides a process and supporting mechanisms that facilitate the derivation of statecharts, key decisions on what flow to adopt still rest entirely with the architect. The integration of this work with ontology-based approaches (AMELLER; FRANCH, 2011; DERMEVAL et al., 2015) and the identification of behavioral patterns could add additional guidance, thereby improving the quality of resulting models.

The design goal model describes some of the architectural design decisions of the software project. However, it does not capture some of the information that is deemed relevant in the context of software architecture documentation, such as rationale, status and source (DERMEVAL et al., 2012).

Lastly, whereas flow expressions proved to be useful as an intermediary artifact between goal models and statecharts, they are an additional element to be maintained by the architect during system evolution. For this reason, a prototype tool (GATO) was developed aiming to reduce the effort of creating and maintaining design goal models, as well as to support statechart derivation.

8.5 Limitations

This work presents a series of limitations, regarding the following aspects:

- **Expressiveness of the design goal model —** The design goal model proposed in this thesis is based on the extended goal model, originally proposed in LAPOUCHNIAN (2011) and further developed on SOUZA (2012). That extension includes awareness requirements and parameters, which are relevant as they correspond to the control theory concepts of *reference value* and *control input*. However, as a result of the focus on these control theory concepts, two other important concepts have been partially neglected: contribution links and context. The explicit use of contribution links and context annotations may improve the expressiveness of the design goal model. Nonetheless, it is necessary to balance this expressivity with the complexity of the proposed model.

Moreover, since the focus of this research project was on system behavior, we were not able to include proper support to other architectural design elements, such as components and connectors. although not as fully fledged as it could be, some support is provided, since variations on these elements can be expressed as parameters, and decisions regarding these elements can be expressed through design constraints.

- **Heuristics for selecting optimal flows —** In the MULAS framework we propose the use of flow expressions to define the possible flows of the system. However, we do not provide any guidance that helps the architect in the decision of which flow may be best in different contexts and scenarios. Further investigation is required in order to identify heuristics, patterns, or techniques to facilitate such decision.

- **Derivation patterns —** In this thesis we have proposed a series of patterns that allow the generation of base statecharts from a goal model annotated with flow expressions. This derivation results in statecharts of a particular style, which may not be suitable for every software architect: goals, tasks, and design tasks of the design goal model are mapped onto states in a statechart. However, different architects may prefer different styles — for instance, YU et al. (2005) maps leaf elements in a goal model to transition actions. It is also possible to envision cases where it may be desired to map some elements onto events, instead of onto states.

  The decision of providing a uniform mapping is to systematize the process. Further evaluation is necessary in order to identify cases where this mapping may not be satisfactory, as well as how to handle these cases.

- **Modularity of the resulting statecharts —** The statechart resulting from the automatic derivation does not follow good practices related to modularity (e.g., transitions between super-states). As result, sometimes the models has more transitions than what is strictly necessary, increasing the complexity of the statechart. The derivation is performed in this way in order to allow the architect to specify each transition, with its events and conditions, as it is not possible to know beforehand whether a transition may be grouped with other transitions.

In order to mitigate this possible, we believe that it is possible to develop an algorithm that automatically refactors the statechart after the transitions are specified.

- **Tool support —** In this thesis we have described the GATO (Goal to Architecture) tool, which was developed specifically to support the MULAS framework. This tool is functional, which is evidenced by its use in the creation of all the goal models depicted in this thesis. However, more effort is required in order to make the tool suitable for public use, related not only to actual development but also to the creation of user documentation, such as user guides or tutorials.

- **Compositional adaptation —** Parameterized adaptation is adaptation related to the modification of variables. In contrast, *compositional adaptation* is related to modifying structural parts of the system (MCKINLEY et al., 2004). While we have conducted early endeavors on the latter (PIMENTEL et al., 2012) during this research, the MULAS framework is focused only on the former.

8.6 Future Work

We expect to continue this research project with the following improvements:

1. **Other enhancements for the supporting tool.** We plan to investigate which improvements could be made to the GATO tool in order to increase productivity and usability. Some of the features that could be developed are collaborative modeling, server-side saving and versioning, customized views, and tablet-tailored user interface.
2. **Further architectural adaptation.** In this thesis we cover architectural adaptation in terms of design decisions and alternative flows for statecharts. We plan to expand on this repertoire by also supporting adaptation related to components and connectors models, as well as deployment models.
3. **Further adaptation expressiveness.** The Zanshin framework supports the definition of which elements must be monitored and what adaptations can be performed. However, existing approaches handle additional adaptation concerns, such as context (VILELA et al., 2015; DALPIAZ; GIORGINI; MYLOPOULOS, 2013; VILELA et al., 2015) and uncertainty (WHITTLE et al., 2010; SALAY et al., 2013). We plan to analyze which additional concerns would be relevant to support in the MULAS framework and how they can be integrated in the current feedback loop.
4. **Further modeling expressiveness.** Some architectural decisions are dependent on other decisions — for instance, the decision of which application server to use depends on the adopted programming language. As future work, we plan to investigate

if expressing this kind of dependencies in design goal models is relevant and how it could be included.

5. **Heuristics and guidelines.** In order to facilitate the use and improve the results obtained with this framework, further guidance can be provided on the decision-making aspects of the proposed architectural design process.
6. **Further validation and improvements of the MULAS process.** By applying the MULAS framework to larger projects, as well as to projects on industrial settings, further limitations could be identified and new improvements could be devised.

References

ABDELWAHED, S.; KANDASAMY, N. A Control-Based Approach to Autonomic Performance Management in Computing Systems. In: **Autonomic Computing - Concepts, Infrastructure, and Applications**. [S.l.]: CRC Press, 2007. p.149–168.

ALI, R.; DALPIAZ, F.; GIORGINI, P. A Goal-based Framework for Contextual Requirements Modeling and Analysis. **Requirements Engineering Journal**, [S.l.], v.15, n.4, p.439–458, 2010.

ALLEN, R.; DOUENCE, R.; GARLAN, D. Specifying and analyzing dynamic software architectures. **1998 Conference on Fundamental Approaches to Software Engineering**, [S.l.], p.21–37, 1998.

AMELLER, D.; FRANCH, X. Ontology-based Architectural Knowledge representation : structural elements module. In: ADVANCED INFORMATION SYSTEMS ENGINEERING WORKSHOPS. **...** [S.l.: s.n.], 2011. p.296–301.

ANGELOPOULOS, K.; SOUZA, V. E. S.; PIMENTEL, J. Requirements and Architectural Approaches to Adaptive Software Systems: a comparative study. **SEAMS**, [S.l.], p.23–32, 2013.

ASADOLLAHI, R.; SALEHIE, M.; TAHVILDARI, L. StarMX: a framework for developing self-managing java-based systems. **2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems**, [S.l.], p.58–67, May 2009.

ASTRöM, K. J.; MURRAY, R. M. **Feedback Systems - An Introduction for Scientists and Engineers**. [S.l.]: Princeton university press, 2012. 408p.

BACHMANN, F. et al. **Documenting Software Architecture : documenting behavior**. [S.l.: s.n.], 2002. (January).

BALSER, M.; BäUMLER, S.; KNAPP, A.; REIF, W.; THUMS, A. Interactive verification of UML state machines. **Formal Methods and Software Engineering**, [S.l.], v.LNCS3308, p.434–448, 2004.

BARESI, L.; PASQUALE, L. Live goals for adaptive service compositions. In: ICSE WORKSHOP ON SOFTWARE ENGINEERING FOR ADAPTIVE AND SELF-MANAGING SYSTEMS (SEAMS '10), 2010., New York, USA. **...** ACM Press, 2010. p.114–123.

BASTOS, L. R. D.; CASTRO, J. From requirements to multi-agent architecture using organisational concepts. **SIGSOFT Software Engineering Notes**, [S.l.], v.30, n.4, p.1–7, 2005.

BENCOMO, N.; WHITTLE, J.; SAWYER, P. Requirements reflection: requirements as runtime entities. **ACM/IEEE 32nd International Conference on Software Engineering**, [S.l.], p.199–202, 2010.

BERRY, D.; CHENG, B. H. C.; ZHANG, J. The four levels of requirements engineering for and in dynamic adaptive systems. In: INTERNATIONAL WORKSHOP ON

REQUIREMENTS ENGINEERING FOUNDATION FOR SOFTWARE QUALITY (REFSQ), 11., Porto, Portugal. **...** [S.l.: s.n.], 2005. p.5.

BJORK, R. C. An Example of Object-Oriented Design: an atm simulation. **http://www.math-cs.gordon.edu/courses/cs211/ATMExample/**, [S.l.], 2004.

BOER, R. C. de; VLIET, H. van. On the similarity between requirements and architecture. **Journal of Systems and Software**, [S.l.], v.82, n.3, p.544–550, Mar. 2009.

BROOKS, F. P. No Silver Bullet — Essence and Accident in Software Engineering. **Information Processing**, [S.l.], p.1069–1076, 1986.

BRUN, Y. et al. Engineering Self-Adaptive Systems through Feedback Loops. In: **Software Engineering for Self-Adaptive Systems**. [S.l.]: Springer Berlin Heidelberg, 2009. p.48–70.

CARVER, J.; JACCHERI, L.; MORASCA, S.; SHULL, F. Issues in using students in empirical studies in software engineering education. **Proceedings of the Ninth International Software Metrics Symposium**, [S.l.], p.239–249, 2003.

CASTRO, J.; KOLP, M.; MYLOPOULOS, J. Towards Requirements-Driven Information Systems Engineering: the tropos project. **Information Systems**, [S.l.], v.27, n.6, p.365–389, 2002.

CASTRO, J.; LUCENA, M.; SILVA, C.; ALENCAR, F.; SANTOS, E.; PIMENTEL, J. Changing attitudes towards the generation of architectural models. **Journal of Systems and Software**, [S.l.], v.85, n.3, p.463–479, Mar. 2012.

CETINA, C.; GINER, P.; FONS, J.; PELECHANO, V. Autonomic computing through reuse of variability models at runtime: the case of smart homes. **Computer**, [S.l.], n.October, p.46–52, 2009.

CHEN, B.; PENG, X.; YU, Y.; NUSEIBEH, B.; ZHAO, W. Self-adaptation Through Incremental Generative Model Transformations at Runtime. **36th International Conference on Software Engineering**, [S.l.], p.676–687, 2014.

CHENG, B. H. C. et al. Software engineering for self-adaptive systems: a research roadmap. **Software Engineering for Self-Adaptive Systems - LNCS**, [S.l.], v.5525/2009, p.1–26, 2009.

CHENG, S.-W.; GARLAN, D. Stitch: a language for architecture-based self-adaptation. **Journal of Systems and Software**, [S.l.], v.85, n.12, p.2860–2875, Dec. 2012.

CHENG, S. W.; GARLAN, D.; SCHMERL, B. Evaluating the effectiveness of the rainbow self-adaptive system. In: ICSE WORKSHOP ON SOFTWARE ENGINEERING FOR ADAPTIVE AND SELF-MANAGING SYSTEMS, SEAMS 2009, 2009. **Proceedings...** [S.l.: s.n.], 2009. p.132–141.

CHOI, H.; YEOM, K. An approach to software architecture evaluation with the 4+1 view model of architecture. **Ninth Asia-Pacific Software Engineering Conference, 2002.**, [S.l.], p.286–293, 2002.

CHUNG, L.; NIXON, B.; YU, E.; MYLOPOULOS, J. Non-functional Requirements. **Software Engineering**, [S.l.], 2000.

DALPIAZ, F.; BORGIDA, A.; HORKOFF, J.; MYLOPOULOS, J. Runtime Goal Models : keynote. In: IEEE SEVENTH INTERNATIONAL CONFERENCE ON RESEARCH CHALLENGES IN INFORMATION SCIENCE. **...** [S.l.: s.n.], 2013.

DALPIAZ, F.; GIORGINI, P.; MYLOPOULOS, J. An architecture for requirements-driven self-reconfiguration. **Advanced Information Systems Engineering**, [S.l.], 2009.

DALPIAZ, F.; GIORGINI, P.; MYLOPOULOS, J. Adaptive socio-technical systems: a requirements-based approach. **Requirements engineering**, [S.l.], v.18, n.1, p.1–24, Sept. 2013.

DERMEVAL, D. et al. Applications of ontologies in requirements engineering: a systematic review of the literature. **Requirements Engineering journal**, [S.l.], 2015.

DERMEVAL, D.; PIMENTEL, J.; SILVA, C.; CASTRO, J.; SANTOS, E.; GUEDES, G. STREAM-ADD – Supporting the Documentation of Architectural Design Decisions in an Architecture Derivation Process. In: IEEE 36TH INTERNATIONAL CONFERENCE ON COMPUTER SOFTWARE AND APPLICATIONS, 2012. **...** [S.l.: s.n.], 2012. n.i, p.602–611.

DIAS, M. S.; VIEIRA, M. E. R. Software architecture analysis based on statechart semantics. In: SOFTWARE SPECIFICATION AND DESIGN, 2000. TENTH INTERNATIONAL WORKSHOP ON. **...** [S.l.: s.n.], 2000. p.133–137.

DOWLING, J.; CAHILL, V. The K-Component Architecture Meta-model for Self-Adaptive Software. **Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION '01)**, [S.l.], p.81–88, 2001.

EGYED, A.; WILE, D. Statechart simulator for modeling architectural dynamics. In: SOFTWARE ARCHITECTURE, 2001. PROCEEDINGS. WORKING IEEE/IFIP CONFERENCE ON. **...** [S.l.: s.n.], 2001. p.87–96.

EGYED, A.; WILE, D. Statechart simulator for modeling architectural dynamics. **Proceedings of the Working IEEE/IFIP Conference on Software Architecture, 2001**, [S.l.], n.August, p.87–96, 2001.

EMERY, F. **Characteristics of Socio-Technical Systems**. London: Tavistock Institute, 1959. (1959).

FEATHER, M.; FICKAS, S.; FINKELSTEIN, A.; LAMSWEERDE, A. Requirements and specification exemplars. **Automated Software Engineering**, [S.l.], v.4, n.4, p.419–438, 1997.

FEATHER, M.; FICKAS, S.; LAMSWEERDE, A. van; PONSARD, C. Reconciling system requirements and runtime behavior. **Proceedings Ninth International Workshop on Software Specification and Design**, [S.l.], p.50–59, 1998.

FERRENTINO, A. B.; MILLS, H. D. State machines and their semantics in software engineering. In: IEEE COMPSAC'77 CONFERENCE. **Proceedings...** [S.l.: s.n.], 1977. p.242–251.

