PhD-FSTC-34-2010

Faculté des Sciences, de la Technologie et de la Communication

THÈSE

Soutenue le 07/12/2010 à Luxembourg

En vue de l'obtention du grade académique de

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

par

Alfredo Capozucca
né le 13 novembre 1976 à Casilda (Argentine)

DT4BP: A Business Process Modelling Language for Dependable Time-Constrained Business Processes

Jury de thèse

Dr Pascal Bouvry, président
Professeur, Université du Luxembourg

Dr Eric Dubois, président suppléant
CRP Henri Tudor (Luxembourg)

Dr Nicolas Guelfi, directeur de thèse
Professeur, Université du Luxembourg

Dr Alexander Romanovsky
Professeur, Université de Newcastle (Royaume-Uni)

Dr Andreas Roth
SAP AG (Allemagne)

To Paola, Matías and Santiago.

ABSTRACT

Today, numerous organisations rely on information software systems to run their businesses. The effectiveness of the information software system then, depends largely on the degree to which the organisation's business is accurately captured in the *business model*. The *business model* is an abstract description of the way an organisation's functions. Thus, the more precise the *business model*, the more accurate the requirement definition of the information software system to be engineered.

There are an abundance of tools and notations available today to support the development of many types of business process. Many of these artifacts rely on the concept of a *business process* to describe a business model. A business process is commonly known as *"a set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships"*. This thesis is concerned with modelling business processes as a means to accurately capture an organisation's activities and thus, the requirements of the software system that supports these activities.

Among the infinite set of possible business processes, this thesis targets only those characterised by the qualities of *dependability, collaboration* and *time*. Business processes having these specific dimensions are referred to as *Dependable, Collaborative and Time-Constrained (DCTC) business processes*. A *dependable* business process is one whose failures or the number of occurrences in which business process misses its goal are not unacceptably frequent or severe (from certain viewpoint). A *collaborative* business process is one that requires the interaction of multiple participants to attain its goal. A *time-constrained* business process is one that owns at least one property expressed in terms of an upper or lower time bound. This thesis investigates how DCTC business processes can be described such that the resulting model captures all the relevant aspects of each dimension of interest. In addition, the business model must be comprehensible to the stakeholders involved not only in its definition, but also in its further use throughout the software development life cycle.

A revision and analysis of notations that exist for modelling business processes conducted in this thesis have revealed that today there does not exist any modelling language that provides comprehensible, suitable and sufficiently expressive support for the characteristics of dependability, collaboration and time in an integrated manner. Hence, a significant part of this thesis is devoted to the definition of a new business process modelling language named *DT4BP*. The aim of this new modelling language is to be comprehensible, suitable and expressive enough to describe DCTC business processes.

The definition of this new modelling language implies that a *concrete syntax*, an *abstract syntax*, a *semantic domain* and a *semantic mapping* is provided. The definition of this new modelling language is given following the Model-Driven Engineering (MDE) approach, and in particular the metamodelling principles. Thus, *meta-models* and *model transformations* are used to precisely specify the abstract syntax and semantic mapping elements of the language definition,

respectively. Since *DT4BP* is a *textual* modelling language, its concrete syntax is specified by a context-free grammar. The *Coordinated Atomic Actions* conceptual framework with real-time extensions (*Timed-CaaFWrk*) is used as the semantic domain as it covers a large part of the abstractions included in dependable collaborative time-constrained business processes. The formalisation of this semantic domain according to the metamodelling principles is also part of the material presented in this thesis.

Since the business model is considered as a representation of the requirements document the software system to be developed, it is crucial to validate whether it captures the requirements as intended by the stakeholder before going further in the software development process. Hence, besides the comprehensibility, suitability and expressiveness of the modelling language with respect to the domain of interest, it is of special interest to provide a mechanism that allows modellers to ensure that the business model is correct with respect to the stakeholder's expectations. One way of achieving this goal is to provide the modelling language with an executable semantics. In this manner, any business model can be executed on sample input data, and its dynamic behaviour observed. The observation of the dynamic behaviour of the model may be considered as a *simulation* of the model based on the sample input data.

By performing several simulations of the model, the modeller, in cooperation with the stakeholder, can judge whether the business model is correct. This thesis provides an executable semantics for *Timed-CaaFWrk* that, used in combination with the model transformation that defines the semantic mapping element of the language definition, allows *DT4BP* models to be validated by simulation. In this manner, the dynamic behaviour of a particular *DT4BP* model for a given sample input data can be observed by transforming it into a *Timed-CaaFWrk* model, which is then run thanks to the given executable semantics.

ACKNOWLEDGEMENTS

This thesis could not have been completed without the help of others. I would like to express my gratitude to them.

First of all, I am grateful to Prof. Nicolas Guelfi for supervising this research work and for supporting me over the years. But above all I would like to thank him for having guided me over the years with his talent and sharp criticism, which has made me (I guess) a better researcher.

I am grateful to the jury members Prof. Pascal Bouvry, Prof. Eric Dubois, Prof. Alexander Romanovsky and Dr. Andreas Roth for having accepted to serve on my examination board and for the time they have invested in reading and evaluating this thesis.

I greatly appreciate and wish to thank all the (former and current) LASSY Lab. team members of the University of Luxembourg for providing a great atmosphere. Special thanks go to Benoîte Ries for his encouragement during the (difficult) writing period of the thesis, and Moussa Amrani for the fruitful (and not always) subject-related discussions we had and which helped me to develop a different understanding of the subject and for his careful review of (certain parts) of the thesis.

I owe my deepest gratitude to Federico Wiecko (my office companion) for his great work in implementing the (ever changing) Timed-CAAA-DRIP framework and for assisting me throughout the implementation of the ATL transformation. I also want to thank him for having carefully reviewed parts of this thesis and having pointed out issues that led to significant improvements of the final version of the manuscript.

I am grateful to Dr. Reza Razavi, who gave me the opportunity to join the LASSY Lab. (SE2C at that point in time) in the context of the CORRECT project.

I also want to thank Alexandre Ewen for his accurate and quick response every time I needed to print a version of the thesis, and Mireille Kies and Daniele Flammang for their help in dealing with the administrative side of the thesis.

I am grateful to Prof. Avelino Zorzo for the interesting discussions we had during my visit to the Ponthifical Catholic University of RS. His knowledge of multiparty interactions and and CAA-DRIP have been invaluable to me. I also want to thank Prof. Joerg Kienzle not only for the discussions we had during his visit to the LASSY Lab. about time-related exception handling, but also for his general advice and suggestions.

My eternal gratitude goes to my parents Norma Busilacchi and Raúl “Cuadro” Capozucca for their support, comprehension, confidence and, especially for instilling into me that education is the basis of progress. Gracias Mami y Papi.

Finally, I want to thank my sons Matías and Santiago for their smiles and energy that helped me fight the stress involed in writing my thesis. But most of all I thank my wife Paola, whose

care, devotion, support and love have allowed me to reach the final stage of this great challenge of completing a PhD.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Abstract

This chapter introduces the problems that this thesis aims to address. In addition, the issues that motivated the thesis are discussed. The chapter begins with a presentation of the dimensions (or concerns) that circumscribe the research domain. Next, the problems that have been identified within this domain are introduced. This discussion is followed by a list of contributions aimed at solving these problems. The chapter concludes with a description of the organisation of the thesis. Please note, that no references are provided in this chapter. Here the focus is on the scope of the research domain along with its associated concepts. These are both discussed in depth later in the Chapter focusing on the state of the art, Chapter 2.

1.1 Research domain

In today's world, numerous organisations[1] rely on software systems[2] to run their businesses. The software system must assist the organisation to provide the services or products it offers to its customers. The effectiveness of the software system in achieving these goals depends largely on the degree to which the organization's business is accurately captured in the business model. A business model is a simplified view (i.e. abstraction) of the business that has to be supported by the software system. An accurate business model can, thus, be considered as the requirements document that is needed for developing a software system to support the organisation's activities.

1.1.1 Business process modelling

A central concept used for modelling businesses is the *business process.* A business process is defined by the community as *"a set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships"*. A *process definition* is a description or representation of what a particular business process is intended to do. The process definition, then, is the model that captures those aspects of the business that must be supported by the software system. This thesis is concerned with 1) the modelling of business processes as a means to capture the organisation's activities and 2) the requirements of the software system that supports these activities.

[1] Here the term organisation is used as synonym of enterprise.

[2] The expression *software systems* is used to indicate the piece of software that satisfies the user's requirements along with the hardware and environment where it is deployed.

1.1.2 Dependable collaborative business processes with time constraints

Among the infinite number of possible business processes, this thesis targets only a particular set of business processes. The first characteristic that narrows the domain of business processes to investigate is *dependability*. Other characteristics that further specify the kind of business process are the notions of *collaboration* and *time*. Hence, attention here is focused not only on dependable business processes, but also on those that are collaborative and time constrained. Collaborative here means that multiple participants take part throughout the process to achieve. Time-constrained indicates that there exists at least one element in the business process that owns a time-related property.

1.1.2.1 Dependability

A dependable business process is one whose failures, i.e. occurrences when the business process misses its goal, are not unacceptably frequent or severe from some particular viewpoint. Business process reinforcement is achieved by following an iterative modelling process aiming at improving the dependability of the business process. Dependability capabilities are achieved by introducing explicit recovery activities into the business process model. As a result, a business process[3] is defined as dependable when it is capable of achieving its business objective completely, as initially promised or partially when encountering situations that would cause the business process to fail. Partially in this context means sufficiently well enough to satisfy the requester's expectations. Situations that may lead a business process to fail range from technological, e.g. the Y2K problem, or financial circumstances, e.g. the sub-prime mortgage crisis, to environmental, e.g. the global greenhouse effect. All these examples have the common characteristic of forcing *changes* in the way the business process is being performed. The ability of the business process to tolerate, recover and react to these changes while providing its business objective, defines its dependability.

1.1.2.2 Collaboration

Nowadays companies often require several people with various skills to provide the services they offer. This requirement imposes the constraint that business processes are composed of multiple participants that collaborate at different stages in the process in order to provide the service or product (i.e. the process' goal). Thus, business process with two or more participants are named *collaborative business processes*. In this kind of business process, the collaboration is defined by the participants and their interactions. In *intra-organisational* collaborative business processes, participants represent different units or people within the same organisation. Whereas in *cross-organisational* collaborative business process each participant represents an entire organisation.

Collaborative business processes are becoming the rule rather than the exception due to the fact that companies try to keep the number of permanent employees to a minimum. Such a requirement has the result that companies rely more and more on *outsources*, i.e third-party companies that perform some specific activities for the company that contracts them. In companies with this kind of organisation, the collaboration among the participants turns from intra-enterprise to inter-enterprise, which implies not only an increase in the communication among participants, but also a change in the communication pattern employed, i.e. peer-to-peer rather than a centralised server-based service.

[3] In this context, the notion of business process includes the organisation that owns it.

Another reason collaborative business processes are becoming more important, is due to intensified globalisation as organisations need to transfer information faster, make decisions quicker, adapt to changing demands, deal with larger pools of international competitors and reduce their cycle times.

1.1.2.3 Time

Time constraints are different rules expressed in terms of maximums and/or minimums, which are aimed at determining how a particular business process is expected to behave from a temporal viewpoint. A business process is considered as time-constrained when it includes at least one time constraint in its definition.

A time constraint may be set over different elements within the process definition. For example, a time constraint can be used to determine the maximum allowed duration of the overall business process, or just one of the activities performed by a certain participant. Moreover, the (direct and indirect) effects of a time constraint also depend on the element to which it is attached. A time constraint set over an activity only affects the time related to its execution, whereas a time constraint set over a participant indirectly affects all the activities enclosed by the participant, since they are expected to be completed by certain point in time (at the latest).

Time constraints may also be applied over the data a business process requires to achieve its goal. In this case, a time constraint attached to certain data value may be used to determine the validity of its time stamp, in the sense that beyond that point in time the data becomes obsolete.

Depending on the application domain where the business process is engineered, time constraints such as those mentioned may be required by the end-users. However, it is worth noting that the existence of time constraints in a business process may not only be due to user requirements, but may also be due to organisational rules, laws, commitments, policies and standards to which the business process must adhere. Hence, it often arises that time-related information must be included in the process definition.

1.1.3 Metamodelling

The particular set of business processes addressed in this thesis is defined by the intersection of the orthogonal dimensions of dependability, collaboration and time. Business processes belonging to this set are referred to as *Dependable, Collaborative and Time-Constrained (DCTC) business processes*. This thesis is focused on modelling DCTC business processes.

The success of modelling DCTC business processes depends largely on the how close the modelling notation maps to the problem domain. Modelling notations with built-in support for the concerns of the domain of interest not only ease the modelling burden, but also enhance the readabillity and comprehension of the resulting models. It is thus of particular interest to have a domain-specific modelling language (DSML) to describe DCTC business processes.

This thesis adheres to the Model Driven Engineering (MDE) principles, and in particular to the metamodelling technique when addressing the definition of modelling languages. Hence, the notions of meta-model, model compliance, and model transformation are major concerns on which this thesis relies on for its contributions.

1.1.4 Business process validation

Since the business model or process definition may be considered as the requirements document needed to accurately develop the software system, it is crucial to validate whether the business model captures the requirements intended by the customer before progressing in the software development process. Hence, in addition to having the modelling language close to the domain of interest, it is of special interest to have means that allow business analysts and software engineers to check the correctness of the business model with respect to customer's expectations. One way of achieving this is to use a modelling language with an executable semantics. In this manner, any business model can be executed on sample input data, and then its dynamic behaviour can be observed. The observation of the dynamic behaviour of the model may be considered as a *simulation* of the model based on the sample input data. The customer then, by performing several simulations of the model, can judge whether the business model adequately captures their expectations or not. This procedure for validating business processes is also a concern of this thesis.

1.1.5 Coordinated Atomic Actions

The idea of a dependable business process relies on concepts and principles coming from the dependable computing domain, in particular those features concerned with fault tolerance. Coordinated Atomic Actions is a fault tolerance conceptual framework (CaaFWrk) aimed at designing complex distributed systems consisting of components that can both cooperate and compete amongst themselves. The CaaFWrk unifies the concepts of *conversations*, originally intended to ensure fault tolerance of cooperative systems and *transaction*, originally intended to ensure fault tolerance and structuring of the competitive systems. In addition, it incorporates a concurrent exception handling scheme that provides dependability in distributed and concurrent systems.

CaaFWrk, introduced in the early 90's, has since become a rich and sound set of results, which have been developed by the community. The concepts brought by the CaaFWrk are, for the most part, abstractions of the concepts included in collaborative business processes. Therefore, this thesis selects CaaFWrk as the foundation for the conceptual framework underpinning the language for modelling DCTC business processes.

1.2 Problem statement

The *business analyst* is the person in charge of providing the process definition or model. As the name states, this person should be business oriented since the modelling requires some understanding of organisational issues, policies and corporate directions. People with a background in software engineering may also participate in the production of the process definition since the business solution is required to be aligned with the software system that supports its execution.

The fact that people with different profiles, i.e. business and IT, need to cooperate and communicate in the production of the business process, implies that the notation for achieving the models must be comprehensible, i.e. the notation should ease the writing of models capable of being understood by every stakeholder. Along with comprehensibility, suitability and expressiveness are also important characteristics to be considered when selecting the modelling language. While suitability determines the ease with which a concept can be captured, expressiveness focuses on the facilities that formulate the different concepts of interest to the type of business process being modelled.

This thesis defines a modelling language that is comprehensible, suitable and expressive enough for modelling dependable, collaborative and time-constrained business processes. This business process modelling language should:

- include concepts related to dependability, collaboration and time, that result from adapting, at a syntactic and semantic level, the Coordinated Atomic Action conceptual framework,
- adhere to the model driven engineering paradigm by using metamodelling and model transformation techniques, and
- be provided with an executable semantics to allow validation by simulation.

1.3 Contributions of this thesis

The main contributions of this thesis are the following:

1. A new business process modelling language, *Dependability and Time for Business Processes (DT4BP)*, suitable for modelling dependable, collaborative, and time-constrained (DCTC) business processes.

   The *DT4BP* modelling language has been defined by focusing on the three equally important aspects of *dependability, collaboration*, and *time*. The language allows the modeller to explicitly capture information related to the different actions the entities must perform within the same business process. In addition, the language also allows the modeller to determine the way these entities interact amongst themselves to achieve the business process goal. The language also provides features to model exceptional situations that may arise during execution, as well as the steps that allow the business process to recover from such a situation. *DT4BP* has a powerful set of time-related primitives that allow the modeller to attach time constraints to the different kinds of elements involved in a process definition. However, the most important aspect of the language is the integrated way in which the concerns of dependability, collaboration and time have been addressed. It is because of this integration that there exist primitives, to model, among other things, exceptional situations that arise during the interaction of two different entities or due to a missing time constraint. Last, but not least, the fact that the language has been defined following the metamodelling principles also contributes to the validation of the Model Driven Engineering (MDE) approach as a means for the definition of domain-specific languages.

2. A new version of the CaaFWrk conceptual framework with real-time extensions (called Timed-CaaFWrk) to support the design of dependable distributed real-time object-oriented software systems.

   The Timed-CaaFWrk provided in this thesis allows the software designer to define different temporal reference frames for the participants entering into the same Coordinated Atomic Action (CAA). For example, the maximum allowed time participants have to get involved into the execution of a CAA and the maximum allowed elapse time to complete the execution of the CAA may both be constrained. The time-related extensions given to the conceptual framework also allow the software designer to define periodic CAAs, set constraints over the data objects used during the execution of the CAA, or set delays or

deadlines over a subset of the instructions to be performed by certain participant once engaged within a CAA. The definition of the Timed-CaaFWrk has been given according to the metamodelling principles, which contribute to the formalisation of the conceptual framework.

3. An implementation of the Timed-CaaFWrk for the Java programming language called Timed-CAA-DRIP.

   This implementation framework is meant exclusively to support the development and execution of a software system designed according to the Timed-CaaFWrk. It provides a set of Java classes to allow programmers to implement the particular functionalities of the software system, while at the same time respecting the structure defined at the design level. The Java classes with which the programmers have to interact, adhere to the same terminology employed at the design level, making the gap between design and implementation shorter. At run-time, the customisations made by the programmers are automatically bound up with the built-in classes enclosed within the framework, such that the implemented software system executes according to the conceptual framework principles.

4. A *model-to-model (M2M)* transformation to automate the generation of a Timed-CaaFWrk-compliant model from a given DT4BP-compliant model.

   Defining the *DT4BP* modelling language according to the metamodelling principles implies providing a model-to-model (M2M) transformation between *DT4BP* and a *Timed-CaaFWrk* to supply the *semantic mapping* (one of the language definition elements). Providing the semantic mapping as an M2M transformation has the potential benefit of translating *DT4BP* models into Timed-CaaFWrk models.

5. A *model-to-text (M2T)* transformation to automate the generation of Java code from a given design described in terms of the Timed-CaaFWrk.

   Having formalised the Timed-CaaFWrk according to the metamodelling principles allows the software designer or programmer to define a M2T transformation, whose goal it is to automate the development phase. This M2T transformation then takes a Timed-CaaFWrk-compliant model and generates Java source code that interfaces with the Timed-CAA-DRIP. The source code obtained as result of this M2T transformation adheres to the best practices principles regarding the interfacing with Timed-CAA-DRIP.