FILIERI, A.; GHEZZI, C.; LEVA, A.; MAGGIO, M. Reliability-driven dynamic binding via feedback control. In: ICSE WORKSHOP ON SOFTWARE ENGINEERING FOR ADAPTIVE AND SELF-MANAGING SYSTEMS. **...** [S.l.: s.n.], 2012. v.2, p.43–52.

FRANCH, X. A Method for the definition of metrics over i* models. **Advanced Information Systems Engineering**, [S.l.], v.5565 LNCS, p.201–215, 2009.

FRANCH, X. et al. Goal-Driven Adaptation of Service-Based Systems from Runtime Monitoring Data. In: IEEE 35TH ANNUAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE WORKSHOPS, 2011. **...** IEEE, 2011. p.458–463.

GANE, C. P.; SARSON, T. **Structured systems analysis: tools and techniques**. [S.l.]: Prentice Hall Professional Technical Reference, 1979.

GARLAN, D.; CHENG, S.-W.; HUANG, A.-C.; SCHMERL, B.; STEENKISTE, P. Rainbow: architecture-based self-adaptation with reusable infrastructure. **Computer**, [S.l.], v.37, n.10, p.46–54, Oct. 2004.

GARLAN, D.; MONROE, R.; WILE, D. Acme : an architecture description interchange language. **Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research (CASCON)**, [S.l.], p.1–15, 1997.

GARLAN, D.; MONROE, R.; WILE, D. Acme: an architecture description interchange language. **CASCON First Decade High Impact Papers**, [S.l.], 2010.

GENERO, M.; MIRANDA, D.; PIATTINI, M. Defining and validating metrics for UML statechart diagrams. **6th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering**, [S.l.], p.120–136, 2002.

GHANBARI, H.; SIMMONS, B.; LITOIU, M.; BARNA, C.; ISZLAI, G. Optimal autoscaling in a IaaS cloud. **Proceedings of the 9th international conference on Autonomic computing - ICAC '12**, New York, New York, USA, p.173–178, 2012.

GOMAA, H.; SHIN, M. Variability in Multiple-View Models of Software Product Lines. In: INTERNATIONAL WORKSHOP ON SOFTWARE VARIABILITY MANAGEMENT (SVM). **...** [S.l.: s.n.], 2003. p.63–68.

GRALHA, C.; GOULÃO, M.; ARAúJO, J. Identifying modularity improvement opportunities in goal-oriented requirements models. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, [S.l.], v.8484 LNCS, p.91–104, 2014.

GRAU, G.; FRANCH, X. On the adequacy of i* models for representing and analyzing software architectures. **Advances in conceptual modeling: foundations and applications - LNCS**, [S.l.], v.4802, p.296–305, 2007.

GRAU, G.; FRANCH, X.; MAIDEN, N. PRiM: an i*-based process reengineering method for information systems specification. **Information and Software Technology**, [S.l.], v.50, n.1-2, p.76–100, 2008.

GREENSPAN, S.; MYLOPOULOS, J.; BORGIDA, a. On formal requirements modeling languages: rml revisited. **Proceedings of 16th International Conference on Software Engineering**, [S.l.], n.May, p.1–13, 1994.

GRüNBACHER, P.; EGYED, A.; WAY, A.; PLACE, W.; REY, M. D.; MEDVIDOVIC, N. Reconciling software requirements and architectures: the cbsp approach. In: FIFTH IEEE INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING. **Proceedings...** IEEE Comput. Soc, 2001. p.202–211.

GURP, J. V.; BOSCH, J.; SVAHNBERG, M. On the notion of variability in software product lines. **Proceedings of the Working IEEE/IFIP Conference on Software Architecture**, [S.l.], p.45–54, 2001.

GURP, J. van; BOSCH, J. Design erosion: problems and causes. **Journal of Systems and Software**, [S.l.], v.61, n.2, p.105–119, Mar. 2002.

HAREL, D. Statecharts: a visual formalism for complex systems. **Science of computer programming**, [S.l.], v.8, n.3, p.231–274, 1987.

HAREL, D.; KUGLER, H.; PNUELI, A. Synthesis Revisited: generating statechart models from scenario-based requirements. In: **Formal Methods in Software and Systems Modeling**. [S.l.: s.n.], 2005. n.287, p.309–324.

HE, Y.; YE, Z.; FU, Q.; ELNIKETY, S. Budget-based control for interactive services with adaptive execution. **Proceedings of the 9th international conference on Autonomic computing - ICAC '12**, New York, New York, USA, p.105–114, 2012.

HEINIS, T.; PAUTASSO, C. Automatic configuration of an autonomic controller: an experimental study with zero-configuration policies. **Autonomic Computing, 2008. ICAC'08. International Conference on**, [S.l.], p.67–76, 2008.

HELLERSTEIN, J. L.; DIAO, Y.; PAREKH, S.; TILBURY, D. M. **Feedback control of computing systems**. [S.l.]: John Wiley & Sons, 2004.

HEVNER, A. R.; MARCH, S. T.; PARK, J. Design Science in Information Systems Research. **MIS Quarterly**, [S.l.], v.28, n.1, p.75–105, 2004.

HOARES, C. A. R. Quicksort. **The Computer Journal**, [S.l.], v.5, n.1, p.10–15, 1962.

HORKOFF, J. et al. Taking Goal Models Downstream: a systematic roadmap. **8th International Conference on Research Challenges in Information Science**, [S.l.], 2014.

HORN, P. **Autonomic computing: ibm's perspective on the state of information technology**. [S.l.: s.n.], 2001.

JACOBSON, I.; SPENCE, I.; BITTNER, K. **Use-case 2.0**. [S.l.]: Ivar Jacobsen International, 2011. n.December.

JIAN, Y.; LI, T.; LIU, L.; YU, E. Goal-Oriented Requirements Modelling for Running Systems. In: INTERNATIONAL WORKSHOP ON REQUIREMENTS@RUN-TIME, 1., Sidney, Australia. **...** [S.l.: s.n.], 2010.

JURETA, I. J.; MYLOPOULOS, J.; FAULKNER, S. Revisiting the core ontology and problem in requirements engineering. **Proceedings of the 16th IEEE International Requirements Engineering Conference, RE'08**, [S.l.], v.2008, p.71–80, 2008.

KAYE, J. M.; CASTILLO, D. **Flash MX for interactive simulation**. [S.l.]: Cengage Learning, 2003.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. **Computer**, [S.l.], v.36, n.1, p.41–50, 2003.

KLEPPE, A. G.; WARMER, J. B.; BAST, W. **MDA explained, the model driven architecture: practice and promise**. [S.l.]: Addison-Wesley Professional, 2003.

KOLP, M.; GIORGINI, P.; MYLOPOULOS, J. Multi-Agent Architectures as Organizational Structures. **Autonomous Agents and Multi-Agent Systems**, [S.l.], v.13, n.1, p.3–25, July 2006.

KOTONYA, G.; SOMMERVILLE, I. Requirements engineering with viewpoints. **Software Engineering Journal**, [S.l.], v.11, n.1, p.5–18, 1996.

KOTONYA, G.; SOMMERVILLE, I. **Requirements Engineering: processes and techniques**. [S.l.]: John Wiley & Sons, 1998.

KRUCHTEN, P. An Ontology of Architectural Design Decisions in Software-Intensive Systems. **2nd Groningen Workshop Software Variability**, [S.l.], p.54–61, 2004.

LAMSWEERDE, A. V. From System Goals to Software Architecture. **Formal Methods for Software Architectures - LNCS**, [S.l.], v.2804/2003, p.25–43, 2003.

LAMSWEERDE, A. van. Goal-oriented requirements engineering: a guided tour. **Proceedings Fifth IEEE International Symposium on Requirements Engineering**, [S.l.], p.249–262, 2001.

LAMSWEERDE, a. van; DARIMONT, R.; MASSONET, P. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. **Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)**, [S.l.], p.194–203, 1995.

LAPOUCHNIAN, A. **Exploiting Requirements Variability for Software Customization and Adaptation**. 2011. 227p. Ph.D. Thesis — University of Toronto.

LAPOUCHNIAN, A.; MYLOPOULOS, J. Modeling domain variability in requirements engineering with contexts. **Conceptual Modeling - ER 2009 - LNCS**, Gramado, Brazil, v.5829/2009, p.115–130, 2009.

LAPOUCHNIAN, A.; YU, Y.; LIASKOS, S.; MYLOPOULOS, J. Requirements-driven design of autonomic application software. **Proceedings of the 16th IBM Centre for Advanced Studies Conference**, [S.l.], 2006.

LETIER, E.; KRAMER, J.; MAGEE, J.; UCHITEL, S. Deriving event-based transition systems from goal-oriented requirements models. **Automated Software Engineering**, [S.l.], v.15, n.2, p.175–206, May 2008.

LEVI, F. **Verification of Temporal and Real-Time Properties of Statecharts**. 1997. Ph.D. Thesis — University of Pisa.

LIAN, J.; HU, Z.; SHATZ, S. M. Simulation-based analysis of UML statechart diagrams: methods and case studies. **Software Quality Journal**, [S.l.], v.16, n.1, p.45–78, 2008.

LIASKOS, S.; KHAN, S. M.; LITOIU, M.; JUNGBLUT, M. D.; ROGOZHKIN, V.; MYLOPOULOS, J. Behavioral adaptation of information systems through goal models. **Information Systems**, [S.l.], v.37, n.8, p.767–783, Dec. 2012.

LIKERT, R. A technique for the measurement of attitudes. **Archives of psychology**, [S.l.], v.22, n.140, 1932.

MAHONEY, M.; ELRAD, T. Distributing Statecharts to Handle Pervasive Crosscutting Concerns. **Building Software for Pervasive Computing Workshop at OOPSLA '05**, [S.l.], 2005.

MAIDEN, N.; YU, E.; FRANCH, X.; MYLOPOULOS, J. **Proceedings of the iStar Showcase '11 - Exploring the Goals of your Systems and Businesses**. London: [s.n.], 2011. 1–98p.

MCKINLEY, P. K.; SADJADI, S. M.; KASTEN, E. P.; CHENG, B. H. C. **A Taxonomy of Compositional Adaptation**. [S.l.: s.n.], 2004. (May).

MIRANDA, D.; GENERO, M.; PIATTINI, M. Empirical validation of metrics for UML statechart diagrams. **Enterprise Information Systems**, [S.l.], p.87–95, 2003.

MOODY, D. The "Physics" of Notations: toward a scientific basis for constructing visual notations in software engineering. **IEEE Transactions on Software Engineering**, [S.l.], v.35, n.6, p.756–779, Nov. 2009.

MOODY, D. L.; HEYMANS, P.; MATULEVICIUS, R. Visual syntax does matter: improving the cognitive effectiveness of the i* visual notation. **Requirements Engineering**, [S.l.], v.15, n.2, p.141–175, May 2010.

MORANDINI, M.; PENSERINI, L.; PERINI, A. Towards goal-oriented development of self-adaptive systems. In: ICSE WORKSHOP ON SOFTWARE ENGINEERING FOR ADAPTIVE AND SELF-MANAGING SYSTEMS (SEAMS '08), 2008., New York, USA. **...** ACM, 2008. p.9–16.

NETO, G. G. d. C.; GOMES, A. S.; CASTRO, J. Mapping Activity Theory Diagrams into i* Organizational Models. **Journal of Computer Science & Technology**, [S.l.], v.5, n.2, p.57–63, 2005.

NIAZ, I. A.; TANAKA, J. Code generation from UML statecharts. In: IASTED INTERNATIONAL CONF. ON SOFTWARE ENGINEERING AND APPLICATION (SEA 2003), MARINA DEL REY, 7. **Proceedings...** [S.l.: s.n.], 2003. p.315–321.

NUSEIBEH, B. Weaving together requirements and architectures. **Computer**, [S.l.], v.34, n.3, p.115–119, Mar. 2001.

NUSEIBEH, B.; EASTERBROOK, S. Requirements engineering: a roadmap. **in ICSE '00: Proceedings of the Conference on The Future of Software Engineering. New York, NY, USA: ACM**, [S.l.], p.35–46, 2000.

OMG. **Business Process Model and Notation (BPMN) Version 2.0**. [S.l.: s.n.], 2011. (January).

OMG. **Information technology - Object Management Group Object Constraint Language (OCL)**. [S.l.]: OMG, 2012. (April).

OREIZY, P. **Issues in the runtime modification of software architectures. UCI-ICS-TR-96-35**. [S.l.]: University of California, Irvine, 1996.

OREIZY, P.; MEDVIDOVIC, N.; TAYLOR, R. Architecture-based runtime software evolution. **Proceedings of the 20th international conference on Software engineering**, [S.l.], p.177–186, 1998.

PASTOR, O.; MOLINA, J. C. **Model-driven architecture in practice: a software production environment based on conceptual modeling**. [S.l.]: Springer, 2007.

Pettit Iv, R. G.; GOMAA, H. Modeling State-Dependent Objects using Colored Petri Nets. **CPN 01 Workshop on Modeling of Objects, Components, and Agents**, [S.l.], 2001.

PIMENTEL, J.; CASTRO, J.; MYLOPOULOS, J.; ANGELOPOULOS, K.; SOUZA, V. E. S. From Requirements to Statecharts via Design Refinement - in press. **Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC 2014**, [S.l.], 2014.

PIMENTEL, J.; FRANCH, X.; CASTRO, J. Measuring Architectural Adaptability in i* Models. In: XIV IBERO-AMERICAN CONFERENCE ON SOFTWARE ENGINEERING (CIBSE 2011). **...** [S.l.: s.n.], 2011. p.115–128.

PIMENTEL, J.; LUCENA, M.; CASTRO, J.; SILVA, C.; SANTOS, E.; ALENCAR, F. Deriving software architectural models from requirements models for adaptive systems: the stream-a approach. **Requirements Engineering**, [S.l.], v.17, n.4, p.259–281, June 2012.

PIMENTEL, J.; SANTOS, E.; DERMEVAL, D.; CASTRO, J.; FINKELSTEIN, A. Towards Architectural Evolution through Model Transformations. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING (SEKE), 24. **Proceedings...** [S.l.: s.n.], 2012. p.448–451.

QURESHI, N. A.; PERINI, A.; ERNST, N. A.; MYLOPOULOS, J. Towards a Continuous Requirements Engineering Framework for Self-Adaptive Systems. In: INTERNATIONAL WORKSHOP ON REQUIREMENTS@RUN-TIME, 1., Sidney, Australia. **...** [S.l.: s.n.], 2010.