6. A tool to validate *DT4BP* models by simulation

   Let the M2M transformation that provided the Timed-CaaFWrk-compliant model from a DT4BP-compliant model be T_1, and let the M2T transformation that provided a Java implementation from a given Timed-CaaFWrk-compliant model be T_2 (both contributions previously mentioned). Then it is possible to compose T_1 with T_2 (i.e. $T_1 \circ T_2$) such that given a certain *DT4BP* model M_{DT4BP}, a Java source Timed-CaaFWrk-compliant M_{Java} implementation can be automatically obtained. Hence, this Java source code M_{Java}, after a by-hand completion, can be compiled and subsequently executed to simulate M_{DT4BP}.

It is worth emphasising that the results of this thesis contribute to the domains of business processes and fault tolerance. With respect to the business processes domain, the thesis contribution focuses on modelling and validation (contributions (1) and (6). For the fault tolerance domain, the contributions focus on Coordinated Atomic Actions (contributions (2),(3), (4) and (5)). Therefore, anyone interested in either of these domains may benefit from the novel approaches considered in this thesis.

Finally, a running example is used to demonstrate how the different *DT4BP* features are used to explicitly to capture the dependable, collaborative and time-related aspects of this case study. In addition, this running example demonstrates the comprehensibility, suitability and effectiveness of *DT4BP*, which allow business analysts to model DCTC business process. However, it is worth mentioning that no empirical assessment has been carried out to evaluate whether *DT4BP* properly fulfils the attributes of comprehensibility, suitability and effectiveness in broader application domains.

The same running example is also used as vehicle to realise a proof-of-concept of the proposed validation tool.

1.4 Thesis organisation

The thesis is organised into seven chapters, plus five appendices containing additional information the reader may find useful for understanding some of the concepts that are introduced. Here, a summary of the content of each chapter and appendix is given.

Chapter 2 introduces the background related to the areas of *business processes, dependability,* and *model-driven engineering* as the thesis contributions rely heavily on these concepts. In this chapter the thesis introduces: 1) the business processes terminology, 2) the concepts coming from the dependability computing area that underlie the notion of dependable business processes and 3) the concepts and principles that govern the Model-Driven Engineering field, with particular emphasis on those concepts and principles dealing with the definition of domain-specific languages.

Chapter 3 describes DT4BP, a novel business process modelling language designed for specifying dependable, collaborative and time-constrained (DCTC) business processes. The chapter describes each of these dimensions. This discussion is then followed by an analysis of the support provided by existing business process modelling languages with respect to these dimensions. After having introduced the domain of interest and the state of the art with respect to its modelling, the DT4BP modelling language is described. The chapter concludes with an analysis and comparison of DT4BP with existing modelling languages to demonstrate how this novel language exceeds the capabilities of existing notations.

Chapter 4 describes Timed-CaaFWrk, a new version of the Coordinated Atomic Actions conceptual framework that includes real-time extensions. The chapter starts by presenting the fundamental concepts that form the Coordinated Atomic Action conceptual framework (referred to as CaaFWrk). After establishing the basis of the conceptual framework, its real-time extensions are described. This chapter also includes a description of the Timed-CAA-DRIP implementation framework along with the interface that allows programmers to use it. The chapter closes with a description of the model-to-text transformation that is used to automatically obtain Java source code. The code is generated by taking as input a software system design given according to the Timed-CaaFWrk principles.

Chapter 5 describes the semantic mapping that relates each *DT4BP* concept with a concept of the chosen semantic domain, which in this case is the Timed-CaaFWrk conceptual framework. The chapter starts by introducing the notation used to describe the semantic mapping. This is followed by a description of the semantic mapping. The chapter concludes with a description of how the semantic mapping can be combined with other model driven artifacts to execute *DT4BP* models. The discussion also covers how this approach can be used for validation purposes.

Chapter 6 describes the extensions, improvements and potential directions for future research with respect to the *DT4BP* modelling language, Timed-CaaFWrk and the Timed-CAA-DRIP implementation framework. Chapter 7 completes the thesis, summarising the main results that were achieved.

There are five appendices. Appendix A contains the grammar that specifies the concrete syntax of the *DT4BP* modelling language. Appendix B demonstrates, in one consolidated figure, all the concepts and relationships that form part of the *DT4BP* meta-model. Appendix C provides the full *DT4BP* models concerning the *diagnosis* case study, which is used mainly in Chapter 3 as a vehicle to exemplify the different features that make up the *DT4BP* modelling language. Next, Appendix D uses a single figure to combines all the concepts and relationships that form part of the *Timed-CaaFWrk* meta-model. Finally, Appendix E lists all the transformation rules that define the semantic mapping between *DT4BP* and the semantic domain *Timed-CaaFWrk*, as well as the meta-models used by such transformation.

2. BACKGROUND

Abstract

The scope of the research presented in this thesis covers the areas of: (1) business processes, (2) dependability, and (3) Model-Driven Engineering (MDE). The goal of this chapter is to introduce relevant aspects belonging to each of these areas providing the background on which the thesis relies. The presentation of the background is structured into three parts, each representing the three areas of interest. Section 2.1 introduces the terminology regarding business processes, as well as concepts and aspects regarding the modelling and validation of business processes. Section 2.2 gives a detailed presentation of the background regarding the dependability of the computing area, with a special emphasis on those concepts that underpin the notion of dependable business processes as considered within this thesis. Section 2.3 presents the MDE field by giving information about its aim, and describing in detail the concepts and principles that govern this topic when specifying domain-specific languages.

2.1 Business processes

Within the context of this thesis, it is used the definition of *business process* provided by the Workflow Management Coalition (WfMC) [Wor] in its Terminology & Glossary [Wor99] released at the end of the 90's. According to this document, a *business process* is considered as *"a set of one or more linked procedures or activities, which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships"*.

Many people consider the workflow domain as the predecessor to *Business Process Management* (BPM), the current area that *"supports business processes using methods, techniques, and software to design, enact, control, and analyse operational processes involving humans, organisations, applications, documents and other sources of information."* [van04]. Therefore, since BPM has its origins in the workflow domain, it makes sense to consider such a definition for the term *business process*.

Other definitions for business process have also been provided. Ko in [Ko09] lists those definitions that have their origins in the area of business process re-engineering (BPR). However, all of these (with more or less emphasis) view a business process in the same perspective, i.e. as a set of ordered activities designed to take one or more inputs, produce a specific output for a particular stakeholder and which is carried out collaboratively by a group of humans and/or machines within an organisation.

It is worth noting that the definition of BPM provided by van del Aalst restricts business processes only to those that are *operational*. Operational business processes are those that enclose

the current activities the organisation carries out in order to meet (medium-term) tactical or (long-term) strategic decisions. Business processes that cannot be made explicit (like *strategic* or *management* business processes) are left out of the BPM's domain. This method of classifying business processes is inherited from the way in which traditional organisation charts are organised: *operational control*, *management control* and *strategic planning* [Ant65].

The definition about BPM encompasses the concept of "supporting" business processes. This support represents the different phases that allow a business process to be realised. These phases, which are enclosed in what it is called the BPM life cycle, are:

- **Design:** in this phase, a business process is realised by a model, which must capture all its relevant aspects, in a manner such that the model is understood by the person providing the model (aka modeller), but also any other individual concerned with its definition (e.g. stakeholders) and implementation (e.g. IT managers or developers),
- **Configuration:** this phase is concerned with the implementation of the business process model (produced in the previous phase) by means of configuring a process-aware information system in order to obtain IT support for the execution of the business process,
- **Enactment:** this stage covers the deployment of the business process to achieve its execution using the IT support,
- **Diagnosis:** the results of having executed a business process (aka traces) can be used to analyse its behaviour in order to identify existing problems or ways in which the model can be improved.

Once problems or areas for improvement have been identified, the business process has to be re-designed to address these issues. This leads to the restart of the BPM life cycle, thus, defining an iterative closed-loop life cycle.

With the exception of the first phase (i.e. Design), the rest of the life cycle phases are concerned with the process-aware information system that provides support for running the business process. Information technologies that focus on process management are good candidates for achieving the process-aware information system that will provide the required support for running such business. Workflow Management Systems (WfMS) and Enterprise Resource Planning (ERP) systems are two distinct solutions that focus on business processes that have received special interest in the past two decades [CBS04].

A WfMS is a system that defines, creates and manages the execution of business processes through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with business process participants and, where required, invoke the use of IT tools and applications [Wor99].

On the other hand, an ERP is a generic off-the-shelf system that supports most of the key functions (e.g. logistics, sales, and financial management) any organisation requires [SGD03]. Since an ERP is a generic system, its implementation in a particular organisation involves a process of customisation in order to align it with the specific needs of the organisation. The ERP capabilities are configured according to the information provided by the process definition, since it is the model that describes the requirements of the organisation. Therefore, process definition plays a key role in determining the appropriate information system (whether it is a WfMS or an ERP) that supports the running of the organisational business.

The Service-Orientation (SO) (also known as Web Service) landscape is yet another category that plays an important role when providing IT support for business processes. Technologies within this area are meant to support the implementation of business processes through web services. A web service can represent an atomic activity within a business process, a sub-process component of a larger business process, or even an entire business process. Furthermore, since these alternatives are complementary, a business process may be implemented such as the coordinated composition of several web services. Methodologies that allow web services to be derived from a business process model can be found in [Pv07] and [Erl05].

In any case, the design phase determines the business process model, which represents the requirements to be satisfied for providing the needed IT-support, whatever approach is taken. Since this thesis is mainly concerned with the modelling of business processes, the design phase deserves special attention.

2.1.1 Business process modelling

The aim of the design phase is to produce a model of the business process run by the organisation. As every useful model, it is an abstraction of the real business process. This model has to offer a simplified view of the business process which will be used not only in the subsequent phases of the BPM life cycle, but also as a communication medium between the actors involved in the design phase, who may not necessarily may have the same professional background. In fact, van del Aalst [van02] points out that the engineering of business processes is a complex activity that requires the participation of not only organisational and business experts, but also of information technologists (i.e. IT Managers and Developers). Juric and Pant [PJ08] go further and provide one possible structure (which has proven to be efficient) of the team in charge of the modelling process. Such a structure is composed of:

- a process owner,
- two persons to assist the process owner, coming from the same department,
- a process quality representative,
- a business process analyst (usually the modeller),
- an IT representative, and
- optionally, an external consultant

A process model is expected to contain all the relevant information about the business process. Since the relevant information and aspects to be considered can easily over complicate the business process, a method for organising the information to be placed within a model is required such that this problem is avoided. The notions of *view* and *diagrams* represent a means to organise the way the information is organised within a model [EP98]. A view is an abstraction that captures information about one or more specific aspects of the business process from only a particular viewpoint. Each view consists of one or more diagrams, each of which shows only one relevant component of the view. Examples of views that may be considered when modelling a business process are *data, functionality, resources* and *motivation* [Toh99, LZRT08, Zac99].

The reasons for creating a business process model range from the need to optimise the throughput for reaching the final product or business objective, to storing corporate knowledge. Having a

model of a particular business process helps employees to understand how the business goal is obtained, or helps them calculate the cost in reaching such goal [Sch98].

The process of modelling a particular business process starts when the organisation realises the need for automating or improving (either fully or partially) at least one of its business processes. The event that triggers the initiation of the process (usually) comes from the management sector of the organisation as a response to the strategic plan defined by the organisation's board director. The management then presents the requirements' business process along with a high level overview of the required steps to achieve the business process' goal.

The business process requirement elicitation phase is still manually practised in the industry and it involves several meetings and discussions until the business process modelling team (and fundamentally the analyst) fully understand what it is required [Ko09]. It is worth noting that people who are not part of management nor the business process modelling team but who have a good knowledge regarding the current or existing process that need to be automatised (or improved) should also be included in these meetings and discussions.

Once the business process modelling team feels comfortable with the input collected about the management's requirements, the modeller describes the requirements using a language that allows them to be easily interpreted. Thus, the language chosen to carry out the modelling of the business process plays an important role in this phase as it determines how understandable (for all the involved actors) the resulting model is.

2.1.1.1 Languages

A language is inherent to a modelling process. Whatever needs to be modelled, a language is required to generate the description of the targeted concepts. The design phase is not the exception, so that a language is required for carrying out the modelling of the business process.

Flow diagrams and block diagrams represent the first-draft of notations used to model business processes [PJ08]. Over the years, a number of standards and/or widely used approaches have emerged. However, not all the standard languages that appeared within the BPM landscape were designed modelling business processes.

Some of these standards were created to facilitate the portability of business process models across different *Business Process Management Systems (BPMS)* (known as *interchange standards*), whereas others were just meant to allow a deployed business process to be executed (known as *execution standards*). The XPDL [Wor08] language, which is aimed at facilitating the interoperability between workflow technologies, is an example of an *interchange standard*. BPEL [OAS07] is a language that is used for modelling of business processes using web services, providing a special emphasis on describing the interactions between the web-services that compose the process. BPEL is an example of *execution standard*.

The categorisation of current standards by both their features and the BPM life cycle has been proposed by Ko et al. [LLK09] as an attempt to bring clarity to the language in the BPM field. Languages that allow modellers to describe business processes and their possible flows and transitions in a diagrammatic way belong to the type of *graphical standards*, and represent those to be used in the design phase of the BPM life cycle.

The same authors (i.e. Ko et al.) classify UML Activity Diagrams (UML-AD) [Obj09b], Business Process Modelling Notation (BPMN) [Obj09a] and Event-driven Process Chains (EPC) [ARI09] into the category of graphical standards. Pant and Juric [PJ08] also identify the same languages as sophisticated notations defined for coping with the modelling process, which demonstrates the

relevance of such languages in the BPM field in general, and in the design phase, in particular. In an historical revision regarding BPM standard approaches, ter Hofstede et. al. [tvAR09] includes BPMN, UML-AD and EPC as well-known approaches for specifying business processes. However, it is worth noting that EPC is considered not for being a formal standard, but for having been around for over 15 years as it is the modelling language used within the ARIS environment [Sch00, Sch98].

There are also some languages, like Yet Another Workflow Language (YAWL) [YAW09a], that span both the design and enactment phases of the BPMN life cycle. Since YAWL brings a new notation to the BPM arena, within the context of this thesis, it is only considered as part of the design phase. Further details about EPC, BPMN, UML-AD and YAWL, from the perspective of the business processes being targeted in this thesis (i.e. DCTC business processes), are given in Chapter 3.

Almost all languages belonging to the *graphical standard* category[1] lack formal semantics in their definition. This represents a major problem when there is a desire for some kind of analysis to be performed over the business process model before its deployment and enactment. This has led people to either give formal semantics to the business process modelling language of interest, or to use directly an existing formal notation for specifying the process. Examples of assigning a formal semantics to BPMN can be found in [DDO08, WG08, WG09, PW06, AKM08], whereas Petri nets [Pet96] is the usual formalism used to model directly business process [van02].

As pointed out in [LLK09], formal languages like Petri nets, pi-calculus [Mil82] or CSP [Hoa85] define the BPMN theory on which BPMN standards are based. This leads towards the interpretation that formal languages should be used as means to assign semantics to BPMN standards instead of being used directly to model business processes. One of the main reasons supporting this thought is that graphical notations like UML-AD and BPMN are easy for non-technical users to understand and use. Therefore, "easiness", "understandability" or "formality" are relevant notational characteristics that may make ease (or make difficult) the work of the modeller when it comes to describing the concepts of interest. The next section inspects the desirable characteristics a business process modelling language should contain.

2.1.1.2 Quality attributes

In the IEEE Standard Glossary of Software Engineering Terminology [Ins90], the term *quality* is defined as *"the degree to which a system, component, or process meets specified requirements"*. The "system, component or process" that is under consideration in the context of this work is the process definition (i.e. the model) that describes certain business process. The "requirements" are those expected properties, features or characteristics that such a model should contain. Despite the fact that the "requirements" vary with the needs and priorities of the person requiring such model, there are some characteristics that any model should contain as these affect directly its quality. These characteristics are called *quality attributes*. The fact that a particular model contains or does not contain them depends considerably of the person in charge of writing the model. However, the language used for modelling plays a key role in facilitating the models that do contain these desirable attributes. A modelling language composed of a particular quality attribute will help in writing models that need such a quality attribute. According to [Bt98, FHL+96, Har97, Rus07] the following quality attributes are important to be included into any business process modelling language:

[1] YAWL is an exception.

- *Conceptuality:* the International Standards Organisation (ISO) in its standard ISO/TR-9007 [Int] defines a model adhering to the "conceptualisation principle" as one that includes only the relevant aspects of the universe of discourse. Furthermore, such aspects are expected to be platform independent in the sense that they should not hold any kind of technology or implementation-related information. A modelling language with constructs that allow for the capture of the central concepts of the problem domain at the adequate abstraction level would hold such quality attribute. In any case, it is worth noting that conceptuality is a subjective quality: while some modellers agree that the concepts of the domain of interest are the correct constructs to capture (as long as they are at the right abstraction level) others modellers might disagree.

- *Suitability:* this attribute identifies the ease with which a concept and its interrelationship with other concepts can be captured within a model. Thus, a modelling language suitable to a particular domain of interest is one that holds features that allow both concepts and the ways they are related to each other to be captured and represented in the same manner in which they appear in the domain of interest. This direct representation of concepts and relationships would simplify not only the modelling process, but also the understanding of the model's content (although this is considered as yet another quality attribute called *comprehensibility* -see further). Suitability is a subjective quality since the assessment depends on who writes (or reads) the model.

- *Expressiveness:* the "100% principle" mentioned by the ISO in its ISO/TR-9007 [Int] states that a model should include all relevant information concerning the problem being tackled in the domain of interest. This quality attribute then is concerned with communicating all the different concepts than can be of interest for the problem at hand. As opposed to *suitability*, this attribute focuses on the general means provided by the modelling language such that a concept not bound to any domain can be always expressed somehow. Expressiveness is an objective attribute: let L_{bp} be a business process modelling language that provides the different constructs $C_1, .., C_n$, it can be proven whether or not a particular business process can be modelled.

- *Formality:* a language that has both its syntax and semantics formally defined (i.e. they are grounded on mathematical bases) is considered as *formal*. A language that has its syntax described in a formal way, but its semantics given by statements written in a natural language is considered to be *semiformal*. Languages that are not formal nor semiformal are considered to be *informal*. This quality attribute thus determines the precision with which both the syntax and semantics of a particular modelling language are given. Obviously, the most desirable approach is to rely on a formal modelling language for describing business processes as its use eases the process of creating consistent and unambiguous models. Formality is an objective attribute: the way the syntax and semantics of the language are given determine the level of formality.

- *Enactability:* Rusell claims that *"ultimately, there is no benefit in proposing a business process modelling language that is not capable of being enacted"*. Such a claim indicates that modellers look for languages that allow them to run models such that their correctness can be validated before going further with any tasks that take them as input. Therefore, this quality attribute determines the degree to which a particular modelling language facilitates the execution of the models it allows to describe. It is worth noting that running a business process model is not only useful for validating its correctness, but also for analysing its current performance for optimisation purposes. Enactability is an objective attribute

since it is determined by the degree of formality with which the semantics of the modelling language is given.

- *Comprehensibility:* a model does not make sense if it can only be understood by the person who wrote it (although it also happens that, after writing the model, its creator is also unable to understand its meaning). A model then has to be written such that the information it carries allows any other stakeholder to have the same interpretation as the individual/individuals who wrote it. Despite there being no guarantee that a modeller will come up with a comprehensible model for the particular problem at hand, a modelling language can be thought to ease this requirement with just such a desirable attribute. Comprehensibility, thus, is the ease with which models can be understood by stakeholders. Such an attribute is very subjective as it depends on the skills, experience and taste of the reader.
- *Maintainability:* a model represents an abstraction of a real fact. In our context, models describe business processes. It is accepted without question that business processes change very often because the organisations that run such processes change their needs continuously to adjust to the business environment. The ease with which a model can be modified to meet a new requirement or corrected when a mistake is discovered determines the maintainability of the model. It is important to notice that new requirements might necessitate modifications at the level of the modelling language (e.g. a new construct has to be added or modified), so that both the language and its associated tools must also be maintainable. Despite the maintainability of a model, it is always arguable, that there are good accepted practices (e.g. information hidding, modularity, low coupling/high cohesion between modules, etc.) that can help assess the maintainability attribute.

2.1.1.3 Assessment criteria

In order to evaluate the achievement of certain quality property, a *quality assessment criteria* must be defined. A quality assessment criteria is defined [CLV02] as a set of explicit rules and conditions, which are used to evaluate one or more quality properties of a modelling language. Different quality criteria have been reported as means to assess business process modelling languages.

Rusell [Rus07] relies on the notion of *workflow patterns* [vtKB03] to assess the *expressiveness* of business process modelling languages. The more workflow patterns a particular modelling language supports, the higher its power of expression.

Vasko et al. [VD06] use the *functional, behavioural, informational, organisational,* and *operational* aspects[2] identified by Jablonski et al. [JB96] to assess the *comprehensibility* of workflow modelling languages. Each aspect is evaluated separately. The functional aspect, that is concerned with the activities a particular process must perform and how these activities are structured, is evaluated by analysing the possibilities the language provides for nesting activities. The other aspects (except the *operational* aspect[3]) are evaluated by analysing the support provided by the language to model the *workflow patterns* as defined in [Rus07].

Hommes et al. [HR00] assess the quality of a business modelling language by means of the following quality properties: *suitability, completeness, coherence, expressiveness, comprehensibility, arbitrariness, effectiveness* and *efficiency.* These quality attributes are measured by

[2] It is worth noting that such aspects are claimed to be widely accepted in the workflow technology area [HOP05].

[3] At the writing time, we are not aware of any available methodologies for evaluating this aspect.

extracting the information provided by the *Modelling Concept Table* and *Model Table*. The *Modelling Concept Table* contains those modelling concepts that constitute an individual model M_i. In this table, each modelling concept is described by its name, meaning, and notation. A fourth column in the table contains the model that describes the relationships among the different modelling concepts (i.e. M_i meta-model -called MM_i). The *Model Table* lists the different kind of models (i.e. $MM_{1..n}$) and their goals (i.e. what are they meant for). A third column in the table provides the model that describes the relationships among these different kind of models (i.e. the meta-model of $MM_{1..n}$). For example, a model is claimed to be *coherent* if there are not isolated parts in its metamodel (fourth column in the *Modelling Concept Table*).

Aguilar et al. [ARGP06] assess a business process by its *maintainability*. This quality attribute is evaluated by measuring the structural complexity of the business process models. The metrics used for evaluating the structure of the models are adapted from those used to measure the structural complexity of software processes [GPR+06].

List et al. [LK06] evaluate the *expressiveness* of business process modelling languages by means of a generic meta-model that captures a wide range of business process concepts. These concepts are categorised in a similar way to the approach proposed by Jablonski et al. in [JB96]. Business process modelling languages with means to describe the concepts contained in the meta-model are considered expressive enough to model a very broad range of business processes.

Ortiz et al. [OHNAEE+07] also assess a business process modelling language by means of its *expressiveness*. The evaluation is made based on a set of concepts which authors claim to be the key elements any business process should be made of. Such concepts are also categorised by levels in accordance with the different perspectives by which a business process can be viewed. A quantitative metric is given to evaluate the presence of each concept in certain business process modelling languages. The higher the score of the modelling language, the more expressive the language is considered to be.

Gruhn et al. [GL07] adopt metrics used for analysing the complexity of software programs to assess the control flow of business process models. These metrics measure the effort to understand a model as well as several of its structural aspects (i.e. modularisation, nesting depth, and crossed reference with other models). Since such metrics are useful to evaluate the *structure* and *legibility* of a model, according to the categorisation of quality attributes proposed by Boehm [BBL76] it can be concluded that, the characteristic that is actually being assessed is the *maintainability* of the model.

2.1.2 Business process validation

Once the modeller has reached a business process model (or process definition), two kinds of problems regularly appear:

1. the process definition is neither syntactically nor semantically correct, and/or
2. the information held by the process definition does not actually capture the intentions of the stakeholders.

The first kind of problem reveals that the process definition *is not right*, whereas the second problem indicates that the process definition *is not the right one* from the stakeholder's viewpoint. Both problems are concerned with the *correctness* of the process definition and they

are identified when the process definition is subjected to analysis. The analysis of the process definition is addressed through verification and validation [JB96].

In the IEEE Standard Glossary of Software Engineering Terminology [Ins90], the term *verification* is defined as *"(1) the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase, (2) formal proof of program correctness"*. *Validation* is defined as *the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements*, where a requirement represents a condition or capability needed by a stakeholder to solve a problem or achieve an objective. Looking at these definitions from the perspective of a modeller in charge of providing a process definition, *verification* is the activity (process, or means) that allows problems of the first type to be either detected or shown to be in absence. On the other hand, *validation* is the activity (process, or means) that deals with problems of the second type as it allows the modeller to check whether the process definition captures the stakeholders' requirements.

It is thus important not only to verify the process definition before going further in the BPM life cycle, but also to validate it in order to assess its overall correctness. The modeller then, during the design phase, has to iteratively model the right process definition (i.e. the model is syntactically and semantically correct) while ensuring that the process captures the stakeholders' requirements. Notice that this way of proceeding is not only required by the BPM life cycle, but also by every development process that is driven by models. The determination of whether a model is correct or not, i.e., model verification and validation, is part of any development process that relies on models to obtain the final product.

Within the context of this thesis special attention is given to the validation of the process definition, as the major concern is (along with the modelling of DCTC business process) to provide a means to ensure that the stakeholders' requirements have been captured. For the reader interested in business process verification, a list of existing verification techniques is provided by Chen et al. in [CY06].

A way of validating a model consists of observing the dynamic behaviour of the system described by the model in order to check whether it conforms to the intuitive behaviour of the ideal system that the stakeholders have in mind [GJM02]. Observing the dynamic behaviour of the modelled system may consist of (1) providing an interpreter for the language in which the model is described and (2) executing the model by providing some input data. This manner of observing the dynamic behaviour of a modelled system is also called *simulation*.

It is worth noting that the simulation of a model requires, first instantiating the model, and next executing scenarios using some input data. This procedure indicates that one simulation of a model showing that it does not behave as expected by the stakeholders is enough to conclude that the model is incorrect. Conversely, a successful simulation of the model on a finite number of cases is not proof of a valid model in the general case. This conclusion stresses the fact that the absolute validity of a model cannot be obtained from simulating a model [DG98].

Nevertheless, simulation has proven to be a useful means for obtaining better insight into the operation of the process being modelled as it is usually the only tool available in workflow management systems[4] to perform analysis [van02]. Furthermore, Ren et al. [RWD+08] argue that, under complex business scenarios, simulation is the only technique that enables modellers to analyse not only the behaviour of the modelled business process, but also its performance over time providing better insight regarding bottlenecks and hand-over times.

[4] Known as the early generation of *Business Process Management Systems* (BPMS) [van04].

It is worth noting that not only the business process simulation, but also the analysis of both the required input data for performing the simulation and the output data generated by the simulation improve the knowledge about the process itself. This sometimes leads to valuable insight for the re-design of the business process. In case an existing version of the business process is already deployed and running, its actual behaviour can be recorded and then used as input data for performing simulations of the re-designed version. The technology that uses event logs to analyse a business process is called *process mining* [tvAR09]. In this manner, process mining represents a complementary technology for simulating business processes. However, as was already stated, the use of process mining depends on the existence of an earlier version of the business processes under execution. This is the reason why process mining is primarily used for the diagnosis of existing operational business processes. Diagnosis represents the fourth phase in the BPM life cycle.

In the case that the operational behaviour of the model is displayed graphically as it moves through time, the simulation is said to be *animated*. Otherwise, every model simulation (at least) has to produce an *execution trace* that can be subsequently analysed in order to determine whether the model captures the expected requirements.

A different way of validating a model consists of analysing the properties that can be deduced from the model, and comparing them to the properties the ideal system is expected to contain. Examples of how this approach can be used to allow a more precise understanding of the model can be found in [KPR04, FLM+04].

Other techniques used to perform model validation (which are not confined to the business processes domain) are listed by Sargent in [Sar00]. It is worth mentioning that the author in that work emphasises the fact that a combination of techniques is generally used to validate a model.

2.2 Dependability

2.2.1 Dependable computing

The idea of a dependable business process, as introduced in Section 1.1, relies on concepts and principles that come from the dependable computing area [IFI]. In the software engineering literature, the term *dependability* is defined as *the ability to deliver service that can justifiably be trusted* [ALRL04]. Such a service is delivered by a system: i.e. an entity made of hardware, software and humans. In the business domain, the service is the business process, whereas the system is the business organisation that owns such a process.

It is assumed that systems are not perfect, i.e. they are expected to fail from time to time. A system fails (aka system failure, or simply **failure**) when there is a service failure: i.e. the delivered service is judged (by a particular judgemental entity [RK07]) as being different from its intended state. As system failures are unavoidable, the challenge is to reduce their frequency and severity. A dependable system, thus, is one that has the ability to avoid service failures that are more frequent and more severe than is acceptable from the judgemental system's point of view. The term "judgemental system" covers concepts ranging from failure detectors implemented in hardware or software to the legal justice system. Furthermore, since the judgemental system is itself a system, it might also fail (as judged by another higher level judgemental system). In this work, as will be demonstrated later, the role of judgemental system is played by the stakeholders that request the service that is to be provided. An **error** is the part of a system

state that might lead to a failure. The hypothesised cause of the error is a **fault**. An error does not necessarily lead to a failure as it may be avoided by chance or design, or simply because it does not constitute a fault for the enclosing system.

There exist four general mechanisms to achieve dependability [ALRL04]: fault prevention, fault tolerance, fault removal, and fault forecasting. *Fault prevention* deals with the objective of avoiding the introduction of faults during the software development process. As such objective is a part of the general aim of every software development methodology, fault prevention can be considered as an inherent part of it. Thus, good practices in software development (e.g. modularity with low coupling and high cohesion, information hiding, use of strongly typed languages for the specification, design and implementation) help in reducing the number of faults when developing a system. *Fault tolerance* is aimed at allowing the system to provide the service in spite of the presence of faults. The basic activities required to achieve fault tolerance are error detection (i.e. to identify the presence of an error) and system recovery (i.e. to lead the system to a well-defined state without detected errors from which the system can continue its normal execution). *Fault removal* deals with uncovering faults that have occurred at any phase in the development process. Activities covered by this mechanism range from checking the specification of the system to uncover specification faults up to exercising the system (i.e. testing) to uncover development faults. *Fault forecasting* is aimed at evaluating the behaviour of the system under the occurrence of faults such that it can be concluded which ones would lead to system failure.

The effectiveness of each mechanism depends on the context and nature of the fault. Thus, the dependability of a system can be increased by a combined use of these mechanisms. It is worth mentioning that a totally dependable system (i.e. a perfect system) is impossible to achieve, but it should be the objective that any development process must try to attain. This work takes the view that fault tolerance is a means to achieve dependability, complemented with the other existing ones (i.e. fault prevention, fault removal, and fault forecasting). Based on the assumption that faults cannot be fully avoided or removed, the choice is to enrich the system with means to detect erroneous system states and then to perform the necessary recovery steps that lead the system to a well-defined state (fault tolerance view). Both the error detection tools and the recovery steps are made part of the model that describes the system. This model is produced during the analysis phase of the development methodology that is being followed (fault prevention view). The model not only describes the functional aspects of the system, but also the fault tolerant ones. Such model can be used to perform an early evaluation of the system behaviour with respect to its adherence to the expected functional properties as well as with respect to the occurrence of faults (fault forecasting view)[5]. If such early evaluation reports that the system model either does not adhere to certain functional property or does not behave as expected when facing a particular fault, then the model has to be corrected because it is faulty (fault removal view). This early evaluation is carried out until the model fulfills both the functional and fault-tolerant aspects. Once this point is reached, the next phase of the development process (i.e. design) begins.

2.2.2 Fault tolerance

Fault tolerance is achieved by error detection and subsequent system recovery. There are two kinds of error detection techniques: concurrent and preemptive [ALRL04]. Systems that allow errors to be detected during the delivery of the normal service are said to support *concurrent*

[5] Modelling and simulating error detection and recovery activities could be used as an effective method to estimate the consequences of a fault.

error detection. Those that detect errors only while running in specific modes or at a particular period of time (e.g. audit and start up, respectively) are said to support *preemptive error detection*.

The purpose of the recovery phase is to lead the system back to a certain state such that it can continue executing. This can be achieved by modifying the system state such that it does not contain errors. This technique is known as *error handling*. It can be implemented by backward error recovery (sometimes called rollback), forward error recovery (sometimes called rollforward), compensation, or any combination of these. Backward error recovery returns the system to a saved state that existed before the error occurred. This state is assumed to be correct since in the past it allowed the system to be fully operational. Implementations of this technique include checkpoints [EAWJ02], conversations [Ran75], and transactions [GR92].

Forward error recovery is aimed at leading the system towards a new (i.e. not reached recently) correct state. Reaching such a state is only possible when there exists precise knowledge about the kind of error that has corrupted the system. When the class of error is known, specific activities meant to deal with this particular error are performed. By executing these procedures, reaching the correct new state should be possible, thus, allowing the system to resume its operation either as before the error was detected (best case scenario) or in a mode where not all its services are available (aka degraded mode). Forward error recovery is usually achieved by using exception handling mechanisms (EHM) [Cri89, BM00] as they embody concepts (i.e. exception and handler) and capabilities (e.g. detection, control flow transfer, exception and handler categorisation) that make this type of error recovery mechanism easy to implement.

Compensation aims to allow the system execution to progress as expected despite it being in an erroneous state. This technique is implemented under the assumption that a system erroneous state holds enough redundant information to allow the error to be masked. Hardware redundancy includes supplementary and potentially similar hardware in the system, whereas software redundancy includes additional components, i.e. programs, objects or data. Software redundancy is complemented with software diversity to solve the problem of replicated design and implementation faults. N-version programming [Avi85] and recovery blocks [HLMSR74, Ran75] are the original and basic techniques that implement software diversity.

Error handling techniques intended to cope with errors such as those previously described (i.e. rollback, rollforward, and compensation), leave the fault untreated. Thus, for a fault that has already led the system to an erroneous state there is clearly a possibility that the fault will continue to produce errors. As the repeated manifestations of a fault can make a system fail despite the efforts of the fault tolerance technique it implements[6] it may be necessary to eradicate the fault from the system.

Techniques aimed at removing the fault from the system to avoid it being reactivated are part of the final phase of fault tolerance. This phase is known as *fault handling*, and it implies identifying the fault (i.e. diagnosing), isolating the faulty element (e.g. component, module, class), reassigning the tasks performed by the faulty element among non-faulty elements (i.e. system re-configuration), and restarting the system.

2.2.2.1 Fault tolerance in distributed real-time systems

As previously stated, the approach to achieving dependability in business processes is centred around fault tolerance. Mechanisms or tools will have to be used in business processes that

[6] Either because the consequences of the fault become more and more serious, or because the system cannot longer provide its service as expected due to the overheads of dealing with recurring errors[AL81].

are collaborative and hold timing constraints. Thus, it is logical to explore how fault tolerance has been successfully applied in the computing domain when engineering "collaborative systems with timing constraints" (known as distributed real-time systems in the computing field).

A "distributed system" [BW01, Lam78] is defined as a system composed of multiple autonomous processing nodes cooperating toward a common purpose or toward achieving a common goal. These autonomous processing nodes communicate with one another by exchanging messages. It is a necessary condition for a distributed system that the message transmission delay is not negligible compared with the time between events in a single processing node (multiprocessor computers are excluded as the message transmission delay is negligible). This definition is compatible with our way of considering collaboration in business processes.

A system is "real-time" [Bur91] if at least one of the computational tasks that it executes is constrained somehow by time (compatible with our way of considering time constraints). Such a definition indicates that the correctness of the result provided for the task depends not only of its logical value, but also on the time at which it is provided. These timing constraints appear in the requirements specification in the form of deadlines. This definition is compatible with our way of considering timing constraints in business processes.

Distributed systems (being real-time or not) have the partial-failure property: the occurrence of a failure (it does not matter of which kind) usually affects only a part of the system [Tel94]. Fault tolerance techniques exploit this property to coordinate the execution of the distributed system so that non-faulty processing nodes can take over the activities of those that are failing.

Fault-tolerant algorithms based on replication (i.e. systematic compensation for fault masking) are an option since (potentially) every processing node can be used for redundancy purposes. Every system component that needs to be replicated (nothing forbids to replicate the entire system) can be deployed on one of the processing nodes that compose the distributed system.

There also exist fault-tolerant algorithms designed to insure the correct behaviour of the system (while certain conditions hold) despite failure occurrences [LSP82]. Still others are meant for identifying the kind of failure (even in the case of multiple occurrences) to be able to perform the necessary recovery actions that will lead the system to either its normal behaviour (best scenario) or a graceful degradation, instead of reaching an overall malfunctioning [CR86]. Such fault-tolerant algorithms have to coexist with scheduling algorithms that take care of the temporal behaviour of the distributed system, when timing constraints need to be satisfied. In this scenario, a trade-off between dependability and performance must be made.

Fault tolerance techniques can be also applied within each processing node that is part of the real-time distributed system. The advantage of this technique is an increase in the dependability of the local computations carried out in the processing node. One method for achieving this is to combine exception handling principles with scheduling practises for providing fault tolerance by means of forward error recovery [dALB05]. This approach categorises processes or activities as primary or alternative tasks: a primary task is one whose execution is required in error-free scenarios, whereas an alternative (i.e handler) is one that must be executed only when some error is detected (i.e. exception is raised).

Such a categorisation helps in scheduling the tasks. Since an alternative task is expected to run less often than a primary one, different priorities can be assigned to them. It might be decided, for example, that alternative tasks run with higher priorities as a way to increase the system's tolerance for detecting errors.

Another alternative, is to use the transaction processing paradigm for executing the timing constrained tasks within a processing node. Turning every task into a transaction allows fault

tolerance to be provided by means of backward error recovery (atomicity property - the A of the ACID[7] properties granted by the transaction processing paradigm). However, the transaction processing mechanism (whatever it is) embedded into the processing node has to include scheduling aspects so that the number of transactions missing their timing constraints are minimised [AGM92]. Advanced transaction mechanisms, like the one introduced in the next section, can be used when some of the ACID properties need to be relaxed, while keeping others in tact. This is the case for long-running transactions (i.e. transactions intended to run over long periods of time) that maintain consistency and durability (C and D), while they relax atomicity and isolation (A and I) [Gra81].

2.2.3 Coordinated Atomic Actions

Coordinated Atomic Actions [XRR+95] represent a fault tolerance conceptual framework[8] used to increase the dependability of distributed and concurrent object-oriented (OO) software systems[9].

The abstract concurrent OO computation model, which the coordinated atomic actions conceptual framework (CaaFWrk) is based, is defined as a collection of interacting objects (which may or may not be distributed) where the processes (or threads) executing concurrently (i.e. these processes may, but need not, overlap [BA06]) correspond to the executions of operations (which may be associated with a single object or several different ones) on a group of objects. Those objects that own the operations concurrently executed can be considered as *active* objects, whereas the objects on which operations are applied can be considered as *passive* objects. The conceptual framework does not make a distinction as a particular object can behave both as active and passive during the software system execution. What the conceptual framework does assume is that each object executes just one of its operations at time.