ROBERTSON, S.; ROBERTSON, J. **Mastering the requirements process: getting requirements right**. 3rd Editio.ed. [S.l.]: Addison-Wesley, 2012.

ROLLAND, C.; ACHOUR, C. B. Guiding the construction of textual use case specifications. **Data & Knowledge Engineering Journal**, [S.l.], v.25, n.March, p.125–160, 1998.

SALAY, R.; CHECHIK, M.; HORKOFF, J.; SANDRO, A. D. Managing requirements uncertainty with partial models. **Requirements Engineering**, [S.l.], v.18, n.2, p.107–128, Apr. 2013.

SANTANDER, V.; CASTRO, J. Deriving use cases from organizational modeling. In: IEEE JOINT INTERNATIONAL CONFERENCE ON REQUIREMENTS ENGINEERING. **Proceedings...** IEEE Comput. Soc, 2002. n.147192, p.32–39.

SAVOLA, R. M.; HEINONEN, P. Security-Measurability-Enhancing Mechanisms for a Distributed Adaptive Security Monitoring System. **2010 Fourth International Conference on Emerging Security Information, Systems and Technologies**, [S.l.], p.25–34, July 2010.

SHAW, A. C. Software Descriptions with Flow Expressions. **IEEE Transactions on Software Engineering**, [S.l.], v.SE-4, n.3, p.242–254, May 1978.

SILVA, L. F.; BATISTA, T. V.; GARCIA, A.; MEDEIROS, A. L.; MINORA, L. On the symbiosis of aspect-oriented requirements and architectural descriptions. **Early Aspects: Current Challenges and Future Directions - LNCS**, Vancouver, Canada, v.4765/2007, p.75–93, 2007.

SJØ BERG, D. I. K. et al. A survey of controlled experiments in software engineering. **IEEE Transactions on Software Engineering**, [S.l.], v.31, n.9, p.733–753, 2005.

SOARES, M. et al. Automatic Generation of Architectural Models From Goals Models. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING (SEKE), 24. **Proceedings. . .** [S.l.: s.n.], 2012. p.444–447.

SOMMERVILLE, I. **Software Engineering: seventh edition**. [S.l.]: Pearson Education, 2004.

SOUZA, V. E. S. **Requirements-based software system adaptation**. 2012. Ph.D. Thesis — University of Trento, Italy. (June).

SOUZA, V. E. S.; LAPOUCHNIAN, A.; MYLOPOULOS, J. System Identification for Adaptive Software Systems: a requirements engineering perspective. In: CONCEPTUAL MODELING – ER 2011. **. . .** [S.l.: s.n.], 2011. p.346–361.

SOUZA, V. E. S.; LAPOUCHNIAN, A.; MYLOPOULOS, J. Requirements-driven qualitative adaptation. **On the Move to Meaningful Internet Systems: OTM 2012**, [S.l.], v.7565 LNCS, p.342–361, 2012.

SOUZA, V. E. S.; LAPOUCHNIAN, A.; MYLOPOULOS, J. (Requirement) evolution requirements for adaptive systems. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE ENGINEERING FOR ADAPTIVE AND SELF-MANAGING SYSTEMS (SEAMS), 7. **. . .** IEEE, 2012. p.155–164.

SOUZA, V. E. S.; LAPOUCHNIAN, A.; ROBINSON, W. N.; MYLOPOULOS, J. Awareness Requirements. In: LEMOS, R.; GIESE, H.; MüLLER, H. A.; SHAW, M. (Ed.). **Software Engineering for Self-Adaptive Systems II**. [S.l.]: Springer, 2013. p.133–161. (Lecture Notes in Computer Science, v.7475).

STOLZE, M.; RIAND, P.; WALLACE, M.; HEATH, T. Agile Development of Workflow Applications with Interpreted Task Models. **Task Models and Diagrams for User Interface Design**, [S.l.], v.LNCS4849, p.2–14, 2007.

SUTCLIFFE, A.; SAWYER, P. Requirements elicitation: towards the unknown unknowns. **2013 21st IEEE International Requirements Engineering Conference (RE)**, [S.l.], p.92–104, July 2013.

SUZUKI, S.; ABE, K. Topological structural analysis of digitized binary images by border following. **Computer Vision, Graphics, and Image Processing**, [S.l.], v.29, p.396, 1985.

TALLABACI, G. **Identification for Adaptive Software Systems - A Case Study**. 2012. 62p. MsC — University of Trento - Italy.

TALLABACI, G.; SOUZA, V. E. S. Engineering Adaptation with Zanshin: an experience report. In: INTERNATIONAL SYMPOSIUM ON SOFTWARE ENGINEERING FOR ADAPTIVE AND SELF-MANAGING SYSTEMS, 8. **Proceedings. . .** [S.l.: s.n.], 2013.

TIELLA, R.; VILLAFIORITA, A.; TOMASI, S. FSMC+, a tool for the generation of Java code from statecharts. In: PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA, 5. **Proceedings. . .** [S.l.: s.n.], 2007. p.93–102.

VILELA, J.; CASTRO, J.; PIMENTEL, J.; LIMA, P. On the behaviour of context-sensitive systems. **XVIII Ibero-American Conference on Software Engineering - CIBSE 2015**, [S.l.], p.10, 2015.

VILELA, J.; CASTRO, J.; PIMENTEL, J.; SOARES, M.; CAVALCANTI, P.; LUCENA, M. Deriving the behavior of context-sensitive systems from contextual goal models. **30th Annual ACM Symposium on Applied Computing - SAC 2015**, [S.l.], p.1397–1400, 2015.

WANG, Y.; MYLOPOULOS, J. Self-Repair through Reconfiguration: a requirements engineering approach. In: IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 2009. **Proceedings. . .** IEEE, 2009. p.257–268.

WANG, Y.; ZHANG, Y.; SHEU, P. C.; LI, X.; GUO, H. The Formal Design Model of an Automatic Teller Machine (ATM). **International Journal of Software Science and Computational Intelligence**, [S.l.], v.2, n.1, p.102–131, Jan. 2010.

WEYNS, D.; Usman Iftikhar, M.; SODERLUND, J. Do external feedback loops improve the design of self-adaptive systems? A controlled experiment. **2013 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)**, [S.l.], p.3–12, May 2013.

WHITTLE, J.; SAWYER, P.; BENCOMO, N.; CHENG, B. H. C.; BRUEL, J.-M. Relax: a language to address uncertainty in self-adaptive systems requirement. **Requirements Engineering Journal**, [S.l.], v.15, n.2, p.177–196, 2010.

WHITTLE, J.; SCHUMANN, J. Generating statechart designs from scenarios. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. ICSE 2000, 2000. **Proceedings. . .** ACM, 2000. n.650, p.314–323.

WOHLIN, C.; RUNESON, P.; HöST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLéN, A. **Experimentation in software engineering**. [S.l.]: Springer, 2012.

YU, E. Towards modelling and reasoning support for early-phase requirements engineering. **Proceedings of ISRE '97: 3rd IEEE International Symposium on Requirements Engineering**, [S.l.], p.226–235, 1997.

YU, E.; GIORGINI, P.; MAIDEN, N.; MYLOPOULOS, J. **Social modeling for requirements engineering**. [S.l.]: Mit Press, 2011.

YU, E.; MYLOPOULOS, J. Why Goal-Oriented Requirements Engineering. In: INTERNATIONAL WORKSHOP ON REQUIREMENTS ENGINEERING: FOUNDATIONS OF SOFTWARE QUALITY - REFSQ'98, 4., Pisa, Italy. **. . .** [S.l.: s.n.], 1998. p.15–22.

YU, E. S.; MYLOPOULOS, J. Understanding "why" in software process modelling, analysis, and design. **Proceedings of 16th International Conference on Software Engineering**, [S.l.], p.159–168, 1994.

YU, Y.; do Prado Leite, J. C. S.; LAPOUCHNIAN, A.; MYLOPOULOS, J. Configuring features with stakeholder goals. In: ACM SYMPOSIUM ON APPLIED COMPUTING - SAC '08, 2008. **Proceedings. . .** ACM Press, 2008. p.645–649.

YU, Y.; LAPOUCHNIAN, A.; LIASKOS, S.; MYLOPOULOS, J.; LEITE, J. C. S. P. From Goals to High-Variability Software Design. In: FOUNDATIONS OF INTELLIGENT SYSTEMS. **...** [S.l.: s.n.], 2008. v.4994/2008, p.1–16.

YU, Y.; MYLOPOULOS, J.; LAPOUCHNIAN, A.; LIASKOS, S.; LEITE, J. C. S. P. **From Stakeholder Goals to High-Variability Software Design**. [S.l.: s.n.], 2005.

ZHANG, Q.; ZHANI, M. F.; ZHANG, S.; ZHU, Q.; BOUTABA, R.; HELLERSTEIN, J. L. Dynamic energy-aware capacity provisioning for cloud computing environments. **Proceedings of the 9th international conference on Autonomic computing - ICAC '12**, [S.l.], p.145–154, 2012.

Appendix

A

Example of iterative and incremental design with MULAS

This appendix shows a hypothetical example of the MULAS design process being used iteratively and incrementally for designing a meeting scheduler system.

Figure A.1: Beginning of the meeting scheduler model, with only three goals: *Schedule Meeting*, *Characterize Meeting*, and *Define Schedule*.

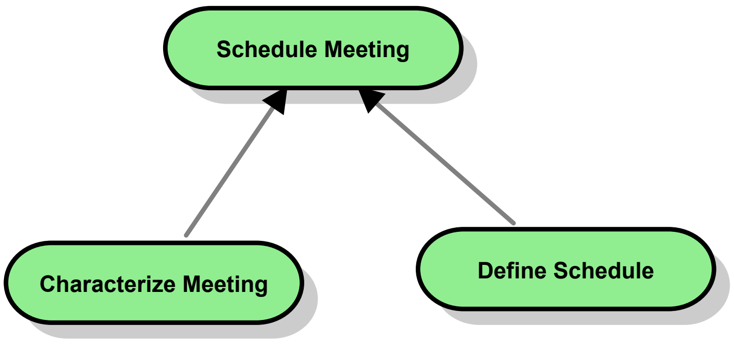


Figure A.2: After further requirements elicitation, two new tasks and a new goal were added to the model: *Schedule Manually*, *Schedule Automatically*, and *Collect Timetables*.

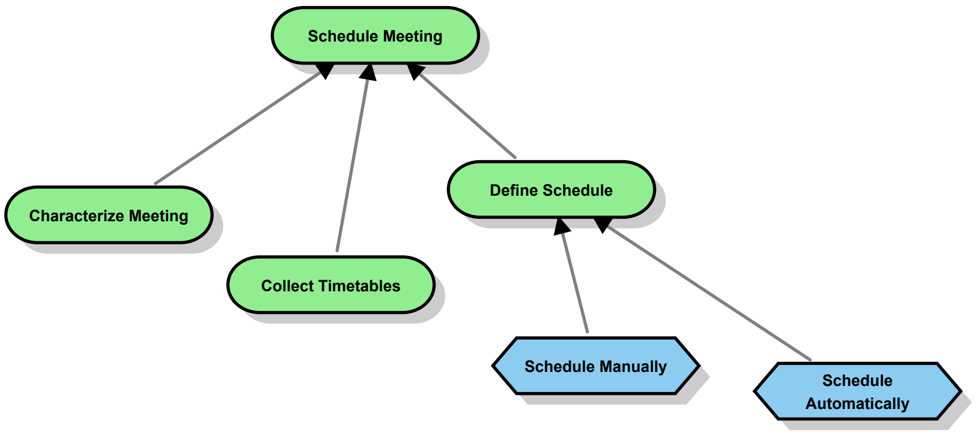


Figure A.3: Considering the desired behavior of the system, it was decided to start with *Characterize Meeting (g2)*, then *Collect Timetables (g6)*, and finally *Define Schedule (g3)*, which may be performed either manually (t4) or automatically (t5). This behavior is represented by the flow expressions on top of *g1* and *g3*. The resulting statechart is presented in the next figure.

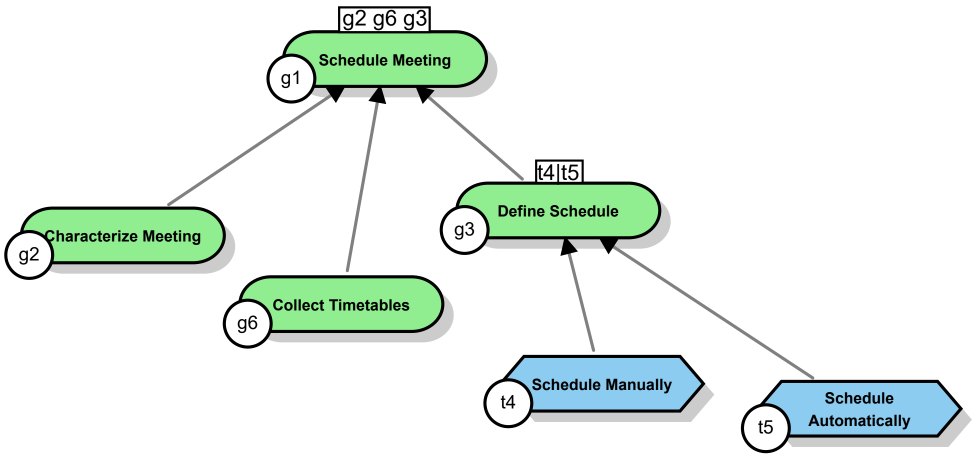


Figure A.4: Initial statechart of the meeting scheduler system, based on the design goal model from the previous figure.

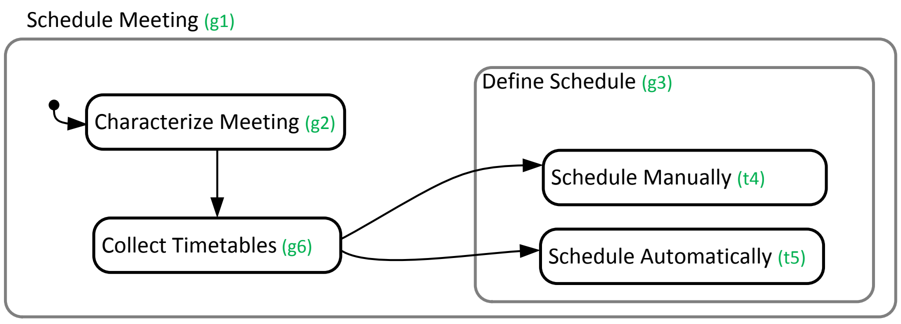


Figure A.5: Continuing on, the following design decisions were made: to provide two algorithms for the automatic scheduling (brute force and heuristics-based), as well as to *Use Web Services*.

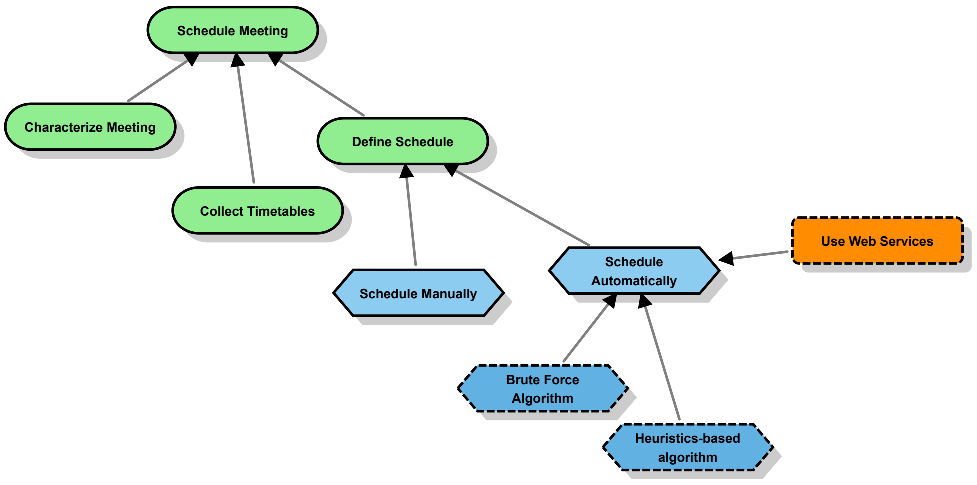


Figure A.6: Running automatic analysis with the GATO tool, it was identified that the following goals were not refined: *Characterize Meeting (g2)* and *Collect Timetables (g6)*. It was also identified that the following tasks were not included in the flow expressions: *Brute Force Algorithm (dt7)* and *Heuristics-based algorithm (dt8)*.

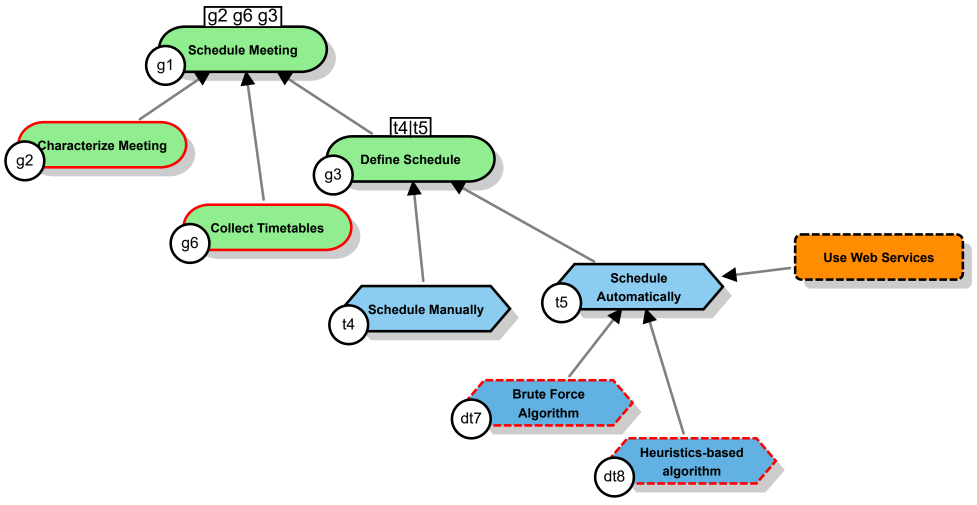


Figure A.7: Based on the previous feedback, the *Characterize Meeting* and *Collect Timetables* goals were refined with additional tasks: *Define Date Range, Define Participants*, and *Collect by Email*.

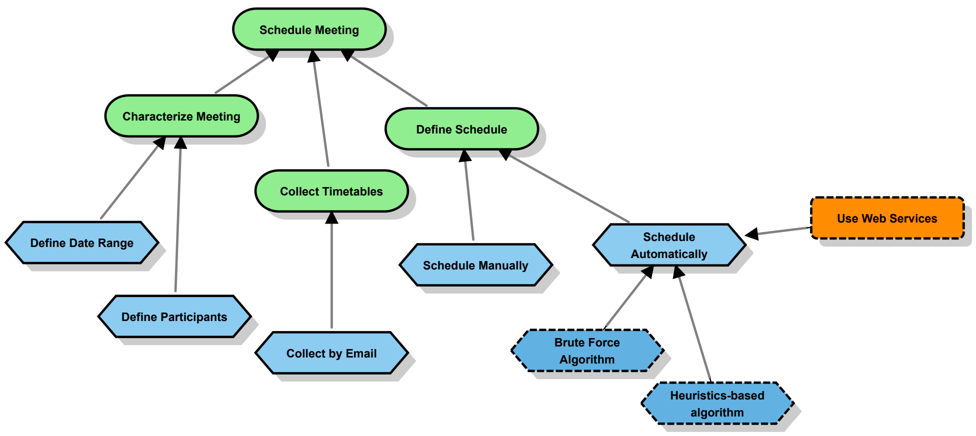


Figure A.8: The new tasks were included in the flow expressions: *t10, t11, t12, dt7,* and *dt8*.

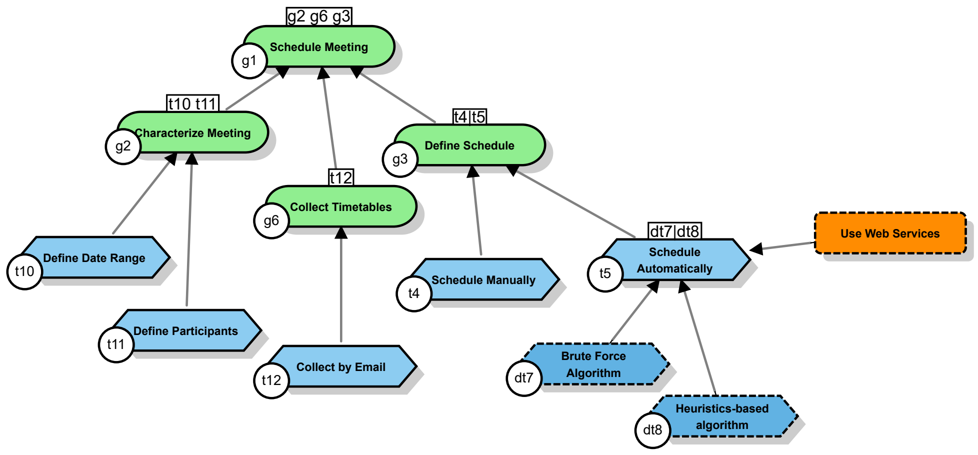


Figure A.9: This is the statechart generated based on the design goal model shown in the previous figure.

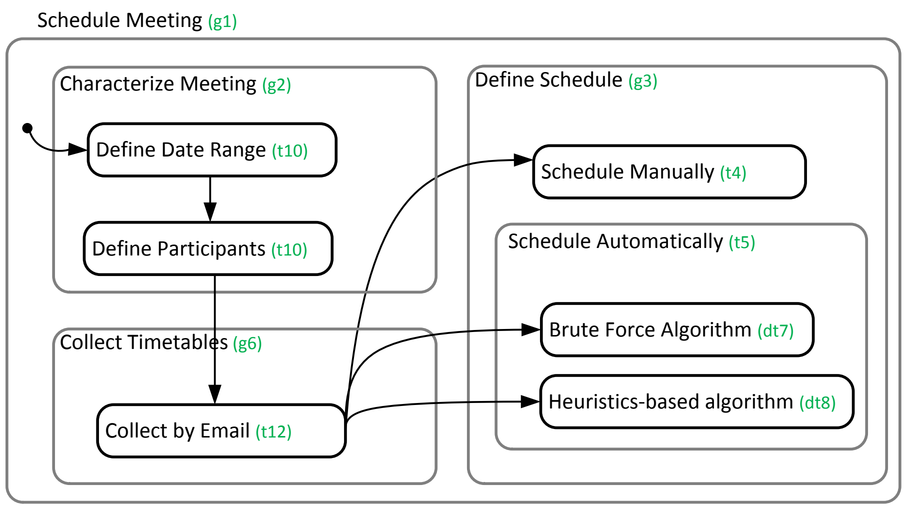

Schedule Meeting (g1)
Characterize Meeting (g2)
Define Date Range (t10)
Define Participants (t10)
Define Schedule (g3)
Schedule Manually (t4)
Schedule Automatically (t5)
Collect Timetables (g6)
Collect by Email (t12)
Brute Force Algorithm (dt7)
Heuristics-based algorithm (dt8)

Figure A.10: Unhappy with the behavior expressed in the previous figure, the designer modified the flow expression on *g1*. The resulting statechart is shown on the following figure.

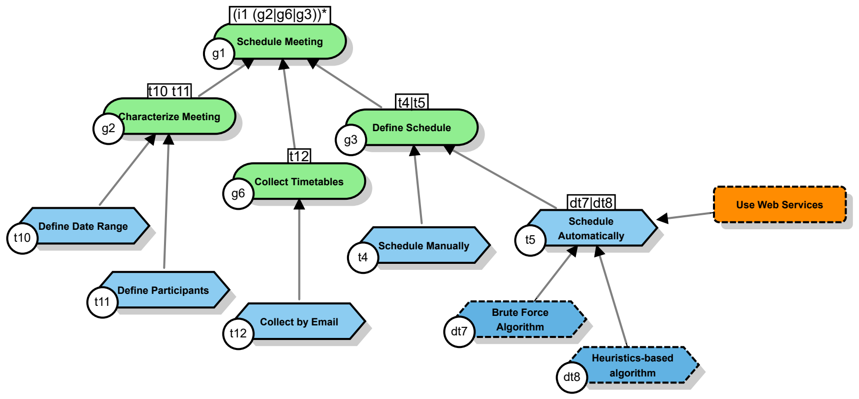


Figure A.11: Statechart representing the behavior of the meeting scheduler system, according to the flow expressions defined in the previous figure.

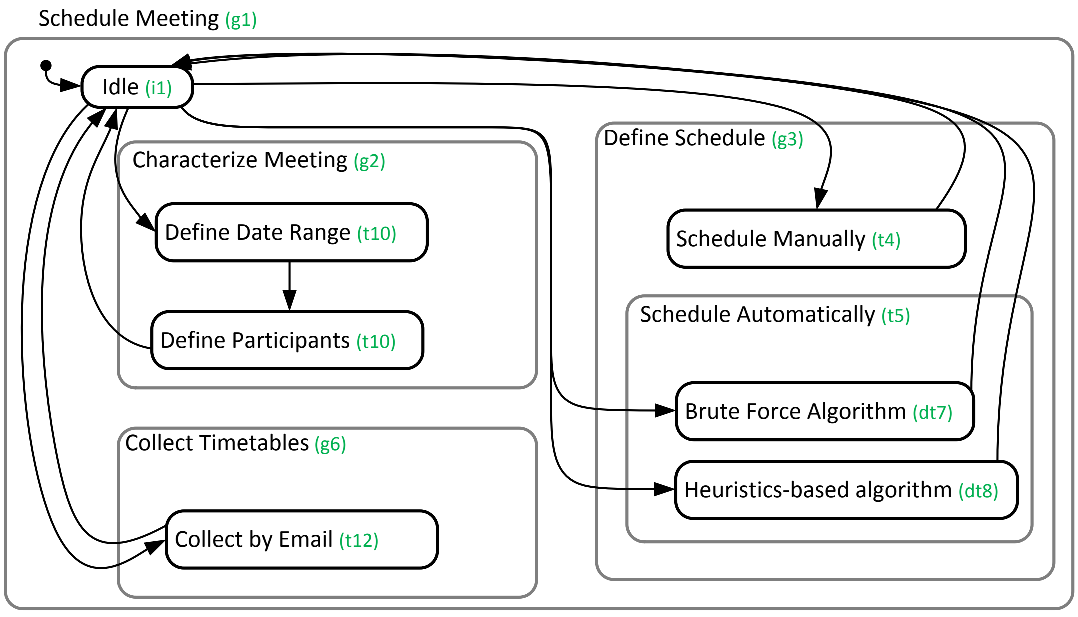
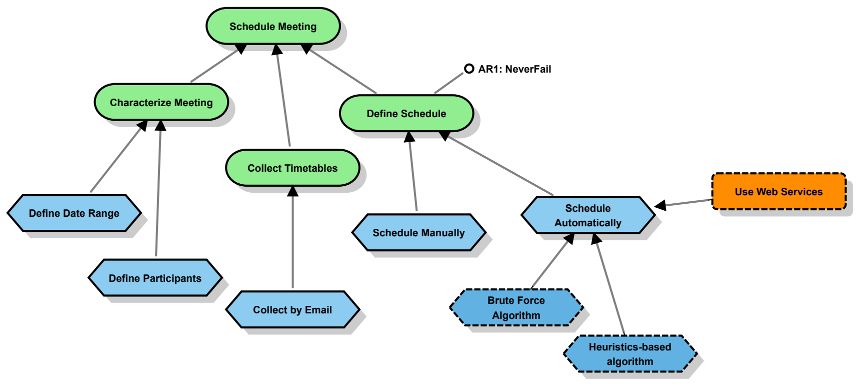


Figure A.12: An awareness requirement was included in the model, expressing that the *Define Schedule* goal should never fail.



Figures A.1 to A.12 show the beginning steps of the creation of a meeting scheduler system with the MULAS design process, illustrating the possibility of enacting it incrementally and iteratively. For the full models of the meeting scheduler system, please refer to Chapter 4.

B
ATM Adaptation Scenarios - Complete Logs

The following pages present the complete log of the execution of different adaptation scenarios, resulting from a simulation of the Automatic Teller Machine (ATM) system. For more information regarding these adaptation scenarios, see Section 6.1.1.4 (Step 4).