The kernel of the CaaFWrk is an abstraction, which is defined as a generalised form of the *atomic action*[10] concept. This abstraction allows a set of concurrent processes to perform a group of operations on a collection of objects. Therefore, it represents an atomic logic unit (i.e. indivisible and with well-defined boundaries) for the execution of a set of concurrent processes. Furthermore, the atomicity, consistency, isolation and durability (ACID) properties are ensured for those objects being accessed by the processes forming part of the atomic logic unit.

This atomic logic unit works also as a damage confinement area. It constrains the spread of errors to its enclosing context. This is achieved by allowing recovery procedures to be associated with each atomic logic unit. Exceptions and exception handling features are used to identify the presence of an error and eliminate it by putting in place one of the recovery procedures associated with the atomic logic unit being executed. Therefore, exception handling forms the basis of the CaaFWrk to implement fault tolerance. In the context of the CaaFWrk, this atomic logic unit is known as a *coordinated atomic action*[11] (CAA), the concurrent processes are the *participants*, and the set of operations that each participant performs inside the CAA is known as the *role*.

The CaaFWrk then can be seen as a tool that designers may use for structuring the software system activities (design) in order to meet the user's requirements (specification). Concurrent

[7] The acronym ACID means: Atomicity, Consistency, Isolation and Durability.
[8] A technique with its own terminology and strategy to implement fault tolerance.
[9] The expression *software systems* is used to mean the piece of software that satisfies the user's requirements along with the hardware and environment where it is deployed.
[10] Abstraction that allows the execution of a set of operation to be seen as only one indivisible operation.
[11] The conceptual framework was named after the coordinated atomic action.

OO software systems with high levels of dependability belong to the domain set for which the CaaFWrk was meant to be used as a design tool.

Real-time software systems are (either inherent or imposed) concurrent and very often have dependability requirements ([BW01], pages 7-12). Thus, these types of software systems are first-class candidates to be designed using the CaaFWrk. However, the timing requirements imposed by most of the real-time software systems are not possible (or, at least, not easily) to be modelled by the conceptual framework. Therefore, extensions to make it available for designing real-time software systems are required.

An initial attempt was proposed by Romanovsky et al. in [RXR99][12]. In this work, the CaaFWrk was extended to allow timing constraints to be placed over a CAA abstraction. These timing constraints then apply over the set of concurrent process that come together to perform the CAA. The violations of any of the timing constraints imposed at the CAA level are seen as exceptions. This leads to situations in which timing and value exceptions can be raised concurrently. How the exception handling mechanism copes with such situations was the main focus of the Romanovsky et al. paper.

The description of the CaaFWrk along with its time extensions are described and analysed in depth in Chapter 4 as it is the chosen semantic domain for providing meaning to the expressions of the *DT4BP* business process modelling language introduced in Chapter 3.

2.3 Model-driven language engineering

2.3.1 Model-driven engineering

Model-Driven Engineering (MDE) refers to a software development approach in which the software system to be developed is initially described by an abstract model and systematically transformed into a concrete implementation [FR07]. The MDE's goal is to help developers reduce the gap between the problem domain level concepts and those that are at the implementation level and which are used to implement the problem domain level concepts. Current efforts to bridge this gap are difficult and costly, not only because these efforts require intensive error-prone hand-made activities at the implementation level, but also because of the complex requirements imposed onto the software (e.g. run over multiple distributed platforms, adaptable to the constant changes of the environment where they are embedded, and behave in a dependable manner) as well as the growing pressure to reduce cost and time to market[13].

MDE attempts to solve this problem by raising the abstraction level at which software is developed, while automating the generation of the final source code using technologies that support systematic transformation of problem-level models (e.g. requirements model) to software-detail models (e.g. source code). The Eclipse Modelling Framework (EMF) [BBM03] is an example of a technological framework intended to support MDE.

It is worth noting that MDE may also be referred to as *Model-Driven Development (MDD)* [AK03, BCT05], *Model Engineering* [Béz04], or *Model-Based approach* [GRS08]. However, the term MDA, which stands for *Model-Driven Architecture* is very often used (wrongly) as a synonym of MDE. MDA [Obj01] is an initiative that adheres to the MDE approach (i.e. MDE encloses MDA) launched by the industry-driven organisation OMG [Obj] in its role of provider of standards for

[12] Similar and extended information can be found in [RXR98, BRR+98].
[13] The length of time it takes from when a software is concieved until it is available for sale.

the software development community. MDA exploits the notion of model by introducing the concepts of *platform independent model (PIM)* and *platform specific model (PSM)*, and the different kinds of mappings these models might involve. While a PIM focuses on those system aspects that are not related to any particular technology, a PSM is the model that defines "a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns" [FR07]. Thus, the approach is to start with a PIM description of the software system, which it is to be refined (PIM to PIM transformation) until it is realised in a platform-specific way (PIM to PSM transformation) and eventually transformed into source code (PSM to PSM transformation). These transformations are expected to be fully or semi-automatised, as advocated by the MDE approach, in order to improve the productivity while simultaneously reducing the complexity of integrating different technologies [Ken02, AK03].

The use and creation of models above the code level is the focus of the MDE, since it is the mechanism for raising the abstraction level at which software is developed. Models thus are the first-class entities being manipulated (i.e. analysed, simulated, transformed) during the mode-based process. The required support for properly manipulating models is also a part of the duties enclosed in the MDE landscape. This support goes from tools for editing, analysing or simulating models, to technologies for extending a modelling language to allow modellers to capture concepts as they appear at the domain level. Moreover, the fact that models are transformed into another models, may lead to the existence of multiple modelling languages within the same model-based development process. In this manner, mechanisms to create and use models having a special emphasis on modelling languages, play a very important role within the MDE field [Kle08].

Since the BPM field has as the ultimate goal providing IT support for the execution of the business processes, it can benefit from the ideas brought by the MDE field. Perez et al. [PRP08] have reviewed the different proposals for exploiting MDE within the BPM. Such systematic reviews have determined that most of the works in this domain point the use of MDE as a valid approach for BPM. The definition of the BPMN language [WM08, Obj09a] from part of the OMG, is additional evidence of the interest of applying MDE in the management of business processes. This notation was engineered to bridge the gap between the format of the initial design of the business processes and the format of the languages that will execute these business processes (e.g. BPEL [OAS07]). Information and examples regarding how BPMN maps into BPEL can be found in the Annex A of the BPMN specification [Obj09a] and in the work of Giner et al. [GTP07].

2.3.2 Modelling language specification

A modelling language is the vehicle (means, or medium) that allows the modeller to write the models that let him reason about a problem within a particular domain. A modelling language (as every language in general, whether it is natural or artificial) consists of a *syntax* and a *semantics*.

Considering a model as made up of a finite sequence of symbols taken from a finite alphabet, the set of lexical rules that decide whether a model is valid or not defines the syntax of the language. On the other hand, the semantics of the language is the set of rules that give meaning to the finite sequence of symbols (i.e. the model) by relating the syntax to a *semantic domain* [HR04]. The semantic domain is a well-defined and well-understood domain on which the language engineer relies to provide meaning to each expression of the language being defined. In this manner, every valid syntactic expression must be bound to an element of the semantic domain such that the

meaning of the syntactic expression is precisely defined (in terms of the semantic domain). The set of bindings between each syntactical expression and its semantic domain element is known as *semantic mapping*.

It is worth observing that the syntax of a language usually consists of both *concrete* and *abstract* parts. The lexical rules that define the concrete syntax specify the permitted sequence of symbols that a model must have in order to be considered as valid, as well as the unique structure or parse tree (i.e. it is not ambiguous) that is assigned to each sequence of symbols. Conversely, the rules that define the abstract syntax specify the structure of the expressions allowed in the language. This means that these rules are not concerned with an acceptable sequence of symbols nor with assigning to each such sequence a unique structure or parse tree. Instead, the abstract syntax rules specify the set of allowable parse trees for the language [AU77, Hen90, App98].

Both the (concrete and abstract) syntax and the semantics (i.e. semantic domain and semantic mapping) need to be described. For textual languages, both syntaxes are described using context-free grammars. A grammar is a tuple of four elements $< T, NT, S, P >$, where

- T is the set of basic symbols (called *terminals*), which the sequences that define the model are composed of,
- NT is the set of special symbols (called *non-terminals*) used to denote sets of symbol sequences such that each of these sets determine a particular syntactic category,
- S is one particular non-terminal selected as the *start symbol*, and
- P is a set of *productions* (or rewriting rules that define the manner in which the syntactic categories determined for each non-terminal can be built up from one another and from the terminals.

A grammar is context-free when each of these production rules has a single non-terminal symbol on the left-hand side. The Backus-Naur Form (BNF) [BBG+63] and its extended versions are the most popular grammar-like notations [KLV05] used by language designers to define the (concrete and abstract) syntax of textual languages.

Kleppe [Kle08] identifies other kinds of grammars that could be used when describing the (abstract rather than concrete) syntax of a language like *attributed* and *graph* grammars [14], while Klint et al. [KLV05] add *algebraic signatures* to this list of grammars formalisms (see [BM07] for an example).

Other alternatives for defining the syntax of a language include: (1) the profiling mechanism provided by the UML [Obj09b] and (2) the metamodelling principle. The UML profiling is a facility provided by the UML language to extend its definition for the purpose of introducing domain-specific concepts such that the modeller deals with concepts that are at the level of problem discourse. It must be observed that this mechanism only allows for the definition of the abstract syntax of the language as the concrete syntax remains the one provided by the UML language. On the other hand, the metamodelling principle allows for the definition ob both the concrete and abstract syntax of a language, so that it can be considered as more powerful than the UML profiling facility. The metamodelling principle is a model-driven language engineering process in which the parts of the language (mainly the abstract syntax) are defined in terms of (usually) UML class diagram models [Kle07, GRS08], although this is not mandatory to

[14] So far, it is too complex to be put in practice in the definition of a language.

adhere to the metamodelling principle. The KM3 language [JBA06] for example, is a textual language meant for supporting the definition of meta-models. Furthermore, there is evidence that modelling language engineers following the metamodelling principle do not always ask for visual languages to define the meta-models [RJK+06]. Please note, that both the UML profiling facility and the metamodelling principle are considered as enclosed in the MDE approach.

Regarding the description of the language semantics, the possible existing approaches are [AU77, Kle08, NN92]:

- *Operational (or interpretive):* the meaning of each possible statement that can be written using the language's constructs is specified by rules (or axioms) that determine the induced computation of such statement when it is executed on a particular abstract machine. The abstract machine is characterised by a *state* (i.e. variables with their current values), whereas the rules specify how the state is transformed by a statement written using the different language constructs. Thus, the abstract machine (or state) and the set of rules (or axioms) determine a *state transition system* that describes how the execution of each statement takes place in a given initial state. It must be emphasised that the state transition system encloses both the semantic domain (i.e. abstract machine's state) and the semantic mapping (i.e. abstract machine's transition rules).
- *Translational:* the meaning of each possible statement that can be written using the language's constructs is specified by giving rules, which associate each statement with one (or more) statements in a language whose semantics is already well-known and well-understood. In this case, the focus of the semantic definition of the language is given on the semantic mapping rather than on the semantic domain, since the latter (i.e. the semantic domain) is assumed not only to exist, but also to be well-understood, even when its own semantics is not mathematically defined (e.g. the *Java* programming language as defined in [GJSB05].).
- *Axiomatic:* the meaning of each possible statement that can be written using the language's constructs is specified by giving rules of the form $\{P\}C\{Q\}$ that relate the state before (i.e. $\{P\}$) and after (i.e. $\{Q\}$) the execution of the statement (i.e. C). The semantic domain (usually a logical system) is used to describe the conditions $\{P\}$ and $\{Q\}$, which define the correctness criteria for the statement C: the statement C is correct if whenever the initial states fulfils $\{P\}$ and C terminates, then the final state is guaranteed to fulfil $\{Q\}$. Notice that the way of describing each of the rules determine the semantic mapping between the language and its associated semantic domain (i.e. logical language used to describe $\{P\}$ and $\{Q\}$).
- *Denotational:* the meaning of each possible statement that can be written using the language's constructs is specified by giving mathematical functions that describe the effects (value of the function) of having executed such statements (arguments of the function). It is probably the semantic approach that best provides for the semantic mapping since every function is the precise description about the binding between a statement and its meaning according to the chosen semantic domain. Please observe that an important aspect of this semantic approach is the use of composition to define the semantics of the language. For example, the meaning of the statement $f(a$ '+' $b)$ is defined as the result of performing $f(a) + f(b)$, where the symbol '+' represents certain operator within the language being specified, and which is mapped onto the common mathematical operator $+$.

Combermale et al. [CCGT09] after having surveyed the different ways of formalising the semantics within the MDE context, concluded that the two principal approaches being considered are

the operational and translational. Kleppe ([Kle07], page 138) agrees with this conclusion and also emphasises that when having a good target language the translational approach is the most suitable.

Terms like *dynamic semantics* [RJK+06, JBC+06] or *behavioural semantics* [CCGT09, RDV09], which can be found in the literature, do not represent new approaches to describe the semantics of a language. These terms are meant to refer to a language in which the semantics is defined as a state transition system, but without giving details as to whether the chosen approach is operational or translational. In other words, both dynamic and behavioural semantics denote a concise way of indicating that the modelling language allows its models (i.e. every valid model that can be created using the modelling language) to be executed.

2.3.3 Metamodelling

Recall that the principal concern of MDE is to use models for raising the abstraction level at which the software is developed. The success of this process (in part) depends on the modelling languages used to support the creation and manipulation of the models. The key role of the modelling language within the MDE approach lies in narrowing the focus it gives to modellers to create and manipulate models using the jargon of the domain of discourse. Furthermore, modellers do not need to understand (or even know) the existance of the mapping that allows such a concepts to be transformed into the final implementation, since it is obtained by applying (semi)automatic systematic transformations.

Engineers providing a modelling language for the MDE community may choose between either an extensible general-purpose language, or defining a domain-specific modelling language (DSML) [FR07]. Engineers choosing the former option will need to use the extension mechanism provided by the general-purpose language to define domain-specific concepts. The UML language with its profiling mechanism is an example of an extensible general-purpose modelling language that can be used to define a DSML. On the other hand, engineers that want to define a domain-specific modelling language will use the metamodelling principles to implement not only the selected concepts of the domain, but also the tools that allow them to transform one model into another. The high-quality typesetting system LaTeX [LaT] is an example of such a domain-specific language.

As usual, every approach has its pros and cons. The principal advantage of using UML with its profiling mechanism to define a DSML is the variety of existing off-the-shelf UML tools (e.g. Magic Draw [No 10], Borland Together [Bor10]) that at the same time allow language engineers to ease the DSML definition and reduce the time-to-use as modellers have tools for dealing with models from the very beginning. The main disadvantage of the UML alternative is that the new DSML must respect the original UML syntax and semantics, which leads to the problem of not having the relevant or required components within the DSML, thus, contradicting the notion of domain-specific. The way of dealing with this drawback is to define a DSML by metamodelling principles (i.e. second choice). The price to be paid for this choice is the high initial cost in developing tools for supporting both the modelling and use of the language. Fortunately, tools for supporting the metamodelling principles (e.g. MetaEdit+ [Met10], EMF [BBM03], ATL [ATL10a], Kermeta [Tri10a] and QVT [Obj08]) have appeared in the last years, making the metamodelling principle the best choice when defining a DSML. Industrial experiences provide evidence for this claim [KTK09].

2.3.3.1 Model-based languages

As already indicated, the term metamodelling denotes the engineering of a (software) modelling language driven by the MDE approach. In metamodelling, a meta-model is an artifact used to define (part of) a language. The metamodelling's origin is related to the launch of the MDA framework (introduced in Section 2.3.1). It is also related to the fact that the OMG realised by the UML language was just one possible meta-model (among many) that could be used to apply MDA [Béz04]. To avoid the proliferation of multiple non-compatible meta-models, the OMG came out with the *Meta Object Facility (MOF)* [Obj06]: a global integration framework for the definition of meta-models, or in other words, a language for defining modelling languages.

The metamodelling principle brings with it the notion of "metalevels". A metalevel represents a conceptual space in a stratified architecture in which a particular model is defined. A model m^i defined in level i of the stratified architecture represents the meta-model for those models m_j^{i-1} (with $j = 1..w$) placed at the level $i-1$ that are defined using the concepts provided in m^i. The relationship *conformantTo* is used as a means to determine those models m_j^{i-1} placed at level $i-1$, which are properly described using the concepts defined in the model m^i at level i. The OMG instantiates this general stratified metamodelling architecture with four levels, placing MOF at the highest level (i.e. $i = 4$). MOF, as claimed in its definition (see [Obj06], page 11) is described using itself, which leads to the notion of self-conformance.

The statement that a meta-model defines "part of" a language refers to the *abstract syntax* of the language. As explained in Section 2.3.2, a language definition also involves the provision of a concrete syntax and semantics. These additional components comprise the definition of the language may or may not be given by a meta-model. Within the context of this thesis, a meta-model is only used as means to formalise the abstract syntax of a language, unless otherwise stated.

The UML class modeling notation is used to describe MOF compliant meta-models. Thus, the abstract syntax definition of a language will look like a UML class diagram. When the UML class modelling notation is not sufficient to describe the required relationship among the concepts of the language under definition, the *Object Constraint Language (OCL)* [WK03] is used. Please observe that the OCL constraints placed on a meta-model must also be considered when evaluating the conformance of a particular model with respect to its meta-model.

2.3.3.2 Model transformation

Model transformation is also a key part in the MDE, since it is the means to reach the fined-grain models (e.g. source code) that implement those models being described using domain-specific concepts. Within the OMG MDA's world, transformations represent the mechanism to refine the initial highest abstract PIM until reaching the most detailed PSM. Transformations within the MDE approach are not only useful for achieving model refinement, but also for refactoring (i.e. the same model is reorganised according to certain criteria) or for migrating (i.e. the same information is modelled using other notation).

A transformation then can be seen as a function that takes n input models (aka *source models*) and returns m output models (aka *target models*). The aim of a model transformation is to transform any valid set of source models into a valid set of target models. This statement merits two observations: first, the source models are assumed to be valid ones, whereas the transformation itself ensures that the resulting target models are valid; second, the model transformation is described in terms of the source and target meta-models [Kle08], because it is expected to be

general enough such that it can be applied over the full set of valid models defined by the source meta-model. Thus, the signature for a transformation T aimed at transforming n input models M_j^s $(j : 1..n)$ into m target models M_k^t $(k : 1..m)$ is the following:

$$T : MM_1^s \times MM_2^s \times ... \times MM_n^s \longrightarrow MM_1^t \times MM_2^t \times ... \times MM_m^t,$$

where MM_j^s is the source meta-model to which the source model M_j^s must conform to, and MM_k^t is the target meta-model to which the generated target model M_k^t will conform to.

There exist several ways of categorising transformations [Kle06, SK03, MV06, CH06]. Despite the fact that some of these criteria overlap in the characteristics considered to classify a transformation (e.g. number of source and target models or languages used to describe the source and target models), there is not a common comparison criteria that can be used to select the best transformation approach according to any specific need. In the context of this thesis, the classification criteria given by Kleppe in [Kle06] is very convenient, since transformations are categorised based on their relationship with the parts used to define a modelling language (i.e. concrete and abstract syntax, semantic domain, and semantic mapping). Thus, let L_1 and L_2 be two different modelling languages defined according to the metamodelling principle, then the transformations can be categorised as follows:

- *intra-language transformations:* are used either to (1) generate one part of the language definition part, or (2) represent a part of the language definition. A *syntax transformation* is an example of the first case as it can be used to generate the concrete (resp. abstract) syntax given the abstract (resp. concrete) syntax. A *semantic transformation* is an example of the second case, as it is used to define how the valid statements that can be defined using the abstract syntax concepts are mapped to the expressions in the semantic domain.
- *intra-model transformations:* are used to transform the same model. *Refactoring* and *view transformation* are examples that fall into this category. The former performs changes over the model according to certain criteria, whereas the latest shows only partial information (also selected according to certain criteria) owned by the model.
- *inter-model transformations:* are used to transform a model written in L_1 into another model written in L_2. Transformations meant to perform a mapping from a concrete syntax to another concrete syntax are called *stream-based transformations*, whereas those transformations that map an abstract syntax into a different one are called *structure transformations*.

Figure 2.1 shows the transformations listed above along with their relationship to the elements of the modelling languages L_1 (i.e. $Cn1, A1, S1$) and L_2 (i.e. $Cn2, A2, S2$). Please note that bidirectional arrows are used to indicate that a transformation can be defined in both directions, but not that it must be bidirectional.

Structural transformation and *semantic definition transformation* are of special interest within this thesis, as they are used when giving semantics to the domain-specific modelling language introduced in Chapter 3. ATL [ATL10a] and Kermeta [Tri10a] are the transformation languages used to implement each of these transformations, respectively. A detailed comparison between Kermeta and ATL can be found in [Wie10].

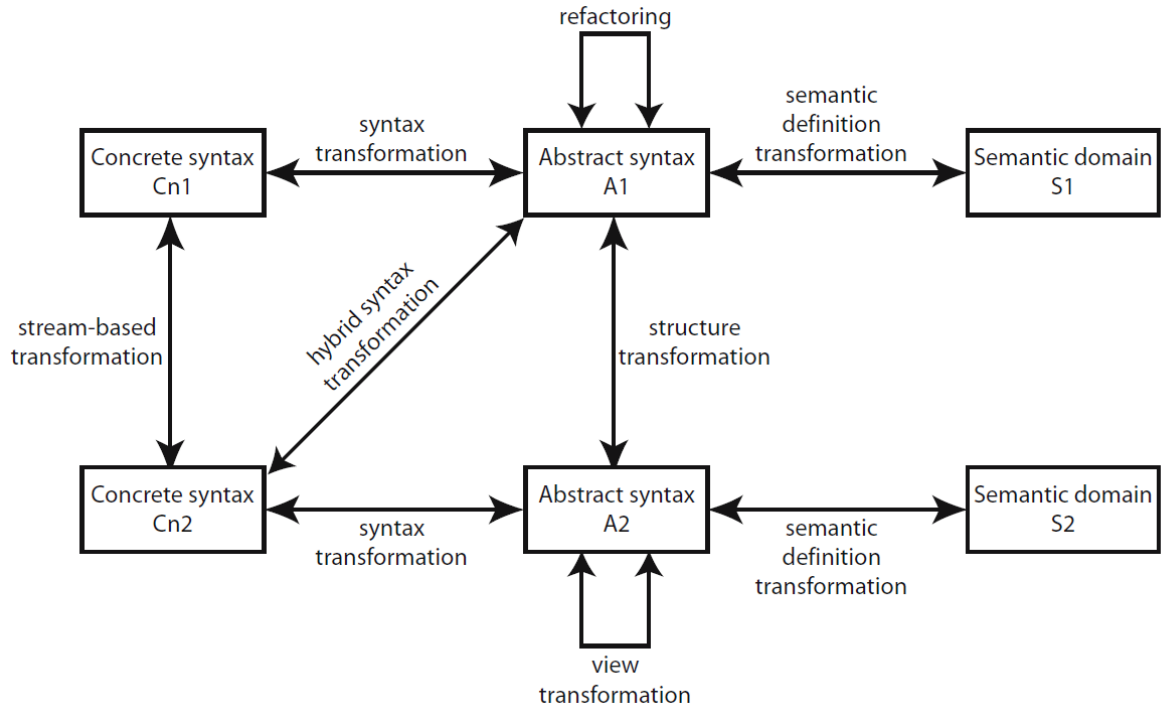

Fig. 2.1: The relationship between transformations and language definition (taken from [Kle06]).

3. THE DT4BP BUSINESS PROCESS MODELLING LANGUAGE

Abstract

This chapter describes DT4BP*, a novel business process modelling language meant for specifying dependable, collaborative and time-constraint (DCTC) business processes. The chapter starts by introducing the dimensions of interest that define the domain space addressed by the language (Section 3.1). Section 3.2 analyses the support provided by existing business process modelling languages with respect to the dimensions of interest. After having introduced the domain of interest and the state of the art regarding to the modelling of such as domain of interest, the* DT4BP *modelling language is described. The description of the language is given in two parts. The first section (Section 3.3) is aimed at providing a quick introduction to the essential elements of the language, but without extensive detail. The purpose of this section is to give the reader an overview of the language and its constituent components, which will be detailed in the following section. The second section (Section 3.4) details the key concepts that determine the relevant information that modellers commonly need to include when describing DCTC business processes, along with the primitives that make their concrete representation possible. The introduction and description of the key concepts of the* DT4BP *modelling language, as well as the relationship between each of these concepts are provided following the section concerning the metamodelling principles. This chapter concludes with an analysis and comparison between* DT4BP *and existing business process modelling languages discussed in Section 3.2. This comparison allows the reader to clearly see where this novel language exceeds current existing notations.*

3.1 Dimensions of interest

As claimed in Chapter 1, the main objective of this thesis is to define a modelling language for describing Dependable Collaborative Time-constrained (DCTC) business processes. The objective is to make this language comprehensible, suitable and expressive enough such that the modelling of this kind of business process becomes "easier". In the end, what is required then, is to define a language for a very specific domain. These kinds of languages are known as Domain-Specific Languages (DSL) [KT08]. The first rule when defining a DSL is to understand the domain of interest. Doing this allows for the identification of the core concepts and components the modelling language should cover. Once the modelling concepts are identified, they are formalised by creating a metamodel, because the metamodelling approach has been chosen to

describe the language. In this case, the domain of interest is determined by the intersection of four concepts: business process, collaboration, time and dependability. Thus, identifying the concepts that play a central role in the domain of interest is a necessary condition to engineer a comprehensible, suitable and expressive modelling language.

3.1.1 Business Process

Some of the factors that should be considered when modelling "what a particular business process is intended to do"[1] can be deduced from its definition[2]. An *activity* represents a piece of work performed within the business process. An activity might require some *data* in order to perform. It might also produce certain data as result of its execution. It is usually the case that more than one activity is required to reach the goal. In that case, activities have to be ordered, using certain *control-flow operators*, to create the different paths the business process can go through during its *enactment*[3]. In any case, these paths should lead to achieving the business process' *goal*. The goal, represents the expected result obtained once the business process was enacted.

A business process is enacted within the context of an enterprise. The activities enclosed in the business process are performed by *resources* that belong (whether directly or indirectly) to the enterprise. Since a resource is defined as an entity with the capabilities to perform a business process' activity, it can be either a machine or a human being. A machine is able to perform only activities that are fully automatic, whereas a human being is required to perform activities that are manual or semi-manual.

Resources are involved during the enactment of the business process. But, they also are part of the process definition. Both the available resources that may be assigned to a particular activity, as well as the policy that determines which one will be selected at enactment-time[4] to perform the activity are part of the information that must be explicitly contained in the process definition.

It is usually the case that enterprises group their human resources according to a certain organisational structure. This organisational structure is generally defines a hierarchical relationship between the enterprise's staff members. However, an organisational structure can also be used to capture details about the human resources that are relevant to drive the business processes run by the enterprise. In this perspective, it is assumed that each business process has an associated *resource model* that describes the resources that are available to enact the business process. The notion of *participant* is used as a grouping mechanism to join (either human or non-human) resources with similar profiles and/or duties (e.g. doctors, nurses, secretaries, etc.).

The notion of participant is also used as grouping mechanism to join the activities that are required to be performed by a same participant within a same business process. Thus, the notion of participant is used as a binding mechanism between the business process definition and the resource model to select (according to certain policy) the resources that will perform the participant's activities during the enactment of the business process.

[1] The outcome of this activity is known as *process definition*.

[2] Business process: a set of one or more linked procedures or activities, which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships [Wor99].

[3] Enactment: the actual execution of the business process.

[4] The expressions *run-time* and *execution-time* are used synonomously with enactment-time.

3.1.2 Collaboration

By definition, a business process takes place within the context of an organisational structure. It is such a structure that determines how the work carried out by the an organisation is divided amongst its staff. In today's world, companies require several people with different skills to succeed in providing their company's services. This requirement leads to business processes involving multiple resources that collaborate along the process to succeed in providing the offered service or product (i.e. the process' goal).

Each resource is assumed to have its own processing capacity for performing activities (one after the other). Thus, while a resource denotes a human or non-human entity capable of executing its activities in parallel with some other part, a collaborative business process denotes a set of resources that interact among them. Two resources interact if they communicate by *exchanging messages*. Thus, a resource should be able to *send* and/or *receive* a message from another peer resource (i.e. a resource that is involved within the same business process). The fact that they interact is what defines them to be collaborative with respect to reaching of the process's goal. The collaboration between resources, thus, is expected to be captured in the business process definition as it is a major concern in the kind of business process being targeted.

Collaboration is defined as the exchange of messages between the different resources required to perform the business process activities. However, since this information must be captured in the business process definition, the notion of participant must be used instead of the notion of resource. As explained in the previous section, the notion of participant is used within a process definition to group the activities that are required to be performed (at enactment-time) by the same resource (selected according to certain specified policy). Therefore, in the business process description, messages are exchanged between participants that are enclosed within the same business processes. A particular participant then may send a message to one or more participants, and receive messages (one at a time) sent by other peer participants. Notice that a message may convey certain information, so that messages can not only be used as a synchronisation means among participants, but also as a medium to share information among them. It is worth emphasising that a business process is considered collaborative if and only if two or more participants enclosed within the same business process exchange messages, in spite of the existence of other means to exchange information by and among the participants (e.g. data sharing).

In other words, the concept of collaboration in the context of this thesis is semantically equivalent to the one assumed in the business process modelling language BPMN: a collaboration simply describes the participants and their interactions, where the interaction represents the exchanged messages among the participants [WM08].

3.1.3 Time

During the modelling of a particular business process, the modeller (i.e. the business analysts or the software engineer) requires means to describe the time constraints that form part of the business process being modelled. In this thesis, a time constraint is seen as information given in the form of upper and/or lower time bounds associated with a business process or any of its constitutive parts (e.g. participant, activity). Examples of business process time-constraints are: minimum delay (i.e. the minimum amount of time that must elapse before the start of the business process), maximum delay (i.e. the maximum amount of time that can elapse before the start of the business process), and maximum elapse time (i.e. the maximum allowed time

for completing the execution of the business process). These time constraints are derived from the business goal, organisational rules, laws, commitment, and so on [EPR99].

The information held by a time constraint can be *relative* or *absolute*. Absolute time refers to time as seen in the physical world, whereas relative time refers to the passage of time with respect to certain event[5]. Within the context of this thesis time constraints set during the modelling phase are assumed to hold relative information, only. Events that can be used as a frame of reference for relative time constraints are: (1) the point in time in which the business process is requested, (2) the point in time in which the business process starts its execution, (3) the point in time at which a business process activity starts being executed, and (4) the point in time at which a business process activity is completed[6].

Nevertheless, it is assumed that during the enactment of the business process, absolute deadlines are computed for any relative time constraint. This indicates that the frame of reference used during the execution of a business process is the one determined by absolute time. Therefore, at enactment-time, it is assumed that there exists a device capable of measuring the passage of the *physical time* such that the time constraints can be monitored to determine whether they are met or not. Such s device is called s *clock*.

Every participant involved in a collaborative business process needs to have the same point of view regarding the passage of the time. The *global time based on physical time* is assumed as the time model when modelling the targeted business processes. This time model assumes that there exists a unique clock that accurately measures the passage of *physical time*[7], which is accessible to all participants involved in the business process. Since all participants access the same clock, all of them have the same perspective regarding the passage of the time.

It is worth observing that the *global time based on physical time* model can be achieved using a clock coordination algorithm [GZ89, CF94, Net09] [8]. This gives evidence that the time model considered during the modelling phase is not an oversimplification of the real time model considered during the business process enactment.

3.1.4 Dependability

A business process definition is expected to capture all the possible paths that can be used to achieve its goal. A process instance that follows one of these paths is termed *well-behaved or normal*. It might be the case that a particular process instance follows a path which has not been considered in the definition. Because it is assumed that all possible paths that can be used to achieve the business process goal are specified in the process definition, instances following a non-specified path fail in reaching the business process' goal. It is assumed that once the instance has *deviated* from the process definition it will eventually fail, if no corrective activities are performed. A process instance missing its goal is termed a *business process failure* A *dependable business process* is one whose failures (i.e. process instances missing the goal) are not unacceptably frequent or severe (from some given viewpoint). Therefore, a business process increases its dependability when for those events that cause it to fail more often than expected, corrective activities are explicitly included in its definition such that instances do not miss the

[5] In theory, *absolute time* is also relative, since it counts from a particular event. (e.g. the Christ's birth is the event on which the Gregorian calendar -unofficial global standard- relies on for numbering the years)

[6] The reader is referred to Section 3.4.3 for the full list of events that can be used as frame of reference for defining relative time constraints over a business process and its constitutive parts.

[7] Atomic clocks are examples of these kinds of clock.

[8] A clock coordination algorithm allows a set of distributed clocks to be strongly synchronised to each other.

goal in spite of facing with such undesirable events. In this case, let BP_{def_1} be the original business process definition, and let BP_{def_2} be the new one obtained by including the corrective actions require to deal with the undesirable event. Then it is said that BP_{def_2} is **more** dependable than BP_{def_1}. New definitions of the process are provided (i.e. BP_{def_3},...,$BP_{def_{(n-1)}}$) until a definition BP_{def_n} is obtained such that it includes corrective activities for every undesirable event that makes it fail more often than desired.

The corrective actions to be included in a process definition depend on the particular kind of undesirable event to be handled. Depending on the severity of failure,it might be possible for the system to recover(1) completely, (2) partially, or (3) not at all. In any case, the required activities to deal with the problems have to be included explicitly in the business process definition. The initial business process goal must always be considered while redefining the dependable business process. This is called *explicit dependability modelling*.

It is up to the stakeholders (i.e. owners of the business, business analysts and IT managers and developers) to define what the expected events the business process might be faced with, and more importantly, what required activities will need to be performed in each case to allow the business process not to fail. Notice that it will depend on the severity of each expected event whether the business process reaches its goal totally or partially. The severity of each event determines whether the business process reaches its goal totally or partially. Unforeseen or even impossible events as defined by the stakeholders define the set of unexpected events. When the circumstances force the business process to confront an unexpected event, the business process (usually) will fail. In this scenario, stakeholders must decide to convert unexpected into expected events, and then modify the business process to allow it to manage the situation properly next time the event arises. If no action is taken (i.e. the business process will fail again when the situation comes), then the business process has not improved its dependability. However, a business process confronted with an unexpected event might not fail. It may happen that the business process can cope with an unexpected event if the event is implicitly included within the total set of expected events.

Once stakeholders have decided which events the business process will be able to accommodate while at the same time providing its business objective (either as promised originally, in the best case, or at least partially), the modelling of dependable business processes is reduced to describing every expected event, along with the activities that will allow the business process to accommodate these events when they occurr (aka handler).

3.2 Existing business process modelling languages

This section examines business process modelling languages that are currently being used to specify business processes. Among all these languages, only those that belong to the **graphical standards** (according to the classification given in Section 2.1.1) that are widely used in both the industrial and academic sectors have been considered. Based on this selection criteria, the specific business process modelling languages examined are:

- UML Activity Diagram (UML-AD) version 2.2 [Obj09b]
- Business Process Modelling Notation (BPMN) version 1.2 [Obj09a]
- Yet Another Workflow Language (YAWL) [YAW09a]
- Event-driven Process Chain (EPC) as implemented by ARIS Express 1.0 [ARI09]

These languages are analysed with respect to the kind of business processes this thesis targets, i.e. DCTC business processes. Therefore, special emphasis is placed on the features these modelling languages provide for describing participants and their collaboration, time constraints and dependability aspects. Both the way and means these languages use to cover any of the dimensions of interest are to be considered when engineering the *DT4BP* language. This is because the here aim is to provide a new business process modelling language that not only encompasses each of these dimensions in an improved but also does in an integrated manner.

It is worth noting that in addition to the individual analysis of each existing modelling language, a general comparison between the four languages is also given in Section 3.5. The reason for deferring this comparison until later is because the *DT4BP* modelling language, which is introduced in the next section, must also be considered in this comparison. However, the reader wanting a compact view of the support provided for each existing modelling languages and its position with respect to the others, can directly proceed to Section 3.5 and obviate the DT4BP-related information for these insights.

3.2.1 UML-AD

A UML-AD is one of the different kinds of diagrams the UML [Obj09b] language provides for describing behaviour. A UML-AD can be considered as a directed graph, since it describes a set of nodes (aka activity nodes) linked by directed connections (activity edges). Activity nodes connected by activity edges define the potential execution paths that may be followed by the UML-AD. However, only one path is to be followed once the UML-AD is initiated.

An activity node can be either an *action*, an *activity*, or a *flow-of-control construct*. An action represents a single step within an activity, i.e. it is not further decomposed within the activity diagram. An activity represents a step that is composed of individual elements that are actions. A flow-of-control construct is a node that allows coordinating the execution flow in a UML-AD (e.g. *DecisionNode, ForkNode, JoinNode, MergeNode*).

Incoming and outgoing activity edges are used in activity nodes not only to specify the control flow from and to other nodes, but also to specify the data flow. In this manner, incoming edges can be used by an activity node to acquire its data inputs. Outgoing edges can be used to deliver data information to the enclosing context of such activity node.

There exist structuring mechanisms to organise the UML-AD layout. An activity partition is a kind of activity group for identifying activity nodes that have some characteristic in common. A UML-AD then can be divided into partitions such that its component activity nodes[9] are organised in a way that eases the understanding of the UML-AD. It is argued in [Obj09b] that a partition often corresponds to an organisational unit of the enterprise in charge of running the process the UML-AD describes. Hence, the UML-AD's partitions give a roughly implicit[10] idea of the type, skill and capability of the resources required to enact the UML-AD.

3.2.1.1 Collaboration

Even though partitions can be used to represent the different participants involved in the same UML-AD, the control and data flow defined by the activity edges are not confined to the same

[9] An activity edge can be contained in multiple partitions, since it is possible it has its activity node source in one partition and the activity node target in a different one.

[10] UML-AD does not provide any means to explicitly model the resources that are necessary to execute the described process.

partition. This means that the partitions placed over the UML-AD do not impose any constraints on the control and data flow between activity nodes. Thus, UML-AD has a more general means to achieve collaboration as considered in the context of this work, i.e. the interaction between two different participants is only achieved by exchanging messages. This results because an activity edge that connects two activity nodes placed in different partitions do not indicate anything about the protocol being used to achieve this interaction.

3.2.1.2 Time

Time information on UML-AD is described using the notion of *Accept Event Action*. As with every action, the *Accept Event Action* describes a single step within an Activity, however in this case, this action only occurs when its associated condition is satisfied. Thus, an Accept Event Action is used to capture the occurrence of a particular event that satisfies a certain condition. When the event to be captured is a time event, the action outputs the output parameter, the point in time at which the event occurred. An Accept Event Action that captures a time event is termed *Accept Time Event Action*. Note that the point in time that is specified by the condition of an Accept Time Event Action might be absolute or might be relative to some other point in time.