B.1 Log of the first adaptation scenario

Complete execution log of the **first** adaptation scenario of the ATM adaptation simulation, described on Section 6.1.1.4 (Step 4). This log shows all information logged by Zanshin's component when performing the first adaptation scenario, related to *AR1 — NeverFail(Detect Cash Amount)*.

```
2015-01-31 11:57:16 [zanshin.controller       ] INFO: Successfully created a new user session for
    target system atm: 1.422.716.236.486
2015-01-31 11:57:16 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.716.236.486: TDetectCashAm.START()
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Requirement started: TDetectCashAm
    (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@1fe5cec (refinementType: and) (time:
    null, state: undefined) (startTime: null))
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Requirement started: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Requirement started: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 11:57:16 [zanshin.monitoring       ] INFO: Processing method call: start / TDetectCashAm
2015-01-31 11:57:16 [zanshin.monitoring       ] INFO: Processing method call: start / GStartATM
2015-01-31 11:57:16 [zanshin.monitoring       ] INFO: Processing method call: start / GProvideATM
2015-01-31 11:57:16 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.716.236.486: TDetectCashAm.FAIL()
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Requirement failed: TDetectCashAm
    (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@1fe5cec (refinementType: and) (time:
    null, state: started) (startTime: null))
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Requirement ended: TDetectCashAm
    (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@1fe5cec (refinementType: and) (time:
    null, state: failed) (startTime: null))
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Requirement failed: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 11:57:16 [zanshin.monitoring       ] INFO: Processing method call: fail / TDetectCashAm
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Requirement ended: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:16 [zanshin.monitoring       ] INFO: Requirement TDetectCashAm has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 11:57:16 [zanshin.adaptation       ] INFO: Processing state change: AR1 (ref.
    TDetectCashAm) -> failed
2015-01-31 11:57:16 [zanshin.adaptation       ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Created new session for AR1
2015-01-31 11:57:16 [zanshin.adaptation       ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523) The
    problem has not yet been solved...
2015-01-31 11:57:16 [zanshin.core             ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Strategy RetryStrategy is applicable.
2015-01-31 11:57:16 [zanshin.adaptation       ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Selected adaptation strategy: RetryStrategy
2015-01-31 11:57:16 [zanshin.core             ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Applying strategy RetryStrategy(true; 5000)...
2015-01-31 11:57:16 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: copy-data(iTDetectCashAm, iTDetectCashAm)
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Replacing requirement instances of class
    TDetectCashAm (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@1fe5cec (refinementType:
    and) (time: null, state: failed) (startTime: null) ->
    it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@f6a6d7 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: The status of GStartATM has been reset to
    undefined
2015-01-31 11:57:16 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: terminate(iTDetectCashAm)
2015-01-31 11:57:16 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: rollback(iTDetectCashAm)
2015-01-31 11:57:16 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: wait(5.000)
2015-01-31 11:57:16 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: initiate(iTDetectCashAm)
2015-01-31 11:57:16 [zanshin.adaptation       ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523) The
    problem has not yet been solved...
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Replacing requirement instances of class AR1
    (it.unitn.disi.zanshin.model.atm.impl.AR1Impl@f5181e (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR1Impl@14c1f00
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: The status of GProvideATM has been reset to
    undefined
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass ReconfigurationStrategy. Make sure this is on purpose...
2015-01-31 11:57:16 [zanshin.core             ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 11:57:16 [zanshin.monitoring       ] INFO: Processing method call: end / TDetectCashAm
2015-01-31 11:57:16 [zanshin.monitoring       ] INFO: Processing method call: fail / GStartATM
2015-01-31 11:57:16 [zanshin.monitoring       ] INFO: Processing method call: end / GStartATM
```

```
2015-01-31 11:57:16 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 11:57:16 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
2015-01-31 11:57:23 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
   session atm/1.422.716.236.486: TDetectCashAm.START()
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Requirement started: TDetectCashAm
   (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@f6a6d7 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Requirement started: GStartATM
   (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Requirement started: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Processing method call: start / TDetectCashAm
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Processing method call: start / GStartATM
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Processing method call: start / GProvideATM
2015-01-31 11:57:23 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
   session atm/1.422.716.236.486: TDetectCashAm.FAIL()
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Requirement failed: TDetectCashAm
   (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@f6a6d7 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Requirement ended: TDetectCashAm
   (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@f6a6d7 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Requirement failed: GStartATM
   (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Requirement ended: GStartATM
   (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Processing method call: fail / TDetectCashAm
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Requirement TDetectCashAm has 1 AwReqs
   referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 11:57:23 [zanshin.adaptation      ] INFO: Processing state change: AR1 (ref.
   TDetectCashAm) -> failed
2015-01-31 11:57:23 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
   Retrieved existing session for AR1, one event already in the timeline
2015-01-31 11:57:23 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523) The
   problem has not yet been solved...
2015-01-31 11:57:23 [zanshin.core            ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
   Strategy RetryStrategy is applicable.
2015-01-31 11:57:23 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
   Selected adaptation strategy: RetryStrategy
2015-01-31 11:57:23 [zanshin.core            ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
   Applying strategy RetryStrategy(true; 5000)...
2015-01-31 11:57:23 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: copy-data(iTDetectCashAm, iTDetectCashAm)
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Replacing requirement instances of class
   TDetectCashAm (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@f6a6d7 (refinementType:
   and) (time: null, state: failed) (startTime: null) ->
   it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@3f4126 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: The status of GStartATM has been reset to
   undefined
2015-01-31 11:57:23 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: terminate(iTDetectCashAm)
2015-01-31 11:57:23 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: rollback(iTDetectCashAm)
2015-01-31 11:57:23 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: wait(5.000)
2015-01-31 11:57:23 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: initiate(iTDetectCashAm)
2015-01-31 11:57:23 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523) The
   problem has not yet been solved...
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Replacing requirement instances of class AR1
   (it.unitn.disi.zanshin.model.atm.impl.AR1Impl@14c1f00 (refinementType: and) (time: null, state:
   failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR1Impl@d5c83f
   (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: The status of GProvideATM has been reset to
   undefined
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Method
   AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
   by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Method
   AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
   by the subclass ReconfigurationStrategy. Make sure this is on purpose...
2015-01-31 11:57:23 [zanshin.core            ] DEBUG: Method
   AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
   by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Processing method call: end / TDetectCashAm
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Processing method call: fail / GStartATM
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Processing method call: end / GStartATM
```

```
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 11:57:23 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
2015-01-31 11:57:30 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.716.236.486: TDetectCashAm.START()
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Requirement started: TDetectCashAm
    (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@3f4126 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Requirement started: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Requirement started: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Processing method call: start / TDetectCashAm
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Processing method call: start / GStartATM
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Processing method call: start / GProvideATM
2015-01-31 11:57:30 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.716.236.486: TDetectCashAm.FAIL()
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Requirement failed: TDetectCashAm
    (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@3f4126 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Requirement ended: TDetectCashAm
    (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@3f4126 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Requirement failed: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Requirement ended: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Processing method call: fail / TDetectCashAm
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Requirement TDetectCashAm has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 11:57:30 [zanshin.adaptation      ] INFO: Processing state change: AR1 (ref.
    TDetectCashAm) -> failed
2015-01-31 11:57:30 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Retrieved existing session for AR1, 2 events already in the timeline
2015-01-31 11:57:30 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523) The
    problem has not yet been solved...
2015-01-31 11:57:30 [zanshin.core            ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Strategy RetryStrategy is not applicable because it has been applied at least 2 time(s) this
    session.
2015-01-31 11:57:30 [zanshin.core            ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Strategy ReconfigurationStrategy is applicable.
2015-01-31 11:57:30 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Selected adaptation strategy: ReconfigurationStrategy
2015-01-31 11:57:30 [zanshin.core            ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Applying strategy ReconfigurationStrategy(qualia; class-level)...
2015-01-31 11:57:30 [zanshin.adaptation.qualia] DEBUG: Creating a default algorithm...
2015-01-31 11:57:30 [zanshin.adaptation.qualia] WARNING: procIds null? true
2015-01-31 11:57:30 [zanshin.adaptation.qualia] INFO: Parameters chosen: [CaD]
2015-01-31 11:57:30 [zanshin.adaptation.qualia] INFO: Values to inc/decrement in the chosen
    parameters: [1.00000]
2015-01-31 11:57:30 [zanshin.adaptation.qualia] INFO: Produced new configuration with 1 changed
    parameter(s)
2015-01-31 11:57:30 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: apply-config(AtmGoalModel,
    it.unitn.disi.zanshin.model.gore.impl.ConfigurationImpl@93dba1, class-level)
2015-01-31 11:57:30 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523) The
    problem has not yet been solved...
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Replacing requirement instances of class AR1
    (it.unitn.disi.zanshin.model.atm.impl.AR1Impl@d5c83f (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR1Impl@146bc20
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass ReconfigurationStrategy. Make sure this is on purpose...
2015-01-31 11:57:30 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Processing method call: end / TDetectCashAm
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Processing method call: fail / GStartATM
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Processing method call: end / GStartATM
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 11:57:30 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
2015-01-31 11:57:37 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.716.236.486: TDetectCashAm.START()
```

```
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Requirement started: TDetectCashAm
    (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@3f4126 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:37 [zanshin.monitoring      ] INFO: Processing method call: start / TDetectCashAm
2015-01-31 11:57:37 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.716.236.486: TDetectCashAm.FAIL()
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Requirement failed: TDetectCashAm
    (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@3f4126 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Requirement ended: TDetectCashAm
    (it.unitn.disi.zanshin.model.atm.impl.TDetectCashAmImpl@3f4126 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Requirement failed: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Requirement ended: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@15e6a12 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@185d6e8 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 11:57:37 [zanshin.monitoring      ] INFO: Processing method call: fail / TDetectCashAm
2015-01-31 11:57:37 [zanshin.monitoring      ] INFO: Requirement TDetectCashAm has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 11:57:37 [zanshin.adaptation      ] INFO: Processing state change: AR1 (ref.
    TDetectCashAm) -> failed
2015-01-31 11:57:37 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Retrieved existing session for AR1, 3 events already in the timeline
2015-01-31 11:57:37 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523) The
    problem has not yet been solved...
2015-01-31 11:57:37 [zanshin.core            ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Strategy RetryStrategy is not applicable because it has been applied at least 2 time(s) this
    session.
2015-01-31 11:57:37 [zanshin.core            ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Strategy ReconfigurationStrategy is not applicable because it has been applied at least 1 time(s)
    this session.
2015-01-31 11:57:37 [zanshin.core            ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Strategy AbortStrategy is applicable.
2015-01-31 11:57:37 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Selected adaptation strategy: AbortStrategy
2015-01-31 11:57:37 [zanshin.core            ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523)
    Applying strategy AbortStrategy...
2015-01-31 11:57:37 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: abort(iAR1)
2015-01-31 11:57:37 [zanshin.adaptation      ] INFO: (Session: AR1 / 2015-01-31 11:57:16.523) The
    problem has been solved or there is nothing else to try. Adaptation session will be terminated.
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Replacing requirement instances of class AR1
    (it.unitn.disi.zanshin.model.atm.impl.AR1Impl@146bc20 (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR1Impl@1778a52
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass ReconfigurationStrategy. Make sure this is on purpose...
2015-01-31 11:57:37 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 11:57:37 [zanshin.monitoring      ] INFO: Processing method call: end / TDetectCashAm
2015-01-31 11:57:37 [zanshin.monitoring      ] INFO: Processing method call: fail / GStartATM
2015-01-31 11:57:37 [zanshin.monitoring      ] INFO: Processing method call: end / GStartATM
2015-01-31 11:57:37 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 11:57:37 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
```