In this manner, the notion of Accept Time Event Action is the only means provided by UML-AD to model time constraints as considered within the context of this work. Even though in principle every minimum and maximum time constraint can be modelled with this concept, its use must be combined with other concepts (e.g. interruptible activity region) and included several times (e.g it must appear twice to define a range of time) to model the desired requirement. This may easily lead to overloaded models.

3.2.1.3 Dependability

Pre- and post conditions may be associated wtih an activity node. A pre-condition is a constraint that must be satisfied when the activity node execution is started. Conversely, a post-condition is a constraint satisfied when its execution is completed.

An activity node[11] is considered as *protected* when an *exception handler* is attached to the node. An exception handler is an element that specifies what will be executed in the case that the specified exception occurs during the protected node execution. The handler body is an activity node that does not have any explicit input or output edges. Regarding the data flow, the handler body has access to the same information as the protected node, and its output must correspond in number and types to the result pins of the protected node. Regarding the control flow, a protected node might have more than one exception handler. This handler might cope with more than one exception. Once an exception is raised, a handler is searched for coping with the exception. The selection policy is based on type matching: the type of raised exception must match one of the types held by a handler. In the case more than one handler satisfies the selection policy, only one is selected in a non-deterministic manner. When the handler body completes (normally) the execution, it is as if the protected node had itself completed the execution.

In the case that there are no handlers that satisfy the selection policy, the exception is propagated to the enclosing context that contains the protected node. Then the handling process is restarted.

[11] Please note that an activity group can play the role of a protected node. In this case, the activity group is named *interruptible activity region*.

This procedure is repeated until either a handler is found or the outermost level is reached and the exception is not caught. In this case, nothing can be said about the UML-AD behaviour, since it is unspecified.

3.2.2 BPMN

BPMN relies on *Business Process Diagram (BPD)* to allow business processes to be graphically modelled. A BPD then is meant to contain one or more business processes, which are all described using a graphical notation. This graphical notation is a flowchart-like notation that has its elements divided in four categories:

1. *Flow Objects*: elements used to defined the behaviour of a business process. There are three Flow Objects: *Events, Activities* and *Gateways*. Events are used to model the "happening" of something during the course of a business process. Depending on when such events occur, they are categorised as *start, intermediate* or *end* events. An Activity is used to described a piece of work. When an activity is not divided into sub-activities it is considered as atomic, otherwise it is non-atomic. Gateways are used to control the flow of the business process by means of decision points, forking points, and merging points.

2. *Connecting Objects:* elements used to connect Flow Objects. There are three kinds of Connecting Objects including: *Sequence Flow, Message Flow* and *Association*. A Sequence Flow is used to show the dependency order between Flow Objects. A Message Flow is used to show the flow of messages between two different participants. A participant represents a business entity or role that is modelled as a *Pool* in a BPD (see below).

3. *Swimlanes:* elements used to group Activities. There are two grouping elements: *Pools* and *Lanes*. A Pool is used to group those activities that the same participant has to perform in a business process. A *Lane* is a sub-partition within a Pool, used to organise and categorise the activities the same participant performs. A Pool allows a business process to be partitioned in such a way that activities are grouped according to the participant in charge of their execution. However, explicit information about the actual resources that takeover the activities enclosed by a participant is not given as the language does not provide a means to specify this information (i.e. organisation structure, resources and policies of allocation).

4. *Artifacts:* elements used to provide additional information about the business process being modelled, without affecting the Sequence Flow or Message Flow of such process. The artifacts provided by the notation are: *Data Object*, *Group* and *Annotation*. A Data Object is used to model the information an activity requires to be performed and/or what information is produced once such activity has executed. Both Group and Annotation are artifacts meant to ease the documentation of the business process. While the Group is used to define categories of activities, Annotation allows the business analyst to provide additional information for the reader of the BPD.

These notation elements provide the support required to model concepts that are applicable to the business process, only. This means that other kind of modelling done by organisations like business rules, data models and organisational structures and resources are out of the scope of BPMN.

3.2.2.1 Collaboration

BPMN defines a collaboration as any BPD that contains two or more participants which interact with each other. These interactions are defined as communication, in the form of message exchange (shown as Message Flows) between two participants. Therefore, the Pool and Message Flow elements provided by the language are the necessary means to model the collaborative aspects of a particular business process.

It should not be surprising that BPMN considers collaboration as a first-class concern, since support for it was sought since its inception. [WM08].

3.2.2.2 Time

Time is supported in BPMN by the notion of *Timer Start Event* and *Timer Intermediate Event*. A Timer Start Event is used to start a business process, whereas a Timer Intermediate Event is used to delay the start of a particular Activity or to interrupt its execution. Both kind of timers posess a time condition that specifies when the event occurs, i.e. is triggered. When a timer is used to start a process (i.e. it is a Timer Start Event), its time condition may be a specific date and time (e.g. Dec 31, 2009 at 8 AM) or a recurring time (e.g. every Monday at 8am). In any case, the condition is compared with a clock[12] that measures the passage of physical time.

As stated earlier, a Timer Intermediate Event is used either to delay the execution of an Activity or to interrupt its execution. A delay is modelled by inserting a Timer Intermeditate Event between Activities in a process. The Flow Objects that can precede a Timer Intermediate Event are Activities, Gateways, or Intermediate Events.

The time condition held by a Timer Intermediate Event describes the *delay* in starting the next Activity of the process. This time condition specifies an absolute time (e.g. wait until Dec 31, 2009 at 8 AM) or a relative time (e.g. wait 2 weeks) that might be repetitive (e.g. wait until next Monday at 8am). However, Time conditions written using absolute time inhibit the re-usability of the process as the absolute time condition will be valid only once. It is good practice then to rewrite absolute time conditions as relative conditions, if possible.

A Timer Intermediate Event attached to the boundary of an Activity represents a deadline for the execution of the Activity. This means that the Activity has the time condition described by the Timer as the maximum allowed time for completing its execution. If the time condition becomes true before the Activity is completed, then the Activity is immediately interrupted. The time condition that a Timer Intermediate Events used as a *time-out* holds is always relative to the start of the Activity to which the Timer is attached.

3.2.2.3 Dependability

BPMN provides elements to support exception handling. An *Intermediate Event* attached to the boundary of an *Activity* is used to model an *exception* that may occur during the execution of the Activity. In the case that an exception is raised (i.e. the trigger the Intermediate Event holds is fired), the Activity is interrupted. As *normal flow* is the flow followed by the process when the Activity terminates its execution normally, *exception flow* is that to be followed when an Intermediate Event occurs. Intermediate Events attached to an Activity are used to denote

[12] This clock should be seen as a calendar that also provides information about the current physical time: i.e. YYYY-MM-DD:HH.MM.SS

the different exceptional flows the Activity can follow. Flow from the Intermediate Events (i.e. exceptional flows) can go anywhere. The execution can follow a completely new path to perform some handling activities for the exception, or it can join the normal path as if the exception would have not occurred. Alternatively it can go go back to attempt a new execution of the activity.

BPMN also provides support for the notion of *transaction*. In BPMN, a transaction is defined as *a formal business relationship and agreement between two or more participants* [WM08]. A transaction is considered as successfully executed when all the participants taking part have reached a common agreement point.

Since a transaction involves multiple participants, its definition is spread over the pools that describe each participant. Each participant (i.e. Pool) includes a *transactional sub-process* defining those activities that form part of the transaction. A double-lined boundary indicates that a sub-process is a transaction.

In the case that any of the participants taking part in the transaction faces with a processing or technical error, then the transaction is interrupted. There are two possibilities for interrupting a transaction. It can be terminated immediately or some compensation can be performed before terminating such that the transaction is considered cancelled. Intermediate Events attached to the transactional sub-process boundary are used to represent each kind of interruption. An *Error Intermediate Event* is used to model the sudden interruption of the transaction, whereas a *Cancel Intermediate* represents the cancellation of the transaction.

As just mentioned, cancelling a transaction may require that some compensation be performed. Compensation is the undoing of the work a particular Activity has completed. Since compensation does not happen automatically, another Activity is required to undo the work of the original Activity. BPMN supports compensation by means of *Compensation Activity, Compensation Intermediate Event* and *Compensation Association*. Every activity that requires compensation must have a Compensation Intermediate Event attached to its boundary. A Compensation Association is used to bind the original Activity with the one that compensates its effects. The Compensation Association must start at the Compensation Intermediate Event attached to the original Activity and end in the Compensation Activity. A Compensation Activity must not have any incoming or outgoing Sequence Flow. In addition, its use is not limited to activities enclosed within a *transactional sub-process* (i.e. every Activity can have an associated Compensation Activity).

3.2.3 YAWL

YAWL is a modelling language meant for supporting most workflow behaviours commonly found in practice. Such common workflow behaviours are known as *workflow patterns* [vtKB03]. They are categorised according to four different perspectives: control flow, data, resource, and exception handling. A business process specification in YAWL is determined by a set of one or more YAWL-nets, which define a hierarchical graph. Every YAWL-net describes part of the work the business process performs[1] by means of tasks and conditions.

Tasks are either *composite* or *atomic*. A composite task in a YAWL-net is a reference to another YAWL-net at a lower level in the hierarchy of the graph, which describes the way in which such a composite task is defined. An atomic task represents a unit of work that is not further subdivided into sub-tasks.

[1] In case the specification is composed of only one YAWL-net, the net contains the entire definition of the business process.

A condition represents a state of the business process, which can be used to make a choice in the flow of the process. A condition must be located in-between tasks, except the mandatory conditions *input* and *output* that every YAWL-net must have to be considered valid. The input condition represents the starting point of the business process, whereas the output condition represents the end of such business process.

For each YAWL specification, there exists one YAWL-net that does not have a *composite task* referred to it, forming the root of the graph. This YAWL-net is known as the *top level process* or *top level net*.

A task is connected to either a condition or another tasks by a *flow*, which is represented by an unidirectional arrow. A YAWL specification is considered as correct if every task is tied into a YAWL-net via flows that can be traced back to the YAWL-net's input condition, and which eventually lead to the YAWL-net's output condition [BD07]. The number of incoming/outgoing flows a particular task has ranges from one to many. Special decorators have to be added over a task that has more than one incoming or outgoing flow. A *split* decorator is used to specify that the task that owns the decorator is followed by one or more tasks. A *join* decorator specifies the required tasks to be completed before allowing the tasks to become available for execution. Both the split and join decorator have the *OR, AND* and *XOR* associated operators, which are used to determine their behaviour. Details about the behaviour of each can be found in Rusell's thesis [Rus07] on page 252.

YAWL uses a terminology that differs from the one used here, i.e. the terminology of reference within the context of this thesis. In YAWL, a participant refers to the actual resource that performs a particular task, whereas the notion of *role* is used to group different participants that share the same feature. In the terminology here, the terms used to are *resource* and *participant*, respectively. To avoid any misunderstanding, YAWl terms have the subscript *yawl*: i.e. $participant_{yawl}$ and $role_{yawl}$.

YAWL uses the concepts of $participant_{yawl}$ and $role_{yawl}$ as the building blocks to define the *Organisational Model* of the enterprise that owns and runs the business process being modelled. The Organisational Model along with additional constraints are the elements used to specify the $participant_{yawl}$ allocation policy: i.e. what are the $participant_{yawl}$ that must perform certain tasks. Despite the different terminology employed by YAWL, the fundamental idea of modelling the resources that will takeover the participant's activities at enactment time is considered in the modelling language as explained in Section 3.1.1.

3.2.3.1 Collaboration

Collaboration, as considered in the context of this work (i.e. message exchange between two different participants), is not supported in YAWL.

3.2.3.2 Time

YAWL has been recently extended with features to define the time behaviour of a particular atomic task [YAW09b]. These features include the possibility to define a *timer*, used to constrain the behaviour of an atomic task. The semantics of a timer depend on whether the atomic task is *manual* or *automated*.

For a manual atomic task, a timer is used to define a time frame onto which the life cycle of

the manual atomic task[13] is allowed to take place. For this kind of atomic task, the timer is initiated either when the task is enabled (i.e. it is *offered* or *allocated*) or started. Activation on enablement means that the timer begins as soon as the manual task is enabled, whereas activation upon starting means the timer starts only when the task has started, i.e. the task will be first offered, then allocated, and once it is started the timer is initiated.

For an automated atomic task, a timer behaves as a delay: the task instance created to execute an automated task delays its execution until the timer's *expiry value* is reached. The expiry value specifies the duration of the timer, and it is either an absolute time (i.e. a specific date and time is defined) or a relative time, i.e. for a manual task the timer is relative to its enabling or starting, whereas for an automated task to the creation. Whether the task is manual or automated, once the expiration value is reached by the timer, the task instance will complete whatever its current status is (i.e. offered, allocated, started).

3.2.3.3 Dependability

YAWL covers dependability by means of exception handling. The actions to be taken when the exceptions that may arise during the execution of a particular business process can be handled, are defined by a set of primitives incorporated into the modelling language. These primitives, which are chained in sequence, define the handling process (called *exlet*) for a particular exception. Such handling primitives are:

- **Suspend Workitem:** a workitem (i.e. a process instance activity) is suspended until it is either continued, restarted, cancelled, failed or completed, or the entire process instance is cancelled or completed. A process instance activity can be suspended only when it has a status of fired, enabled or executing)
- **Suspend Case:** a case (i.e. a process instance) is suspended. Suspending a process instance means that every single activity owned by the suspended process instance is suspended.
- **Suspend All Cases:** all process instances of a particular business process definition are suspended.
- **Continue Workitem:** resumes the execution of a process instance activity, which was previously suspended.
- **Continue Case:** resumes the execution of a process instance (i.e. every activity owned by such process instance resumes its execution)
- **Continue All Cases:** resumes the execution of all process instances belonging to a same business process definition.
- **Remove Workitem:** ends the execution of a process instance activity and marks its status as cancelled. Further activities that are in the same process path of the removed activity are not executed.
- **Remove Case:** ends the execution of the process instance.

[13] At enactment-time, a task is *created* (or instantiated), then it is *offered* to the pool of available $participants_{yawl}$ that can executed; once the task has been *allocate* to a particular $participants_{yawl}$ it can be *started*. While a task is being executed by the $participants_{yawl}$, it can be *suspended*. Eventually the task is expected to either *complete* or *fail*.

- **Remove All Cases:** ends the execution of all process instance of a particular business process definition.
- **Restart WorkItem:** restarts the execution of a process instance activity under the same conditions as it was when executed for first time (i.e. the process instance activity is rolledback).
- **Force Complete WorkItem:** completes a process instance activity that is currently under execution (i.e. its status is fired, enabled or executing). The process instance activity is considered as successfully executed, and the execution proceeds to the next process instance activity.
- **Force Fail WorkItem:** fails a process instance activity that is currently under execution (i.e. its status is fired, enabled or executing). The process instance activity is considered as unsuccessfully executed (but not cancelled), and the execution proceeds to the next process instance activity.
- **Compensate:** executes a business process that, depending on the primitives used previously either runs in parallel along with its parent business process instance, or does so while its parent is suspended or removed.

A handling process that is defined using the above-listed primitives will take over the execution of the parent business process instance only when a certain *exception* occurs. An exception is defined as the occurrence of a certain *kind of event* along with the existence of a *rule* that allows the event to be bound with a handling process. The kind of events for which a rule may exist are:

- **Pre and post Case/WorkItem execution:** events that generate a notification that a particular case (i.e. process instance) or workitem (i.e. process instance activity) is ready to be executed or has just concluded its execution. Such events are meant specifically to check four kind of rules:
  - **CasePreConstraint:** rules that are checked before the process instance begins its execution,
  - **ItemPreConstraint:** rules that are checked before the process instance activity begins its execution,
  - **CasePostConstraint:** rules that are checked just after the process instance concludes its execution,
  - **ItemPostConstraint:** rules that are checked just after the process instance activity concludes its execution.
- **External Trigger:** events that generate a notification of the occurrence of something outside of the process instance execution that affects its continuation. Such events are categorised depending on whether they affect the entire process instance (aka *CaseExternalTrigger*) or just one of its activities (ItemExternalTrigger). Based on this categorisation, the event can be handled at the level of either the process instance or one of its activities.
- **Timeout:** event that generates a notification that a particular process instance activity owning an associated timer has reached its deadline.

- **Resource Unavailability:** event that generates a notification that the selected resource to be allocated for carrying out certain process instance activity is unavailable to accept the allocation.
- **Item Abort:** event that generates a notification that the execution of a particular process instance *automated activity* (i.e. the activity is automatically executed by an external application) has aborted before being completed.
- **Constraint Violation:** event that generates a notification that the data constraint associated with the process instance activity being executed has been violated.

A rule[14] is a predicate, which is used to determine which handling process, if any, to invoke. If for a certain kind of event a rule exits and evaluates to true, the handling process owned by the rule is invoked. If no rule exists for a certain kind of event in a given business process definition, then the event is simply ignored when the process instance is executed.

3.2.4 EPC

EPC is graphical language meant to describe business processes in a manner business people can easily understand and use. EPC has become a widespread modelling language because it is the notation used by some of the leading tools in of the field of business process management such as SAP R/3 and ARIS [van99].

A business process is described in EPC as a set of *events* and *activities* which, when chained in sequence, define the control flow structure of the business process. An event is used to describe both a condition that leads an activity to be started and a result of having executed one. An activity describes a piece of work that needs to be performed in order that the entire business process completes successfully. An activity may require some information (i.e. *input parameter*) to execute or it may produce information (i.e. *output parameter*) due to its execution. The *document* element is the primitive provided by EPC that allows information to be modelled in the business process definition.

A *connector* is the element used to link the different elements that compose a business process definition. An activity can be connected to another by using a connector. Events are allowed to be connected to activities.

Rules are special kind of connectors used to bind both events and activities. There are three kinds of rules:

- **AND rule:** the processing steps that follows the rule are performed once all incoming processing steps are completed.
- **OR rule:** the processing steps that follows the rule are performed once at least one incoming processing steps is completed.
- **XOR rule:** the processing steps that follows the rule are performed once one (and only one) incoming processing steps is completed.

Other elements provided by the language are oriented to the specification of the resources in charge of carrying out the business process activities. These elements are *Organisational Unit,*

[14] A detailed explanation about rules can be found in [YAW09b], in Section 7.5.

Role and *Person*. While the element *Organisational Unit* is used to define the organisational structure of the enterprise that runs the business process(s) being modelled, the *Role* element is used as the means to specify the profile, skills or characteristics of the people belonging to a particular Organisational Unit. The element *Person* is used to define an actual person, who is usually bound to at least one role. Such binding means that the person has the abilities to play such role. Thus, it can be specified at modelling time whether an activity is always performed by the same person, or it is deferred until runtime. The former choice is made by linking the activity to the particular person, whereas the latter is done by linking a role or organisational unit to the activity.

3.2.4.1 Collaboration

Collaboration, as considered in the context of this work (i.e. message exchange between two different participants), is not supported in EPC.

3.2.4.2 Time

The language (at the writing time) does not provide any element that allows time-related aspects of a business process to be modelled (e.g. activity duration, delay between activities, etc.)

3.2.4.3 Dependability

The language (at the time of this writing) does not provide any element for addressing dependability as considered within this work.