B.2 Log of the second adaptation scenario

Complete execution log of the **second** adaptation scenario of the ATM adaptation simulation, described on Section 6.1.1.4 (Step 4). This log shows all information logged by Zanshin's component when performing the second adaptation scenario, related to *AR2 — NeverFail(Setup Connection to Bank)*.

```
2015-01-31 12:29:17 [zanshin.controller      ] INFO: Successfully created a new user session for
    target system atm: 1.422.718.157.453
2015-01-31 12:29:17 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.157.453: TSetUpConnect.START()
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Requirement started: TSetUpConnect
    (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@5a0660 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Requirement started: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Requirement started: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Processing method call: start / TSetUpConnect
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Processing method call: start / GStartATM
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:29:17 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.157.453: TSetUpConnect.FAIL()
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Requirement failed: TSetUpConnect
    (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@5a0660 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Requirement ended: TSetUpConnect
    (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@5a0660 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Requirement failed: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Processing method call: fail / TSetUpConnect
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Requirement ended: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Requirement TSetUpConnect has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:29:17 [zanshin.adaptation      ] INFO: Processing state change: AR2 (ref.
    TSetUpConnect) -> failed
2015-01-31 12:29:17 [zanshin.adaptation      ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
    Created new session for AR2
2015-01-31 12:29:17 [zanshin.adaptation      ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490) The
    problem has not yet been solved...
2015-01-31 12:29:17 [zanshin.core            ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
    Strategy RetryStrategy is applicable.
2015-01-31 12:29:17 [zanshin.adaptation      ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
    Selected adaptation strategy: RetryStrategy
2015-01-31 12:29:17 [zanshin.core            ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
    Applying strategy RetryStrategy(true; 10000)...
2015-01-31 12:29:17 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: copy-data(iTSetUpConnect, iTSetUpConnect)
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Replacing requirement instances of class
    TSetUpConnect (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@5a0660 (refinementType:
    and) (time: null, state: failed) (startTime: null) ->
    it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@16ff60b (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: The status of GStartATM has been reset to
    undefined
2015-01-31 12:29:17 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: terminate(iTSetUpConnect)
2015-01-31 12:29:17 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: rollback(iTSetUpConnect)
2015-01-31 12:29:17 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: wait(10.000)
2015-01-31 12:29:17 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: initiate(iTSetUpConnect)
2015-01-31 12:29:17 [zanshin.adaptation      ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490) The
    problem has not yet been solved...
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Replacing requirement instances of class AR2
    (it.unitn.disi.zanshin.model.atm.impl.AR2Impl@a125fc (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR2Impl@19b07f9
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: The status of GProvideATM has been reset to
    undefined
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:29:17 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Processing method call: end / TSetUpConnect
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Processing method call: fail / GStartATM
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Processing method call: end / GStartATM
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:29:17 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
```

```
2015-01-31 12:29:24 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.157.453: TSetUpConnect.START()
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Requirement started: TSetUpConnect
    (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@16ff60b (refinementType: and) (time:
    null, state: undefined) (startTime: null))
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Requirement started: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Requirement started: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Processing method call: start / TSetUpConnect
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Processing method call: start / GStartATM
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:29:24 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.157.453: TSetUpConnect.FAIL()
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Requirement failed: TSetUpConnect
    (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@16ff60b (refinementType: and) (time:
    null, state: started) (startTime: null))
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Requirement ended: TSetUpConnect
    (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@16ff60b (refinementType: and) (time:
    null, state: failed) (startTime: null))
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Requirement failed: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Requirement ended: GStartATM
    (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Processing method call: fail / TSetUpConnect
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Requirement TSetUpConnect has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:29:24 [zanshin.adaptation       ] INFO: Processing state change: AR2 (ref.
    TSetUpConnect) -> failed
2015-01-31 12:29:24 [zanshin.adaptation       ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
    Retrieved existing session for AR2, one event already in the timeline
2015-01-31 12:29:24 [zanshin.adaptation       ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490) The
    problem has not yet been solved...
2015-01-31 12:29:24 [zanshin.core             ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
    Strategy RetryStrategy is applicable.
2015-01-31 12:29:24 [zanshin.adaptation       ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
    Selected adaptation strategy: RetryStrategy
2015-01-31 12:29:24 [zanshin.core             ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
    Applying strategy RetryStrategy(true; 10000)...
2015-01-31 12:29:24 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: copy-data(iTSetUpConnect, iTSetUpConnect)
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Replacing requirement instances of class
    TSetUpConnect (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@16ff60b (refinementType:
    and) (time: null, state: failed) (startTime: null) ->
    it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@13d176a (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: The status of GStartATM has been reset to
    undefined
2015-01-31 12:29:24 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: terminate(iTSetUpConnect)
2015-01-31 12:29:24 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: rollback(iTSetUpConnect)
2015-01-31 12:29:24 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: wait(10.000)
2015-01-31 12:29:24 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: initiate(iTSetUpConnect)
2015-01-31 12:29:24 [zanshin.adaptation       ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490) The
    problem has not yet been solved...
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Replacing requirement instances of class AR2
    (it.unitn.disi.zanshin.model.atm.impl.AR2Impl@19b07f9 (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR2Impl@14eae38
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: The status of GProvideATM has been reset to
    undefined
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:29:24 [zanshin.core             ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Processing method call: end / TSetUpConnect
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Processing method call: fail / GStartATM
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Processing method call: end / GStartATM
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:29:24 [zanshin.monitoring       ] INFO: Processing method call: end / GProvideATM
2015-01-31 12:29:31 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.157.453: TSetUpConnect.START()
```

```
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Requirement started: TSetUpConnect
   (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@13d176a (refinementType: and) (time:
   null, state: undefined) (startTime: null))
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Requirement started: GStartATM
   (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Requirement started: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Processing method call: start / TSetUpConnect
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Processing method call: start / GStartATM
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:29:31 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
   session atm/1.422.718.157.453: TSetUpConnect.FAIL()
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Requirement failed: TSetUpConnect
   (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@13d176a (refinementType: and) (time:
   null, state: started) (startTime: null))
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Requirement ended: TSetUpConnect
   (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@13d176a (refinementType: and) (time:
   null, state: failed) (startTime: null))
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Requirement failed: GStartATM
   (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Requirement ended: GStartATM
   (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Requirement failed: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Requirement ended: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Processing method call: fail / TSetUpConnect
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Requirement TSetUpConnect has 1 AwReqs
   referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:29:31 [zanshin.adaptation       ] INFO: Processing state change: AR2 (ref.
   TSetUpConnect) -> failed
2015-01-31 12:29:31 [zanshin.adaptation       ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
   Retrieved existing session for AR2, 2 events already in the timeline
2015-01-31 12:29:31 [zanshin.adaptation       ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490) The
   problem has not yet been solved...
2015-01-31 12:29:31 [zanshin.core             ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
   Strategy RetryStrategy is applicable.
2015-01-31 12:29:31 [zanshin.adaptation       ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
   Selected adaptation strategy: RetryStrategy
2015-01-31 12:29:31 [zanshin.core             ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
   Applying strategy RetryStrategy(true; 10000)...
2015-01-31 12:29:31 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
   instruction: copy-data(iTSetUpConnect, iTSetUpConnect)
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Replacing requirement instances of class
   TSetUpConnect (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@13d176a (refinementType:
   and) (time: null, state: failed) (startTime: null) ->
   it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@1c7a0ef (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: The status of GStartATM has been reset to
   undefined
2015-01-31 12:29:31 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
   instruction: terminate(iTSetUpConnect)
2015-01-31 12:29:31 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
   instruction: rollback(iTSetUpConnect)
2015-01-31 12:29:31 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
   instruction: wait(10.000)
2015-01-31 12:29:31 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
   instruction: initiate(iTSetUpConnect)
2015-01-31 12:29:31 [zanshin.adaptation       ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490) The
   problem has not yet been solved...
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Replacing requirement instances of class AR2
   (it.unitn.disi.zanshin.model.atm.impl.AR2Impl@14eae38 (refinementType: and) (time: null, state:
   failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR2Impl@14765f1
   (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: The status of GProvideATM has been reset to
   undefined
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Method
   AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
   by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:29:31 [zanshin.core             ] DEBUG: Method
   AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
   by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Processing method call: end / TSetUpConnect
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Processing method call: fail / GStartATM
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Processing method call: end / GStartATM
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:29:31 [zanshin.monitoring       ] INFO: Processing method call: end / GProvideATM
2015-01-31 12:29:38 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
   session atm/1.422.718.157.453: TSetUpConnect.START()
2015-01-31 12:29:38 [zanshin.core             ] DEBUG: Requirement started: TSetUpConnect
   (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@1c7a0ef (refinementType: and) (time:
   null, state: undefined) (startTime: null))
```

```
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Requirement started: GStartATM
   (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Requirement started: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Processing method call: start / TSetUpConnect
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Processing method call: start / GStartATM
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:29:38 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
   session atm/1.422.718.157.453: TSetUpConnect.FAIL()
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Requirement failed: TSetUpConnect
   (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@1c7a0ef (refinementType: and) (time:
   null, state: started) (startTime: null))
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Requirement ended: TSetUpConnect
   (it.unitn.disi.zanshin.model.atm.impl.TSetUpConnectImpl@1c7a0ef (refinementType: and) (time:
   null, state: failed) (startTime: null))
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Requirement failed: GStartATM
   (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Requirement ended: GStartATM
   (it.unitn.disi.zanshin.model.atm.impl.GStartATMImpl@dbb822 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@115c1a5 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Processing method call: fail / TSetUpConnect
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Requirement TSetUpConnect has 1 AwReqs
   referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:29:38 [zanshin.adaptation      ] INFO: Processing state change: AR2 (ref.
   TSetUpConnect) -> failed
2015-01-31 12:29:38 [zanshin.adaptation      ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
   Retrieved existing session for AR2, 3 events already in the timeline
2015-01-31 12:29:38 [zanshin.adaptation      ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490) The
   problem has not yet been solved...
2015-01-31 12:29:38 [zanshin.core            ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
   Strategy RetryStrategy is not applicable because it has been applied at least 3 time(s) this
   session.
2015-01-31 12:29:38 [zanshin.core            ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
   Strategy AbortStrategy is applicable.
2015-01-31 12:29:38 [zanshin.adaptation      ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
   Selected adaptation strategy: AbortStrategy
2015-01-31 12:29:38 [zanshin.core            ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490)
   Applying strategy AbortStrategy...
2015-01-31 12:29:38 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: abort(iAR2)
2015-01-31 12:29:38 [zanshin.adaptation      ] INFO: (Session: AR2 / 2015-01-31 12:29:17.490) The
   problem has been solved or there is nothing else to try. Adaptation session will be terminated.
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Replacing requirement instances of class AR2
   (it.unitn.disi.zanshin.model.atm.impl.AR2Impl@14765f1 (refinementType: and) (time: null, state:
   failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR2Impl@2b59d1
   (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Method
   AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
   by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:29:38 [zanshin.core            ] DEBUG: Method
   AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
   by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Processing method call: end / TSetUpConnect
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Processing method call: fail / GStartATM
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Processing method call: end / GStartATM
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:29:38 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
```

B.3 Log of the third adaptation scenario

Complete execution log of the **third** adaptation scenario of the ATM adaptation simulation, described on Section 6.1.1.4 (Step 4). This log shows all information logged by Zanshin's component when performing the third adaptation scenario, related to *AR4 — NeverFail(Confirm Transaction)*.

```
2015-01-31 12:38:44 [zanshin.controller      ] INFO: Successfully created a new user session for
    target system atm: 1.422.718.724.215
2015-01-31 12:38:44 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.START()
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement started: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@e8fb61 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement started: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: undefined) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement started: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement started: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:44 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.FAIL()
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement failed: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@e8fb61 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement ended: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@e8fb61 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement failed: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: started) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement ended: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: failed) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement failed: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement ended: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: start / GConfirmTrans
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: start / GConductTrans
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: start / GServeCust
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: fail / GConfirmTrans
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Requirement GConfirmTrans has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:38:44 [zanshin.adaptation      ] INFO: Processing state change: AR4 (ref.
    GConfirmTrans) -> failed
2015-01-31 12:38:44 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Created new session for AR4
2015-01-31 12:38:44 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:38:44 [zanshin.core            ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Strategy RetryStrategy is applicable.
2015-01-31 12:38:44 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Selected adaptation strategy: RetryStrategy
2015-01-31 12:38:44 [zanshin.core            ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Applying strategy RetryStrategy(true; 2000)...
2015-01-31 12:38:44 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: copy-data(iGConfirmTrans, iGConfirmTrans)
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Replacing requirement instances of class
    GConfirmTrans (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@e8fb61 (refinementType:
    and) (time: null, state: failed) (startTime: null) ->
    it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@2d5b6f (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: The status of GConductTrans has been reset to
    undefined
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: The status of GServeCust has been reset to
    undefined
2015-01-31 12:38:44 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: terminate(iGConfirmTrans)
2015-01-31 12:38:44 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: rollback(iGConfirmTrans)
2015-01-31 12:38:44 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: wait(2.000)
2015-01-31 12:38:44 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: initiate(iGConfirmTrans)
2015-01-31 12:38:44 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Replacing requirement instances of class AR4
    (it.unitn.disi.zanshin.model.atm.impl.AR4Impl@1ae2e31 (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR4Impl@6cd0e8
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: The status of GProvideATM has been reset to
    undefined
```

```
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:38:44 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: end / GConfirmTrans
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: fail / GConductTrans
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: end / GConductTrans
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: fail / GServeCust
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: end / GServeCust
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:38:44 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
2015-01-31 12:38:51 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.START()
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement started: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@2d5b6f (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement started: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: undefined) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement started: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement started: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:51 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.FAIL()
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement failed: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@2d5b6f (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement ended: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@2d5b6f (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement failed: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: started) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement ended: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: failed) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement failed: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement ended: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:38:51 [zanshin.monitoring      ] INFO: Processing method call: start / GConfirmTrans
2015-01-31 12:38:51 [zanshin.monitoring      ] INFO: Processing method call: start / GConductTrans
2015-01-31 12:38:51 [zanshin.monitoring      ] INFO: Processing method call: start / GServeCust
2015-01-31 12:38:51 [zanshin.monitoring      ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:38:51 [zanshin.monitoring      ] INFO: Processing method call: fail / GConfirmTrans
2015-01-31 12:38:51 [zanshin.monitoring      ] INFO: Requirement GConfirmTrans has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:38:51 [zanshin.adaptation      ] INFO: Processing state change: AR4 (ref.
    GConfirmTrans) -> failed
2015-01-31 12:38:51 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Retrieved existing session for AR4, one event already in the timeline
2015-01-31 12:38:51 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:38:51 [zanshin.core            ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Strategy RetryStrategy is applicable.
2015-01-31 12:38:51 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Selected adaptation strategy: RetryStrategy
2015-01-31 12:38:51 [zanshin.core            ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Applying strategy RetryStrategy(true; 2000)...
2015-01-31 12:38:51 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: copy-data(iGConfirmTrans, iGConfirmTrans)
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: Replacing requirement instances of class
    GConfirmTrans (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@2d5b6f (refinementType:
    and) (time: null, state: failed) (startTime: null) ->
    it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@1ce21d8 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: The status of GConductTrans has been reset to
    undefined
2015-01-31 12:38:51 [zanshin.core            ] DEBUG: The status of GServeCust has been reset to
    undefined
2015-01-31 12:38:51 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: terminate(iGConfirmTrans)
2015-01-31 12:38:51 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: rollback(iGConfirmTrans)
```

```
2015-01-31 12:38:51 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: wait(2.000)
2015-01-31 12:38:51 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: initiate(iGConfirmTrans)
2015-01-31 12:38:51 [zanshin.adaptation       ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:38:51 [zanshin.core             ] DEBUG: Replacing requirement instances of class AR4
    (it.unitn.disi.zanshin.model.atm.impl.AR4Impl@6cd0e8 (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR4Impl@1d57e48
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:38:51 [zanshin.core             ] DEBUG: The status of GProvideATM has been reset to
    undefined
2015-01-31 12:38:51 [zanshin.core             ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:38:51 [zanshin.core             ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:38:51 [zanshin.monitoring       ] INFO: Processing method call: end / GConfirmTrans
2015-01-31 12:38:51 [zanshin.monitoring       ] INFO: Processing method call: fail / GConductTrans
2015-01-31 12:38:51 [zanshin.monitoring       ] INFO: Processing method call: end / GConductTrans
2015-01-31 12:38:51 [zanshin.monitoring       ] INFO: Processing method call: fail / GServeCust
2015-01-31 12:38:51 [zanshin.monitoring       ] INFO: Processing method call: end / GServeCust
2015-01-31 12:38:51 [zanshin.monitoring       ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:38:51 [zanshin.monitoring       ] INFO: Processing method call: end / GProvideATM
2015-01-31 12:38:58 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.START()
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement started: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@1ce21d8 (refinementType: and) (time:
    null, state: undefined) (startTime: null))
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement started: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: undefined) (startTime: null))
2015-01-31 12:38:58 [zanshin.monitoring       ] INFO: Processing method call: start / GConfirmTrans
2015-01-31 12:38:58 [zanshin.monitoring       ] INFO: Processing method call: start / GConductTrans
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement started: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:58 [zanshin.monitoring       ] INFO: Processing method call: start / GServeCust
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement started: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:38:58 [zanshin.monitoring       ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:38:58 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.FAIL()
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement failed: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@1ce21d8 (refinementType: and) (time:
    null, state: started) (startTime: null))
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement ended: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@1ce21d8 (refinementType: and) (time:
    null, state: failed) (startTime: null))
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement failed: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: started) (startTime: null))
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement ended: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: failed) (startTime: null))
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement failed: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement ended: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:38:58 [zanshin.monitoring       ] INFO: Processing method call: fail / GConfirmTrans
2015-01-31 12:38:58 [zanshin.monitoring       ] INFO: Requirement GConfirmTrans has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:38:58 [zanshin.adaptation       ] INFO: Processing state change: AR4 (ref.
    GConfirmTrans) -> failed
2015-01-31 12:38:58 [zanshin.adaptation       ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Retrieved existing session for AR4, 2 events already in the timeline
2015-01-31 12:38:58 [zanshin.adaptation       ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:38:58 [zanshin.core             ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Strategy RetryStrategy is applicable.
2015-01-31 12:38:58 [zanshin.adaptation       ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Selected adaptation strategy: RetryStrategy
2015-01-31 12:38:58 [zanshin.core             ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Applying strategy RetryStrategy(true; 2000)...
2015-01-31 12:38:58 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: copy-data(iGConfirmTrans, iGConfirmTrans)
2015-01-31 12:38:58 [zanshin.core             ] DEBUG: Replacing requirement instances of class
    GConfirmTrans (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@1ce21d8 (refinementType:
```

```
   and) (time: null, state: failed) (startTime: null) ->
   it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@6717c8 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 12:38:58 [zanshin.core            ] DEBUG: The status of GConductTrans has been reset to
   undefined
2015-01-31 12:38:58 [zanshin.core            ] DEBUG: The status of GServeCust has been reset to
   undefined
2015-01-31 12:38:58 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: terminate(iGConfirmTrans)
2015-01-31 12:38:58 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: rollback(iGConfirmTrans)
2015-01-31 12:38:58 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: wait(2.000)
2015-01-31 12:38:58 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
   instruction: initiate(iGConfirmTrans)
2015-01-31 12:38:58 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
   problem has not yet been solved...
2015-01-31 12:38:58 [zanshin.core            ] DEBUG: Replacing requirement instances of class AR4
   (it.unitn.disi.zanshin.model.atm.impl.AR4Impl@1d57e48 (refinementType: and) (time: null, state:
   failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR4Impl@1f2a26d
   (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:38:58 [zanshin.core            ] DEBUG: The status of GProvideATM has been reset to
   undefined
2015-01-31 12:38:58 [zanshin.core            ] DEBUG: Method
   AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
   by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:38:58 [zanshin.core            ] DEBUG: Method
   AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
   by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:38:58 [zanshin.monitoring      ] INFO: Processing method call: end / GConfirmTrans
2015-01-31 12:38:58 [zanshin.monitoring      ] INFO: Processing method call: fail / GConductTrans
2015-01-31 12:38:58 [zanshin.monitoring      ] INFO: Processing method call: end / GConductTrans
2015-01-31 12:38:58 [zanshin.monitoring      ] INFO: Processing method call: fail / GServeCust
2015-01-31 12:38:58 [zanshin.monitoring      ] INFO: Processing method call: end / GServeCust
2015-01-31 12:38:58 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:38:58 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
2015-01-31 12:39:05 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
   session atm/1.422.718.724.215: GConfirmTrans.START()
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement started: GConfirmTrans
   (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@6717c8 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement started: GConductTrans
   (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
   null, state: undefined) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement started: GServeCust
   (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement started: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
   state: undefined) (startTime: null))
2015-01-31 12:39:05 [zanshin.monitoring      ] INFO: Processing method call: start / GConfirmTrans
2015-01-31 12:39:05 [zanshin.monitoring      ] INFO: Processing method call: start / GConductTrans
2015-01-31 12:39:05 [zanshin.monitoring      ] INFO: Processing method call: start / GServeCust
2015-01-31 12:39:05 [zanshin.monitoring      ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:39:05 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
   session atm/1.422.718.724.215: GConfirmTrans.FAIL()
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement failed: GConfirmTrans
   (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@6717c8 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement ended: GConfirmTrans
   (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@6717c8 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement failed: GConductTrans
   (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
   null, state: started) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement ended: GConductTrans
   (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
   null, state: failed) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement failed: GServeCust
   (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement ended: GServeCust
   (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
   state: started) (startTime: null))
2015-01-31 12:39:05 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
   (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
   state: failed) (startTime: null))
2015-01-31 12:39:05 [zanshin.monitoring      ] INFO: Processing method call: fail / GConfirmTrans
2015-01-31 12:39:05 [zanshin.monitoring      ] INFO: Requirement GConfirmTrans has 1 AwReqs
   referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:39:05 [zanshin.adaptation      ] INFO: Processing state change: AR4 (ref.
   GConfirmTrans) -> failed
2015-01-31 12:39:05 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
   Retrieved existing session for AR4, 3 events already in the timeline
```

```
2015-01-31 12:39:05 [zanshin.adaptation       ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:39:05 [zanshin.core             ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Strategy RetryStrategy is applicable.
2015-01-31 12:39:05 [zanshin.adaptation       ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Selected adaptation strategy: RetryStrategy
2015-01-31 12:39:05 [zanshin.core             ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Applying strategy RetryStrategy(true; 2000)...
2015-01-31 12:39:05 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: copy-data(iGConfirmTrans, iGConfirmTrans)
2015-01-31 12:39:05 [zanshin.core             ] DEBUG: Replacing requirement instances of class
    GConfirmTrans (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@6717c8 (refinementType:
    and) (time: null, state: failed) (startTime: null) ->
    it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@621e69 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:39:05 [zanshin.core             ] DEBUG: The status of GConductTrans has been reset to
    undefined
2015-01-31 12:39:05 [zanshin.core             ] DEBUG: The status of GServeCust has been reset to
    undefined
2015-01-31 12:39:05 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: terminate(iGConfirmTrans)
2015-01-31 12:39:05 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: rollback(iGConfirmTrans)
2015-01-31 12:39:05 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: wait(2.000)
2015-01-31 12:39:05 [zanshin.controller       ] DEBUG: RMI Target System Controller forwarding
    instruction: initiate(iGConfirmTrans)
2015-01-31 12:39:05 [zanshin.adaptation       ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:39:05 [zanshin.core             ] DEBUG: Replacing requirement instances of class AR4
    (it.unitn.disi.zanshin.model.atm.impl.AR4Impl@1f2a26d (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR4Impl@a43578
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:39:05 [zanshin.core             ] DEBUG: The status of GProvideATM has been reset to
    undefined
2015-01-31 12:39:05 [zanshin.core             ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:39:05 [zanshin.core             ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:39:05 [zanshin.monitoring       ] INFO: Processing method call: end / GConfirmTrans
2015-01-31 12:39:05 [zanshin.monitoring       ] INFO: Processing method call: fail / GConductTrans
2015-01-31 12:39:05 [zanshin.monitoring       ] INFO: Processing method call: end / GConductTrans
2015-01-31 12:39:05 [zanshin.monitoring       ] INFO: Processing method call: fail / GServeCust
2015-01-31 12:39:05 [zanshin.monitoring       ] INFO: Processing method call: end / GServeCust
2015-01-31 12:39:05 [zanshin.monitoring       ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:39:05 [zanshin.monitoring       ] INFO: Processing method call: end / GProvideATM
2015-01-31 12:39:12 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.START()
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement started: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@621e69 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement started: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: undefined) (startTime: null))
2015-01-31 12:39:12 [zanshin.monitoring       ] INFO: Processing method call: start / GConfirmTrans
2015-01-31 12:39:12 [zanshin.monitoring       ] INFO: Processing method call: start / GConductTrans
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement started: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:39:12 [zanshin.monitoring       ] INFO: Processing method call: start / GServeCust
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement started: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:39:12 [zanshin.monitoring       ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:39:12 [zanshin.controller       ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.FAIL()
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement failed: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@621e69 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement ended: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@621e69 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement failed: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: started) (startTime: null))
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement ended: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: failed) (startTime: null))
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement failed: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:39:12 [zanshin.core             ] DEBUG: Requirement ended: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: failed) (startTime: null))
```

```
2015-01-31 12:39:12 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:39:12 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:39:12 [zanshin.monitoring      ] INFO: Processing method call: fail / GConfirmTrans
2015-01-31 12:39:12 [zanshin.monitoring      ] INFO: Requirement GConfirmTrans has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:39:12 [zanshin.adaptation      ] INFO: Processing state change: AR4 (ref.
    GConfirmTrans) -> failed
2015-01-31 12:39:12 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Retrieved existing session for AR4, 4 events already in the timeline
2015-01-31 12:39:12 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:39:12 [zanshin.core            ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Strategy RetryStrategy is applicable.
2015-01-31 12:39:12 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Selected adaptation strategy: RetryStrategy
2015-01-31 12:39:12 [zanshin.core            ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Applying strategy RetryStrategy(true; 2000)...
2015-01-31 12:39:12 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: copy-data(iGConfirmTrans, iGConfirmTrans)
2015-01-31 12:39:12 [zanshin.core            ] DEBUG: Replacing requirement instances of class
    GConfirmTrans (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@621e69 (refinementType:
    and) (time: null, state: failed) (startTime: null) ->
    it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@1c7a0ef (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:39:12 [zanshin.core            ] DEBUG: The status of GConductTrans has been reset to
    undefined
2015-01-31 12:39:12 [zanshin.core            ] DEBUG: The status of GServeCust has been reset to
    undefined
2015-01-31 12:39:12 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: terminate(iGConfirmTrans)
2015-01-31 12:39:12 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: rollback(iGConfirmTrans)
2015-01-31 12:39:12 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: wait(2.000)
2015-01-31 12:39:12 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: initiate(iGConfirmTrans)
2015-01-31 12:39:12 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:39:12 [zanshin.core            ] DEBUG: Replacing requirement instances of class AR4
    (it.unitn.disi.zanshin.model.atm.impl.AR4Impl@a43578 (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR4Impl@75c03b
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:39:12 [zanshin.core            ] DEBUG: The status of GProvideATM has been reset to
    undefined
2015-01-31 12:39:12 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:39:12 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:39:12 [zanshin.monitoring      ] INFO: Processing method call: end / GConfirmTrans
2015-01-31 12:39:12 [zanshin.monitoring      ] INFO: Processing method call: fail / GConductTrans
2015-01-31 12:39:12 [zanshin.monitoring      ] INFO: Processing method call: end / GConductTrans
2015-01-31 12:39:12 [zanshin.monitoring      ] INFO: Processing method call: fail / GServeCust
2015-01-31 12:39:12 [zanshin.monitoring      ] INFO: Processing method call: end / GServeCust
2015-01-31 12:39:12 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:39:12 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
2015-01-31 12:39:19 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.START()
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement started: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@1c7a0ef (refinementType: and) (time:
    null, state: undefined) (startTime: null))
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement started: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: undefined) (startTime: null))
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: start / GConfirmTrans
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: start / GConductTrans
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement started: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: start / GServeCust
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement started: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: undefined) (startTime: null))
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: start / GProvideATM
2015-01-31 12:39:19 [zanshin.controller      ] DEBUG: Received log for life-cycle method call in
    session atm/1.422.718.724.215: GConfirmTrans.FAIL()
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement failed: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@1c7a0ef (refinementType: and) (time:
    null, state: started) (startTime: null))
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement ended: GConfirmTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConfirmTransImpl@1c7a0ef (refinementType: and) (time:
    null, state: failed) (startTime: null))
```

```
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement failed: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: started) (startTime: null))
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement ended: GConductTrans
    (it.unitn.disi.zanshin.model.atm.impl.GConductTransImpl@1108e32 (refinementType: and) (time:
    null, state: failed) (startTime: null))
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement failed: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement ended: GServeCust
    (it.unitn.disi.zanshin.model.atm.impl.GServeCustImpl@1e792e9 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement failed: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: started) (startTime: null))
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Requirement ended: GProvideATM
    (it.unitn.disi.zanshin.model.atm.impl.GProvideATMImpl@5e1db8 (refinementType: and) (time: null,
    state: failed) (startTime: null))
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: fail / GConfirmTrans
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Requirement GConfirmTrans has 1 AwReqs
    referring to it. Assuming all AwReqs are NeverFail and reporting AwReq state change: fail
2015-01-31 12:39:19 [zanshin.adaptation      ] INFO: Processing state change: AR4 (ref.
    GConfirmTrans) -> failed
2015-01-31 12:39:19 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Retrieved existing session for AR4, 5 events already in the timeline
2015-01-31 12:39:19 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has not yet been solved...
2015-01-31 12:39:19 [zanshin.core            ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Strategy RetryStrategy is not applicable because it has been applied at least 5 time(s) this
    session.
2015-01-31 12:39:19 [zanshin.core            ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Strategy AbortStrategy is applicable.
2015-01-31 12:39:19 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Selected adaptation strategy: AbortStrategy
2015-01-31 12:39:19 [zanshin.core            ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266)
    Applying strategy AbortStrategy...
2015-01-31 12:39:19 [zanshin.controller      ] DEBUG: RMI Target System Controller forwarding
    instruction: abort(iAR4)
2015-01-31 12:39:19 [zanshin.adaptation      ] INFO: (Session: AR4 / 2015-01-31 12:38:44.266) The
    problem has been solved or there is nothing else to try. Adaptation session will be terminated.
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Replacing requirement instances of class AR4
    (it.unitn.disi.zanshin.model.atm.impl.AR4Impl@75c03b (refinementType: and) (time: null, state:
    failed) (incrementCoefficient: 1.0) -> it.unitn.disi.zanshin.model.atm.impl.AR4Impl@1dc1f4b
    (refinementType: and) (time: null, state: undefined) (incrementCoefficient: 1.0))
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass RetryStrategy. Make sure this is on purpose...
2015-01-31 12:39:19 [zanshin.core            ] DEBUG: Method
    AdaptationStrategyImpl.updateReferences() has been called, indicating it has not been overridden
    by the subclass AbortStrategy. Make sure this is on purpose...
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: end / GConfirmTrans
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: fail / GConductTrans
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: end / GConductTrans
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: fail / GServeCust
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: end / GServeCust
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: fail / GProvideATM
2015-01-31 12:39:19 [zanshin.monitoring      ] INFO: Processing method call: end / GProvideATM
```

C
Experiment Instruments

This appendix contains the instruments used during the empirical study described in Section 7.2: pre-experiment questionnaire, experiment tasks, and post experiment questionnaire. These instruments are a translated copy of their original versions in Portuguese.

C.1 Pre-experiment questionnaire

This appendix presents a questionnaire appplied to all subjects before the experiment. The goal of this questionnaire was to obtain basic information about the background of subjects, for characterization purposes.

Statechart Derivation with Adaptation
Experiment
Date:
Student: ____
Group: ( ) MULAS ( ) Control

Pre-experiment Questionnaire

The aim of this questionnaire is to obtain information about your background on systems modeling and software engineering. Your answers do not affect the other activities of this experiment, they simple provide us a context for the interpretation of results. Feel free to write beyond the designated lines in order to explain or detail your answers, if needed be.