3.3 DT4BP: a tutorial introduction

This section is aims at providing a quick introduction to *DT4BP*. At this point in the discussion, the goal is not to be complete or even too precise. Thus, important features of the language are intentionally left out. The goal is to get the reader to the point where he can understand how business processes are modelled in *DT4BP* as quickly as possible. The best way to do this (and probably the only way) is to take the reader through the actions of modelling business processes. A fictitious and simple business process for diagnosing a patient is used as running example to describe a DCTC business process[15]. This example is adapted from the one described in [MRv+09]. The business process is aimed at dealing with people arriving at a hospital's diagnosis unit with the intention of being diagnosed and getting a treatment for the detected disease. The activities involved in the business process are the following:

1: *registration:* upon the arrival of the person at the diagnosis unit, a secretary checks his identity (by requesting his medical insurance card) and then registers his admission to the diagnosis unit,

[15] This case study will be extensively and intensively used along the thesis as a vehicle to demonstrate how the introduced concepts can be put in practice.

2.1: *examination:* an assistant and a nurse do the first examination of the patient[16] like checking his temperature and blood pressure to determine his overall health status, and in parallel,

2.2: *make document:* a nurse prepares the patient sheet. A patient sheet is a document that includes not only personal information about the patient as his name, address, phone and age, but also relevant medical information gathered from his local record stored in the hospital's database (if any).

3: *consultation:* a doctor evaluates the results of the examination and checks the patient in order to diagnose the disease or problem. Once the doctor has made his diagnosis, he sets a treatment and informs the patient. Both the diagnosis and the prescribed treatment (and the medicine to take, if any) are written in the patient sheet.

4: *give information:* a nurse gives more precise information to the patient about the prescribed treatment and how to get the medicine (e.g. doses and frequency).

The hospital administration wants the diagnosis units to attend to every patient in no more than two hours after their arrival at the unit (if possible). This requirement is refined as multiple time constraints over every individual activity of the process in the following manner: "registration" must take less than 15 minutes, "examination" and "make document" less than 30 minutes, "consultation' less than 1 hour (but more than 15 minutes to ensure there has been some quality in the check), and "give information" less than 15 minutes. These requirements are what make the business process time constrained as considered in this thesis.

Undesirable events that may take place along the diagnosis business process range from non-critical, for example foreigners or people without medical insurance trying to get diagnosed, to critical issues such as a fire in the hospital. The diagnosis business process, thus, has to be enriched such that either the process' goal (i.e. the patient is diagnosed) can be totally or partially reached, or the negative effects of the event over the process can be mitigated. In this manner, while some alternative activities are added to allow a foreigner to be diagnosed, others are added to mitigate the harmful consequences of a fire at the hospital, even knowing the process will fail. Considering these undesirable events[17] as possibilities (along with activities that will be required to deal with them should they arise) during the development of the business process makes the business process a dependable business process as considered in this thesis.

The collaborative aspect of the process is determined by the required interaction between the different categories of people (e.g. a secretary, a nurse, an assistant, and a doctor) in each of the activities that composes the business process. In the "registration", for example, the secretary interacts with the person (i.e. potential patient) that wants to be diagnosed in order to certify that the person has medical insurance. In the "examination" activity, the interaction is between the patient, a nurse and an assistant. The patient also interacts with a doctor and a nurse in the "consultation" and "give information" activities, respectively. Since the patient is required to perform some activities during the process, he is also considered a participant[18].

The requirements previously introduced make the diagnosis component of the business process a part of the business processes this thesis targets (i.e. DCTC business processes). The remainder

[16] Once the person is within the diagnosis unit he is considered a patient.

[17] The reader is referred to the Appendix C for the full list of undesirable events being considered in the running example.

[18] A business entity that is required to perform at least one activity of the process is considered a participant in the process.

of this section uses this business process as a vehicle to demonstrate the main characteristics of the *DT4BP* language that make it suitable for business process modelling.

3.3.1 Getting started with DT4BP

In *DT4BP*, the specification of a business process is split into four different views or models. This division is meant to ease the organisation and interpretation of the process definition. These models are the *Process Model*, the *Data Model*, the *Resource Model* and the *Dependability Model*. While the *Process* and *Resource* models are mandatory, the existence of the *Data* and *Dependability* models depends on whether data elements are required and whether undesirable events are being handled, respectively. The *Process Model* describes the control flow of the business process, whereas the *Resource Model* describes the flow of the actual business entities on which the organisation relies to achieve the enactment of the business process.

In achieving a business process specification using the *DT4BP* language, the initial model to be provided is the *Process Model*. The activities involved in the business process and the flow on which these activities should be performed at enactment-time are part of the information to be described within the *Process Model*.

The notion of participant is used as a means to group the activities tha the participant is responsible. The activities that each participant must perform are either *composite* or *atomic*. A composite activity (for the sake of simplicity) can be seen as one that is made up of sub-activities. Conversely, an atomic activity is one that is not composed of sub-activities. The sub-activities that a composite activity refers to, are defined in another business process. In this manner, a composite activity actually refers to a different business process.

The concept of composite activity determines the hierarchical structure of business processes. A business process containing a composite activity is at a higher level than the business process that encloses the sub-activites the composite activity refers to. It is assumed the existence of a business process which is not referred to by any composite activity. This business process is the root of the hierarchical structure. In *DT4BP*, it is a good modelling practice to restrict the root business process to only one participant (usually the organisational unit in charge of running the business process). This will provide an overall view of the activities that take place within the business process in a condensed view.

Adhering to this good modelling practice, the root *Process Model*[19] that corresponds to the *diagnosis* business process is shown in Figure 3.1. The activities "registration" (line 12), "examination" (line 17), "makeDocument" (line 18), "consultation" (line 21), and "giveInformation" (line 23) are enclosed within the participant *DiagnosisUnit*, which is the hospital unit where these activities take place. These activities are executed in sequence (modelled with the control-flow operator *';'*), except *examination* and *makeDocument* which are executed in parallel (modelled with the control-flow operator *'split'*).

Notice that all these activities are *composite* activites, which means that (1) business processes named *registration, examination, makeDocument, consultation* and *giveInformation* are also part of the overall definition of the *diagnosis* business process[20], and (2) these business processes are located at a lower hierarchical level with respect to the *diagnosis* business process, which is the root business process.

[19] This is a simplified version of the real *Process Model*. See the Appendix C for the full description of the model.

[20] The reader is referred to the Appendix C for the full description of these business processes.

```
1  ;;************************************************************
2  ;; PROCESS MODEL
3  ;;************************************************************
4  business process diagnosis(out PatientSheet ps) {
5
6
7    participant DiagnosisUnit {
8       FireAlarm fa;
9
10      do{
11         Person prs;
12         composite registration[_,p = alloc(new Patient)](out prs) within[_,15min.];
13
14         Temperature t;
15         BloodPressure bp;
16
17         do{split composite examination[p,_,_](out t,bp)
18                 composite makeDocument(in prs; out ps)
19            }within[_,30min.];
20
21         composite consultation[p,_](in ps,t,bp; out ps) within[15min.,1hs.];
22
23         composite giveInformation[p,_](in ps) within[_,15min.]
24
25      }deviation[EX_Fire| fa = #ON)]
26   }
27
28 }
```

Fig. 3.1: *Process Model* of the Diagnosis Business Process.

Figure 3.2 shows part of the Process Model of the *registration* business process. This business process involves two different participants: the *Secretary* and the *Patient*. The Secretary interacts with the Patient by requesting his medical insurance card. This interaction, which defines the collaborative aspect of the business process, is achieved by an exchange of messages between the two participants using the primitives *send* (lines 8 and 17) and *receive* (lines 10 and 15).

The resources in charge of executing the activities enclosed by the *Secretary* and *Patient* participants within the *registration* business process are specified at the level of the composite activity that refers to such business process. In this manner, the resource allocation policy for the *registration* business process is modelled within the *diagnosis* Process Model, in the resource region (i.e. area determined by the [ ] brackets - line 12). The allocation policy specified for the *registration* business process defines that the secretary (modelled with the symbol '_') will be any of those specified within the *Resource Model* associated to the *registration* business process (shown in Figure 3.3). Whereas the resource required to play the activities enclosed by the *patient* participant, is created at enactment-time (i.e. upon the creation of the process instance).

A reference for the created resource (here named p) is kept in order to specify that the same resource must be involved in the subsequent composite activities of the *diagnosis* business process. The reference p is considered a *resource variable* and can be used in any subsequent resource region with respect to its definition. Notice that the activities enclosed by the *secretary* participant within the *registration* business process are to be executed at enactment-time by the resource *Ann* (line 5, in Figure 3.3) as it is the only available resource that matches for that kind of participant.

Both a business process and its composite activities might require some information in order to be able to execute, or they might produce some information as result of their execution. *DT4BP* adheres to an *object-oriented* model to allow modellers to represent information in a

```
1  ;;*************************************************************
2  ;; PROCESS MODEL
3  ;;*************************************************************
4  business process registration(out Person p){
5
6
7    participant Secretary{
8      send reqSSCard to Patient block;
9      ...
10     receive reqSSCard(ssCard) from Patient;
11     ...
12   }
13
14   participant Patient{
15     receive reqSSCard from Secretary;
16     ...
17     send reqSSCard(ssCard) to Secretary;
18     ...
19   }
20 }
```

Fig. 3.2: Extract from *Process Model* of the Registration Business Process.

```
1  ;;*************************************************************
2  ;; RESOURCE MODEL
3  ;;*************************************************************
4  resources{
5          Secretary = Ann;
6          Patient;
7  }
```

Fig. 3.3: *Resource Model* of the Registration Business Process.

business process. An object represents an entity that contains certain information. The kind of information that may be contained by an object is determined by its data type. The data types used to create objects within a business process must be defined in the *Data Model* associated with the business process. Figure 3.4 shows the Data Model of the *diagnosis* business process. A data type is defined in the Data Model using the construct *type* and followed by a name. The name of the data type is its identifier, which must be unique within the same Data Model. A data type can be defined in a structured manner by using other defined data types. This is the case for the *Person* data type, which relies on the data type *Calendar* and the pre-defined types[21] *String* and *Integer* (the other pre-defined types are *Float* and *Boolean*). Please note that the *Integer* type represents mathematical natural numbers, whereas *Float* the mathematical concept of real numbers. The *String* type represents a sequence of characters[22], and the *Boolean* type the values *true* and *false*. A data type's attribute may have the range of possible values constrained. This is achieved by setting conditions (aka invariants) over each of the attributes to be constrained. Conditions (introduced by the construct *where*) are first-order logic formulas written using the OCL language [WK03]. The data types *Temperature* and *BloodPressure* are examples of data types with invariants. A data type can also be defined as an enumeration of elements[23]. This is the case of the data type *Alarm*.

Objects take place within the Process Model, only. The name of the object is its identifier, and

[21] Pre-defined data types are referred to as *types*.
[22] Characters of the ISO/IEC 8859-1:1998 [Int98], known as *Latin-1*.
[23] The elements of an enumeration are considered strings.

```
;;*************************************************************
;; DATA MODEL
;;*************************************************************
type  Person{
        String name;
        String surname;
        Calendar birthday;
        String address;
        String city;
        String country;
        Integer ssn;
}

type Calendar;

type Temperature { Integer t where (30 <= t) and (t <= 50)}

type BloodPressure { Integer bp where (50 <= bp) and (bp <= 250)}

type Treatment, Medicine, Diagnosis, MedicalHistory;

type PatientSheet{
        Integer ssn;
        String name;
        String surname;
        String address;
        String city;
        String country;
        MedicalHistory mh;
        Diagnosis d;
        Prescription p;
}

type Prescription{
        Treatment t;
        Medicine m;
}

type FireAlarm = enum{ON,OFF}
```

Fig. 3.4: *Data Model* of the Diagnosis Business Process.

it must be unique within the context of a same business process. The scope of an object is determined by the place where it is defined. A parameter is an object defined in the header[24] of the business process. The object *ps* is an example of a parameter object (line 4, in Figure 3.1). For this kind of object, the scope is the entire Process Model. It is worth noting that a parameter can be defined as *input* or *output*. Input parameters represent objects that carry information coming from the environment where the business process is enclosed, whereas output parameters are used to pass information to the environment. Notice that an object that wants to be used both to receive and send information from/to the environment must be defined as input and output. An object may also be defined within a participant. In this case case the scope of the object is the participant where it is defined.

An object (whether defined as a parameter or not) is used both to pass information to an activity to allow its execution and to capture the effects of having executed the activity. Objects used to pass information to the activity are known as *input arguments*. Those used to capture the effects of the activity execution are known as *output arguments*. Notice that the same object can be used as both input and output argument. In the example, the (parameter) object *ps* is

[24] Part of the process definition that contains only the *name* and parameters of the business process.

used as both input and output in the *consultation* composite activity (line 21, in Figure 3.1).

The *DT4BP* language offers facilities for modelling time constraints over a business process, as well as its constitutive parts (e.g. participants and activities). The time-related requirements concerning the *diagnosis* business process are modelled using the primitive *within*. This primitive is aimed at defining the minimum and maximum allowed time an activity may take. The requirement that the registration of a patient should take less than 15 minutes is modelled using this primitive (see line 12, in Figure 3.1). Notice that the symbol '_' is used to model the fact that there is no boundary regarding the minimim allowed time for the execution of the activity. In the same manner, the same primitive is used to constrain the other composite activities (lines 19, 21 and 23). Notice that for the composite activities *examination* and *makeDocument*, which are to be executed in parallel, the time constraint modelled with the primitive *within* applies to both activities as it is associated with the block (modelled with the construct *do*) that contains both activities.

Making the *diagnosis* business process more dependable means dealing with certain undesirable events such that their occurrence during the process enactment either do not result in a failure of the business process (ideal scenario), or the negative effects they produce can be mitigated. When modelling a dependable business process, the first step consists of identifying the undesirable events that may take place during the enactment of the business process. For the sake of simplicity, the only undesirable event to be considered here is a fire taking place at the hospital. Once the undesirable events have been identified, the next step consists of modelling how the occurrence of these events are detected. *DT4BP* provides the notion of *deviation* to model the detection of an undesirable event. A fire is detected within the diagnosis unit when the fire alarm (modelled with the object *fa* -line 8, in Figure 3.1) is activated (i.e. the value held by the object *fa* is equal to *#ON*). The fact that the alarm may turn on at any point during the execution of the activities carried out within the diagnosis unit is modelled attaching the deviation to the entire block of activities (lines 10-25).

```
1  ;;*************************************************************
2  ;; DEPENDABILITY MODEL
3  ;;*************************************************************
4  resolution{
5          EX_Fire -> Evacuation;
6  }
7
8  recovery{
9          Evacuation(){
10                 participant DiagnosisUnit{
11                         evacuateDiagnosisUnit();
12                 }
13         }failed
14 }
```

Fig. 3.5: *Dependability Model* of the Diagnosis Business Process.

The concept of deviation is used to model not only how the undesirable event is detected, but also how its occurrence is noted within the business process where it takes place. The occurrence of an undesirable event is noted by means of signals or exceptions. Thus, *EX_Fire* models the exception that will mark (at enactment-time) that a fire is taking place within the diagnosis unit. It is worth noting that the raising of an exception at enactment-time changes the execution flow of the business process to the *handler* meant for dealing with the exception. The binding between the exception used to note the occurrence of an undesirable event and the handler in charge of its handling, as well as the proper definition of the handler, is modelled within the

Dependability Model. Figure 3.5 shows the Dependability Model associated to the diagnosis business process. The model is composed of two parts: the first part, named *resolution* modelles the binding exception-handler (line 5), whereas the second part, named *recovery*, provides the definition of the handler (lines 9-13), which consists of the *evacuateDiagnosisUnit* activity. This activity is executed once the control flow of the business process is transferred to the activity due to the exception. It must be noted that this handler does not help to reach the process' goal (i.e. patients are not supposed to be diagnosed nor receive any treatment in the case of a fire), but it does help to mitigate the effects of the fire (patients are evacuated so that they are not injured because of the fire). The keyword *failed* (line 12) is used to model the fact that the business process fails despite performing activities by the handler.

This section provides an overall view of the different models to be considered when composing a business process in *DT4BP*. In the following discussion, a more detailed description of the language will be given.

3.4 DT4BP: a detailed explanation

This section presents the *DT4BP* language in detail. The language is presented by introducing the domain-specific concepts that allow the modelling of the targeted business process. These domain-specific concepts are introduced according to the dimension of interest enumerated in Section 3.1. In this manner, Section 3.4.1 presents the concepts oriented toward the modelling of the business process aspects, Section 3.4.2 presents the concepts that allow for modelling the collaboration between participants enclosed within a same business process, Section 3.4.3 presents the time-related concepts, and Section 3.4.4 presents the concepts that allow the modeller to describe the dependability-oriented aspects of the business process.

The domain-specific concepts presented in these four different sections are formalised according to the metamodelling principle (see Chapter 2, Section 2.3). This organization demonstrates that the language domain-specific concepts are formalised in terms of meta-model elements. Thus, every time one or more concepts are introduced the meta-model component corresponding to that formalisation is given so that the reader may have a clear understanding how the entire *DT4BP* meta-model is obtained (Appendix B). This meta-model represents the *abstract syntax*[25] of the *DT4BP* language [Kle08].

The *diagnosis* business process presented in the previous section is used again to show how the *DT4BP* domain-specific concepts are implemented when modelling a DCTC business process. Hence, the symbols that allow each domain-specific concept to be present within a process definition are also given in this section. These symbols are part of the *concrete syntax*[26] of the language (often simply referred to as *syntax*). The concrete syntax then is what the language user (the business process modeller in this particular case) needs to know to write syntactically correct *DT4BP* models. The concrete syntax of *DT4BP* (which is a textual modelling language) is specified by a context-free grammar using an extended version of the BNF [27] notation. The full *DT4BP* concrete syntax is given in Appendix A. Thus, the parts of the *diagnosis* business process shown throughout this thesis are written in accordance with this concrete syntax. Please

[25] The part of the language specification, which describes the structure of the allowed expressions in the language [Hen90].

[26] That part of the language specification, which describes the sequence of symbols that must be present in the concrete form of a model, as well as to assign (in an unambiguous manner) to each sequence of symbols a unique structure or parse tree.

[27] Backus-Naur Form.

note, that when referring to the reserved words[28] of the *DT4BP* language, they appear as **bold** text.

3.4.1 Core elements

The aim of this section is to introduce all the features that *DT4BP* provides with regards to modelling the business process dimension. These features have been defined bearing in mind the concepts identified in Section 3.1.1, which correspond to the business process part of the targeted domain. These features then are expected to be sufficient to allow modellers to describe comprehensive process definitions.

3.4.1.1 Activity

An activity represents a logical unit of work that is performed within the context of a business process. A business process is assumed to be composed of one or more activities. The work these activities represent is what allows the business process to reach its goal.

The work an activity represents may be complex. The divide and conquer strategy is often applied when a task is very complex. Hence, decomposition is used as means to allow a complex activity to be sub-divided into sub-activities such that their completion (i.e. each sub-activity successfully perform its task) represents the completion of the enclosing activity. These sub-activities can be seen as activities that are performed within a different business process. Let bp_1 and bp_2 be two different business process such that bp_1 has a complex activity $act_1^{bp_1}$ that itself is divided into sub-activities $act_1^{bp_2}, ..., act_n^{bp_2}$ that are performed within the business process bp_2 and which are not further subdivided into sub-activities. Then, $act_1^{bp_1}$ is known as a *composite* activity, whereas $act_1^{bp_2}, ..., act_n^{bp_1}$ are known as *atomic* activities. As a result, the business process bp_2 is considered as being at a lower hierarchical level with respect to the business process bp_1.

A composite activity, then, can be seen as a reference to another business process. At enactment-time, the execution of a composite activity is indeed made by the execution of the activities enclosed within the referred business process. The composite activity may be considered as the caller of the referred business process, which waits for the completion of the inner business process in order to be considered as completed. In other words, a composite activity represents the embedding of a business process into another business process such that they can be hierarchically structured. It is assumed that there exists one (an only one) business process that is not referred to by any composite activity, and which is known as the root of the hierarchical structure (constraint (1)). For a non-root business process then there exists (at least) one composite activity which refers to such business process (constraint (2)). Business processes at the lowest level are necessary made of atomic activities, only.