1) On which academic level are you enrolled?

   ( ) Undergraduation ( ) Master ( ) Doctoral

2) What is/was your undergraduation course?

   ____

3) Do you have professional experience on software engineering?
   ( ) No ( ) Yes, for how long? ______________________

   If yes, which activities have you performed professionally?

   ____
   ____
   ____
   ____
   ____

4) Have you ever used a modeling language for describing systems behaviour before this training?
5) Você já utilizou alguma linguagem para modelagem do comportamento de sistemas antes desta disciplina?
   ( )No ( )State machines ( )Petri nets ( )Statechart
   ( )Class diagram ( )Activity diagram ( )Use cases
   ( )Goal models ( )Process diagrams
   ( )Other: which one (s)? ______________________________

6) Choose one of the alternatives with respect to the following statement: "I am proficient in a systems modeling language other than statecharts or other kinds of state diagrams"

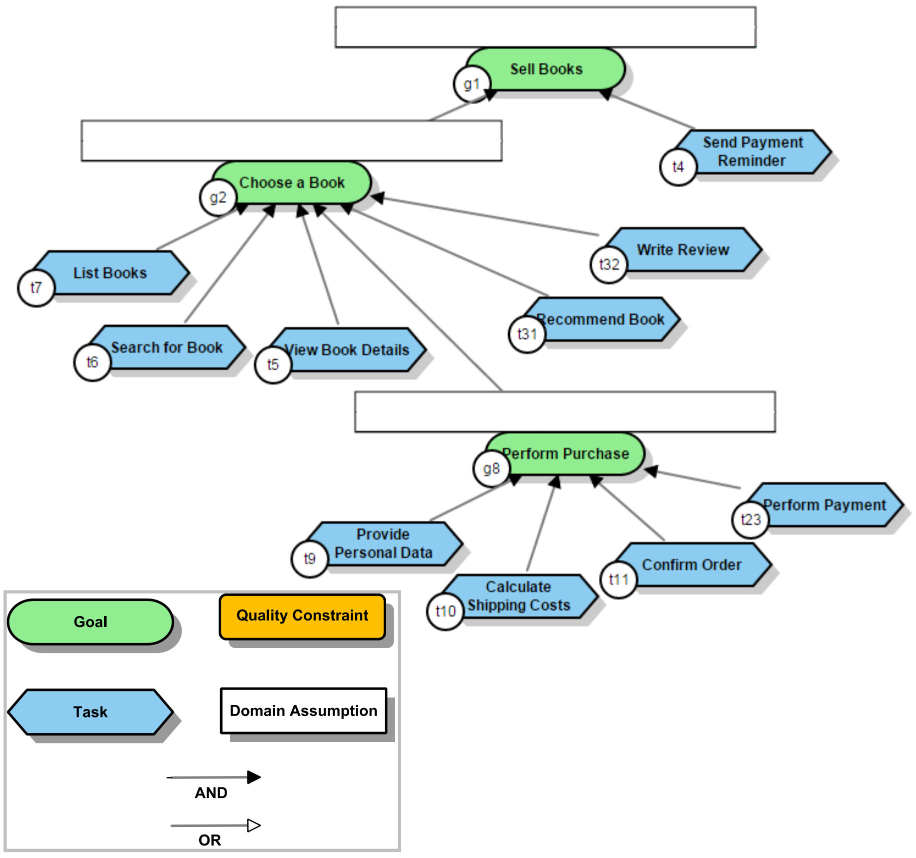
○	○	○	○	○
strongly disagree	disagree	neutral	agree	strongly agree

C.2 Object - MULAS group

This appendix shows a translated version of the main artifact that was handed to participants in the **MULAS** group uring the experiment reported on Section 7.2. This artifact describes the set of tasks that the subjects had to perform during the experiment.

Statechart Derivation with Adaptation
Experiment
Group: MULAS **Date:**
Student: ____

1) **Write in the goal model below the flow expressions for an e-commerce system, according to the behaviour described in the following text.**



Description of the expected behaviour:

The Saravá Bookstore intends to develop a web-based e-commerce system, where it can sell the books in its inventory. Accessing the website, users will see a listing of available books. If the user finds the desired book in the listing, she will be able to view the book details directly from the listing. If the book was not found on the listing, the user can use a search mechanism, as many times as warranted, until the book is found – then, the book details can be viewed.

Once the desired book is found and its details are viewed, the user has 3 options: either she buy the book, send a book recommendation, or write a book review.

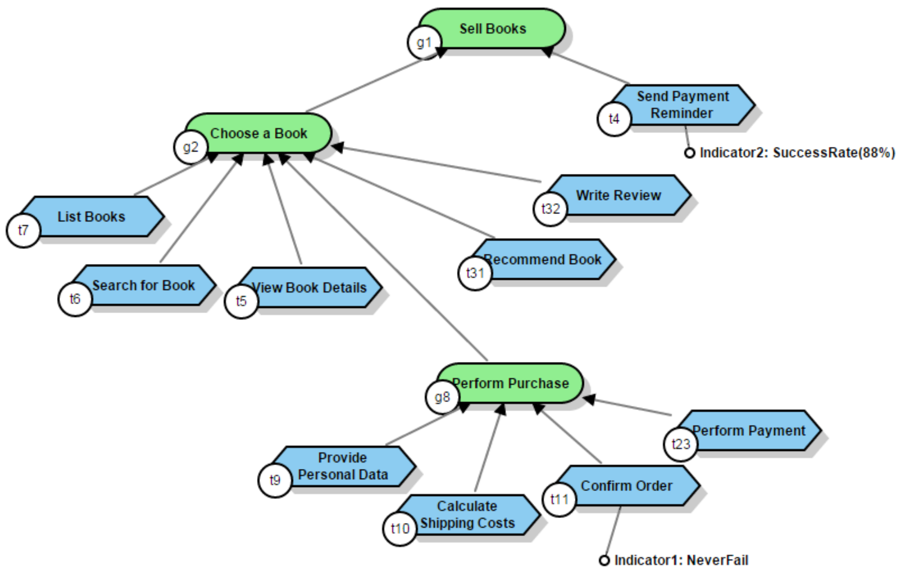
In order to perform the purchase, the user must follow the following procedure: first it is required to provide personal data, then calculate shipping costs, confirm order, and lastly perform the payment. However, the shipping costs calculation is not always performed, as the user may choose to have the book delivered on a physical store. To make a purchase, no kind of registering or login is required.

Regardless of user interaction, the system shall send e-mails at 12-hours intervals for those users whose payment have not been approved yet.

2) **Based on the flow expressions you wrote in the previous question, <u>create a statechart</u> that represents the expected behaviour of the system. This statechart should not contain anything less nor beyond what is specified by those expressions – if needed be, update the expressions in the previous question, but remember to always follow the behaviour described in question 1.**

3) **Include on your statechart the events for every one of its transitions.**

In order to answer the following questions, consider the indicators included to the goal model below.



Indicator	Adaptation Strategies
Indicator1: NeverFail	Retry(t11, *5s*);
Indicator2: SuccessRate(88%)	Notify("SysAdmin");

4) **Write in the table below the monitoring points for this system (<u>start</u> and <u>end</u>), considering its indicators and its behaviour.**

Indicator	Monitoring Points	
	Start	End
Indicator1: NeverFail		
Indicator2: SuccessRate(88%)		

5) **Using pencil or pen with a different colour, <u>include in your statechart the necessary monitoring actions</u>.**

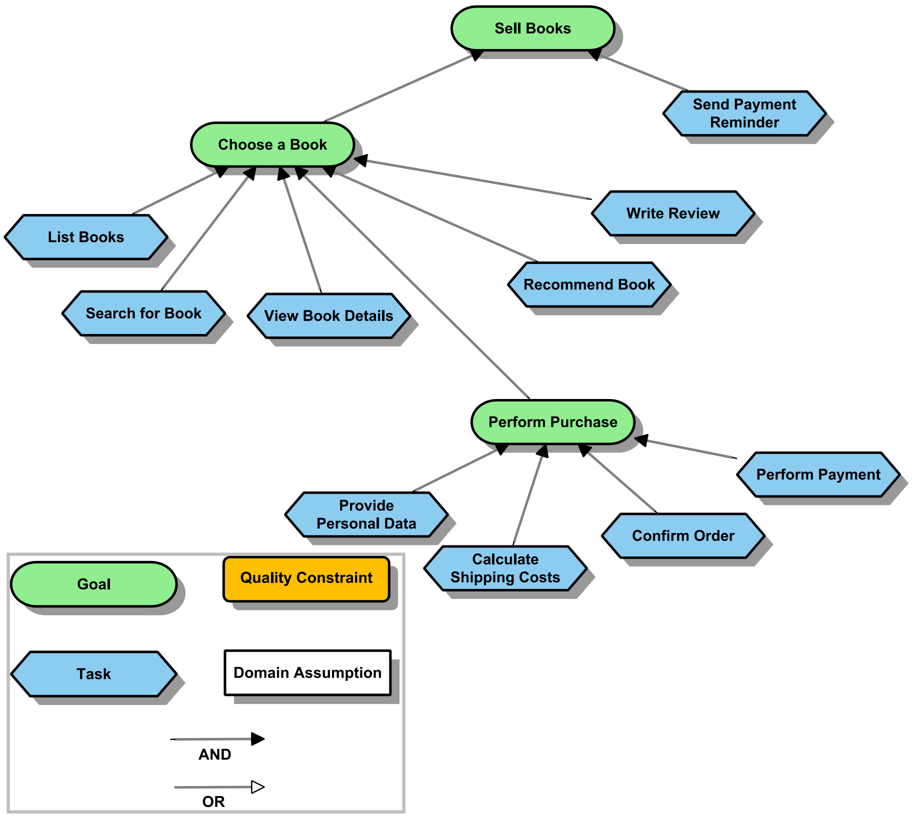
6) **Based on the <u>adaptation strategies</u> specified in the first table on this page, <u>include in your statechart</u> the elements required in order to enact these strategies. Then again, use pencil or pen with a different colour.**

C.3 Object - Control group

This appendix shows a translated version of the main artifact that was handed to participants in the **control** group during the experiment reported on Section 7.2. This artifact describes the set of tasks that the subjects had to perform during the experiment.

Statechart Derivation with Adaptation
Experiment
Group: Control **Date:**
Student: ____

1) **Consider the goal model below, representing an e-commerce system, and the behaviour described in the following text.**



Description of the expected behaviour:

The Saravá Bookstore intends to develop a web-based e-commerce system, where it can sell the books in its inventory. Accessing the website, users will see a listing of available books. If the user finds the desired book in the listing, she will be able to view the book details directly from the listing. If the book was not found on the listing, the user can use a search mechanism, as many times as warranted, until the book is found – then, the book details can be viewed.

Once the desired book is found and its details are viewed, the user has 3 options: either she buy the book, send a book recommendation, or write a book review.

In order to perform the purchase, the user must follow the following procedure: first it is required to provide personal data, then calculate shipping costs, confirm order, and lastly perform the payment. However, the shipping costs calculation is not always performed, as the user may choose to have the book delivered on a physical store. To make a purchase, no kind of registering or login is required.

Regardless of user interaction, the system shall send e-mails at 12-hours intervals for those users whose payment have not been approved yet

2) **Based on the content of the previous question, <u>create a statechart</u> that represents the expected behaviour of the system. This statechart should not contain anything less nor beyond what is specified in the previous question.**
When creating this statechart, the use of states, transitions, and events is <u>mandatory</u>. The use of super-states, orthogonal states, conditions, variables, actions, and default states is <u>optional</u>.

3) **Consider the additional behaviour presented in the following text.**

Some tasks of the system are more critical than others. Considering that "Confirm Order" is a very important task, the stakeholders decided the following: every time a failure occurs during the confirmation, the system must try to perform the confirmation again, at 5 seconds intervals, until the confirmation is performed successfully.

Another important task of the system is the sending of payment reminders. The stakeholders specified that this sending must be performed successfully in at least 88% of the cases – otherwise, the system administrator must receive a notification e-mail.

4) **Using pencil or pen with a different colour from the one you used to draw the statechart, <u>modify the statechart</u> in order to include the additional behaviour described in the previous question. If you need to remove any element, <u>scribble over it, instead of erasing it</u>.**

C.4 Post experiment questionnaire

This appendix presents the post experiment questionnaire that was applied to subjects in the MULAS group, through which it was possible to obtain subjective feedback about the MULAS framework.

Statechart Derivation with Adaptation
Experiment
Group: MULAS **Date:**
<u>This questionnaire is anonymous – do not write your name</u>

For each one of the statements below, select whether you strongly disagree, disagree, is neutral, agree, or strongly agree.

a. The mapping from tasks to states facilitates the creation of statecharts

○	○	○	○	○
strongly disagree	disagree	neutral	agree	strongly agree

b. The mapping from goals to super-states improves the organization/structure of statecharts

○	○	○	○	○
strongly disagree	disagree	neutral	agree	strongly agree

c. The use of flow expressions facilitates the creation of statecharts

○	○	○	○	○
strongly disagree	disagree	neutral	agree	strongly agree

d. The use of goal models facilitates the creation of statecharts

○	○	○	○	○
strongly disagree	disagree	neutral	agree	strongly agree

e. The use of flow expressions makes the creation of statecharts more systematic

○	○	○	○	○
strongly disagree	disagree	neutral	agree	strongly agree

f. The creation of statecharts contributes to a more complete specification of the system

○	○	○	○	○
strongly disagree	disagree	neutral	agree	strongly agree

g. Statecharts facilitate system understanding

○	○	○	○	○
strongly disagree	disagree	neutral	agree	strongly agree

h. The use of patterns facilitate the reification of the specified adaptations

○	○	○	○	○
strongly disagree	disagree	neutral	agree	strongly agree

i. It is easy to specify sequential behaviour with flow expressions

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

j. It is easy to specify alternative behaviour with flow expressions

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

k. It is easy to specify optionality with flow expressions

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

l. It is easy to specify repetition with flow expressions

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

m. It is easy to specify parallelism with flow expressions

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

n. It is easy to include idle states with flow expressions

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

o. The pattern to reify the `abort` adaptation action is easy to understand

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

p. The pattern to reify the `notify` adaptation action is easy to understand

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

q. The pattern to reify the `reconfigure` adaptation action is easy to understand

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree

r. The pattern to reify the `retry` adaptation action is easy to understand

○ strongly disagree ○ disagree ○ neutral ○ agree ○ strongly agree