Figure 3.6 shows the part of the *DT4BP* meta-model that formalises the notions of composite and atomic activity. An *Activity*[29] is defined as an abstract class[30], since it is made concrete either as an *Atomic* or a *Composite* activity. An Atomic activity has a name, its identifier; whereas a Composite activity is associated with the business process to be called at enactment-time. The name of the business process called by the composite activity (modelled with the association

[28] Within the context of this thesis, referred to as *constructs, primitives or keywords.*
[29] When referring to a class that appears on the meta-model, its name is written in capital.
[30] An abstract class is denoted in the meta-model by showing its name in *italic*

named *call*) is the identifier of such composite activities. A composite activity must call a non-root business process (constraint (3)). The business process being called by a composite activity must be different from the one that encloses the composite activity. In other words, a business process cannot be self-referred (constraint (4)).

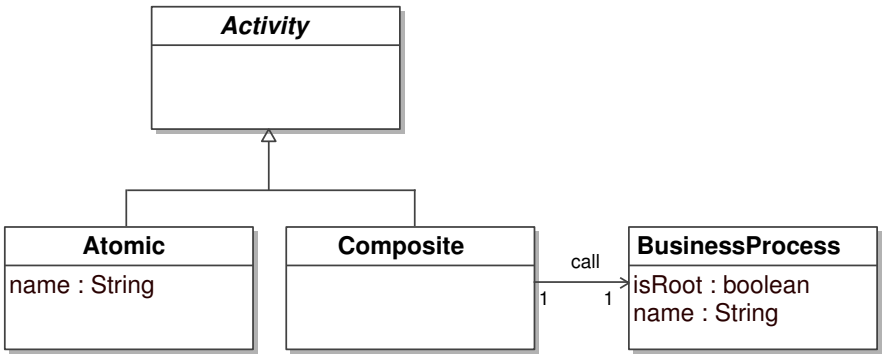

Fig. 3.6: Kind of Activities in DT4BP.

The OCL conditions[31] that check these constraints are shown in Figure 3.7[32].

```
--constraint 1
context BusinessProcess inv uniqueRootBP:
        BusinessProcess.allInstances()->one(bp|bp.isRoot)

--constraint 2
context BusinessProcess inv nonRootBPisCalled:
        BusinessProcess.allInstances()->select(bp|not(bp.isRoot))->
                forAll(bp|Composite.allInstances()-> exists(cmp:Composite|cmp.call=bp))

--constraint 3
context Composite inv callNonRootBP:
        Composite.allInstances()->forAll(cmp:Composite|not(cmp.call.isRoot))

--constraint 4
context Participant inv nonSelfReference:
        Participant.allInstances()->forAll(p:Participant|
                (let
                  cmps:Sequence(Composite)= p.stmts->
                    select(stmt:Statement|stmt.oclIsTypeOf(Composite))->
                        collect(stmt| stmt.oclAsType(Composite))
                 in
                        cmps->forAll(cmp:Composite|cmp.call <> p.bp)
                )
        )
```

Fig. 3.7: Formalisation of the conditions 1-4 in terms of the OCL specification language.

3.4.1.2 Control flow

It is not enough to identify the activities that are required to achieve the business process' goal. It is also necessary to specify the order in which these activities must be completed to achieve

[31] In order to be syntactically correct, OCL expressions were written using the *Dresden OCL2 Toolkit* [Sof99].
[32] The reader is encouraged to examine the full *DT4BP* meta-model shown in the Appendix B to understand the formalisation of the constraint (4).

the goal.

The execution dependencies between activities are specified using control flow operators. The selection of the types of control flow operators included in *DT4BP* for organising the execution order of the activities was strongly influenced by those proposed by van der Aalst and van Hee in [van02]. The types of control flow operators included in *DT4BP* are:

- **Sequential execution:** allows activities to execute one after the other (i.e. act_1; act_2)),
- **Parallel execution:** allows certain activities to execute in parallel, meaning that the activities can be executed simultaneously or in any order. The parallel execution of activities is differentiated into two cases: 1) parallel activities that need to be synchronised before the end of the enclosing business process, and 2) parallel activities that last until the end of the enclosing business process. The concept *split* is used to denote the former case (i.e. *split* $(act_1, ..., act_n), .., (act_1, ..., act_m);$), whereas *spawn* refers to the latter case (i.e. *spawn* $(act_1, ..., act_n), .., (act_1, ..., act_m);$),
- **Conditional execution:** allows execution to choose between two activities based on certain conditions. The concept *if-then-else* is used to denote this control flow operator (i.e. *if cond then* act_1 *else* act_2),
- **Iterative execution:** allows the execution of one or more activities while a certain condition holds. Depending on whether the condition is checked before or after the activities are executed, the iterative execution of the activities can be differentiated between *while* (i.e. *while cond do* $act_1, ..., act_n$) and *repeat* (i.e. *repeat* $act_1, ..., act_n$ *until cond*).

The formalisation of these concepts with regard to the control flow operators is shown in Figure 3.8. These operators (i.e. *Split, Spawn, If, While* and *Repeat*) are specialisations of the class *Control*. The abstract classes *Control* and *Activity* are the two types of *Statements* (also abstract) that can be performed. Please note, that while the control flow operators *If, While* and *Repeat* allow the execution of an ordered (potentially empty) sequence of instructions, the operators *Split* and *Spawn* only allow the execution of two or more activities. The attributes *until* and *cond*, which are used to denote the predicates of the conditional operator *If*, and the iterative operators *While* and *Repeat* are of type *OclConstraint*. The *OclConstraint* data type represents first-order logical expressions, written using the *OCL* language [WK03].

It is worth noting that, in the context of this thesis, OCL is used as a tool to write first-order logical expressions that must be used for decision making[33] In addition, these expression must also be computed within a reasonable amount of time. Thus, the modeller may only rely on a sub-set of the OCL's primitives for writing such expressions.

Figure 3.9 shows how some of the control flow operators are used to order the activities of the *diagnosis* business process. The first activity to be performed is "registration". This is a *composite* activity, therefore a business process definition for "registration" has to be also provided as part of the overall business process definition[34]. The *split* operator is used to start "examination" and "make documents" composite activities in parallel. "Consultation" will wait for the completion of these parallel activities since they were started using a *split* primitive. It is worth recalling that, in *DT4BP*, the information concerned with the control flow of the business process activities is depicted in the **Process Model** view.

[33] There exists a Turing machine that always halts given any finite input string.
[34] Idem for "examination", "make document" , "consultation" and "give information".

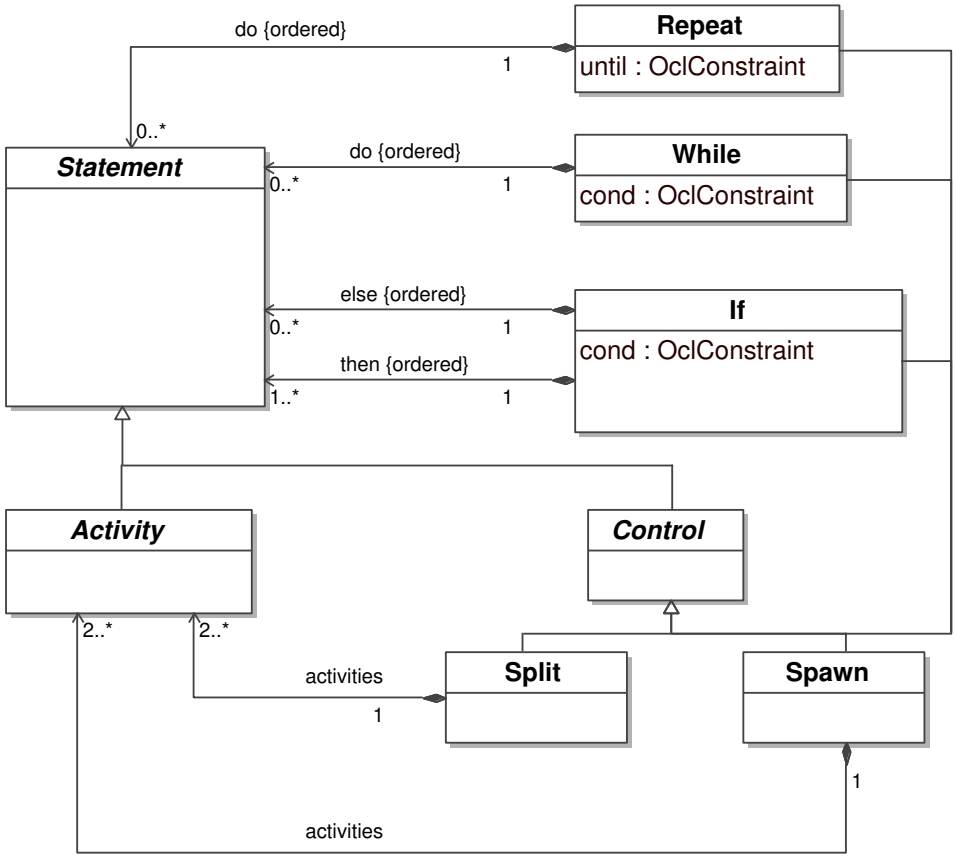

Fig. 3.8: Control flow operators in *DT4BP*.

3.4.1.3 Participant and Resource

The notion of *participant* is used as a structuring mechanism to encapsulate a set of activities and their dependencies under the context of a particular kind of business entity. A business entity capable of doing work is known as a *resource*. A resource can be either human (e.g. a worker) or non-human (e.g. plant or equipment) and can be owned by either the organisation leading the business process or a third-party entity with whom the organisation has an agreement. A participant then prescribes the class of resource required to perform the activities it encapsulates, rather than the actual resource that performs such activities at enactment-time.

A participant does not specify which actual resource performs its activities. This fact points out that some extra information is required to define the resource *candidates* for the participant that has to be included in the business process definition. Such information could be given at modelling time (referred to as *static*) or be deferred until enactment-time, until the point in time in which the process instance needs to be executed (referred to as *dynamic*).

Specifying the identity of the resources that will be assigned to the participant at enactment-time prevents the problem of unexpected or non-suitable resource allocation arising during the execution of the process instance. The resources to be used by the participants are specified in the call of the business process[35], and not in its definition according to the following (concrete)

[35] The root business process is called from the environment where it is enclosed, whereas inner business process

```
business process diagnosis(...){
    ...
    composite registration(...);
    split composite examination(...),composite makeDocument(...);
    composite consultation(...);
    composite giveInformation(...)
}

business process registration(...){...}
```

Fig. 3.9: *Process Model* of the Diagnosis Business Process.

syntax:

$name_{BP}$[**alloc**(res_1),...,**alloc**(res_n)](...);).

This allows the modeller to implement different resource allocations for the same business process definition. The main drawback of this approach is that if the specified resource is not available, then the process might suffer from unacceptable delays. The deferred allocation approach overcomes this problem, since the decision of which resource must perform the participant's activities is postponed until the moment the instance becomes executable. It is at this point that one resource belonging to the class prescribed by the participant in the process definition is chosen to execute the activities. Therefore, the deferred allocation approach based on the class of resources is considered the by-default strategy to allocate resources to participants. However, any static binding between a particular resource and a participant overrides the default allocation strategy.

It might be necessary to constrain the resource population used to select the resource that will execute the participant's activities. This is achieved by embedding OCL predicates in the resource specification (i.e. $name_{BP}$[**alloc**($r \mid pred_{OCL}(r)$)]()). Furthermore, to allow cross references between resource allocations of different business processes the notion of *resource variable* is introduced. A resource variable is used to store the resource that has been selected by the allocation approach (whatever it is) for performing the participant's activities(i.e. $name_{BP}$[p=**alloc**()]()). Resource variables are defined in the resource allocation area (i.e. the scope defined between the "[" and "]" brackets). A resource variable can be used in any subsequent resource allocation area enclosed by the same participant to associate a *reference* with a formerly used resource.

Resources can also be created upon request. This means that a new resource is created *on demand* to allow a particular process instance to be executed. Creating a resource on demand describes the case in which an organisation relies on some external resources to accomplish their business processes. In this case, the organisation does not directly own all the resources required for the accomplishment of the business process, but it has the chance to borrow them from some external source (e.g. by subcontracting them from a third-party company for the duration of the process instance). It is worth noting that a resource variable can be used to store the reference to a resource created on demand, in case it has to be used to execute another business process within the context of the same participant instance.

It may be of interest to model the particularities belonging to each resource (e.g. previous experience, skills, cost, etc) as such information could be used at enactment-time to select the "best suitable" (according to a certain policy) resource elements for the activities to be performed. The notion of *capability* is introduced to describe the characteristics of interest

are called by composite activities.

that a particular resource may have. Notice that if a capability must be used as a criteria for selecting a resource at enactment-time, the capability should be somehow quantified according to a particular pre-defined metric.

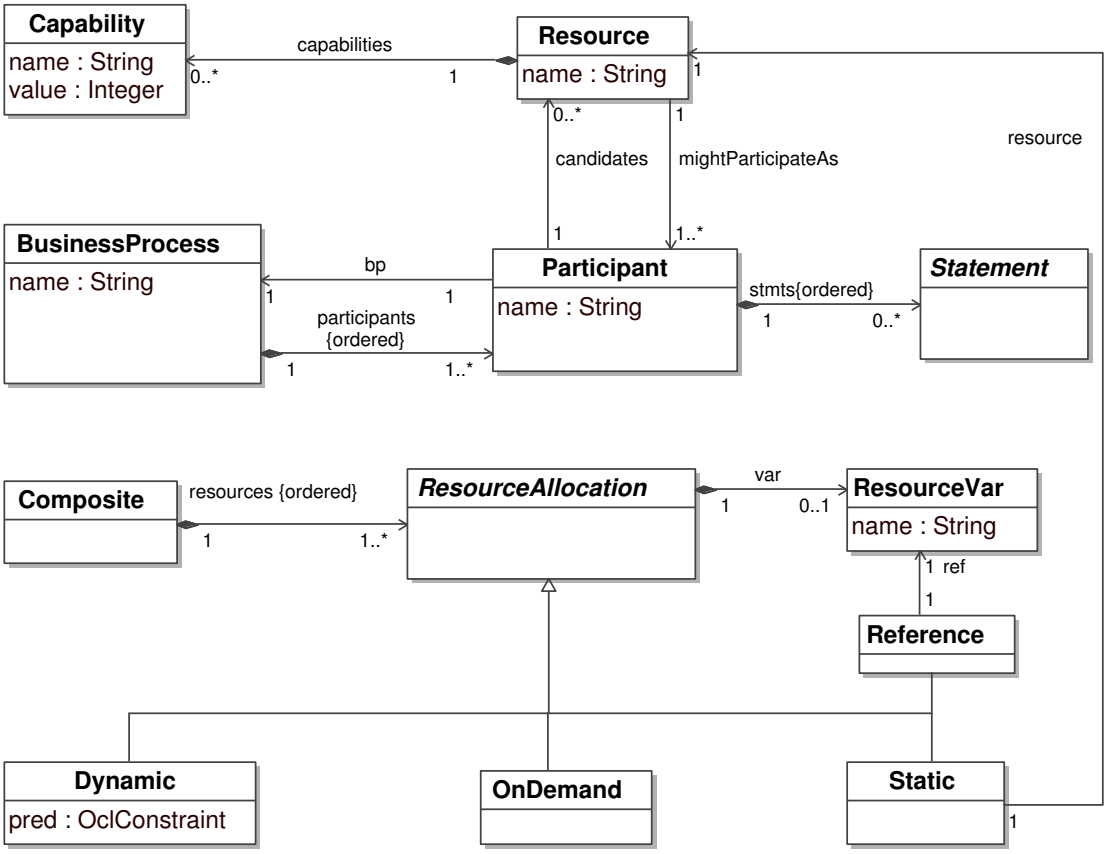

Fig. 3.10: Participant and Resource-related concepts in *DT4BP*.

Figure 3.10 shows how previously introduced concepts, such as participant, resource, resource allocation policy, and resource capabilities, are formalised by means of metamodelling. The entry point for reading this part of the meta-model is the class *BusinessProcess*. A business process has a name (its identifier) and an ordered list of (one or more) participants. Each of these participants is related to only one business process (modelled with the association *bp*) and contains the (ordered) list of statements (modelled with the composite relationship *stmts*) that need to be performed to allow the business process to reach its goal. Notice that the name owned by the participant represents its identifier (which must be unique) within the context of its enclosing business process.

The resources that may execute the statements enclosed by a certain participant are the *candidates* of that participant. Notice that a participant may have none[36] or many candidates (modelled with the association *candidates*), whereas a particular resource must participate in the execution of at least one participant (modelled with the association *mightParticipateAs*).

The capabilities each resource has (which range from none to many) are modelled with the composite relationship *capabilities*. Each capability is modelled as a pair (name, value), where

[36] In this case, a resource is created on demand at enactment-time.

the name is the identifier of the capability (unique within the context of the resource). The value is an integer number that quantifies the resource according to the capability.

Since the allocation resource is defined upon the call of the business process, its modelling must be associated with the place where the call takes place. As previously explained, a business process is called by a composite activity (see part of the meta-model described in Figure 3.6). A composite activity then contains the resource allocation policies (modelled with the composite relationship *resources*) that select each of the required resources to achieve the execution of the process instance at enactment-time. Notice that (1) there must exist as many resource allocation policies as participants in the business process, and (2) the order in which the resource allocation policy is placed determines the participant to be played by the selected resource. A resource allocation policy (modelled by the abstract class *ResourceAllocation*) is either:

- *Static*: a specific resource is selected at modelling time. The association *resource* is used to capture the information about the resource to be used,
- *Reference:* the resource to be used is the one contained in the resource variable being referred to (modelled with the association *ref*),
- *OnDemand:* the resource to be used does not exist, thus it is created at enactment-time,
- *Dynamic:* the resource to be used is the one that satisfies the selected criteria specified by the first-order logic formula contained in the attribute *pred*. In the case that multiple resources satisfy the selection criteria defined by the logical formula, one resource is randomly chosen.

```
1  business process diagnosis() {
2
3    participant DiagnosisUnit{
4       ...
5       composite registration[_,p = alloc(new Patient)](...);
6       split composite examination[p,_,_](), composite makeDocument[_](...);
7       composite consultation[p,_](...);
8       composite giveInformation[p,_](...)
9    }
10 }
11
12 business process registration(...){
13     participant Secretary{...}
14     participant Patient{...}
15 }
```

Fig. 3.11: Participants and resource allocation policies used in the Diagnosis Business Process.

The running example is used again here as a means to demonstrate how the above concepts are used in practice. Figure 3.11 shows how a participant of class *DiagnosisUnit* encapsulates the activities related to the diagnosis process (lines 3-9), while a participant of class *Secretary* (line 13) and another of class *Patient* (line 14) are required by the registration business process to perform its activities. It is also shown how a new resource of class *Patient* will be created at enactment-time (as the specified resource allocation policy is *OnDemand*) to undertake the execution of the registration business process. The keyword **new** followed by the participant name is the way to specify this type of resource allocation policy. The same *Patient* resource that undertakes "registration" must be used to perform "examination", "consultation", and

"giveInformation" business processes. Thus, once the *Patient* resource is created, it is stored in the resource variable *p*. The variable is used upon calls to the other business processes to specify the *Patient* resource to be used by them. An underscore symbol (i.e. '_') is used in the allocation area to denote that the default allocation approach has total freedom to select the resource that will undertake the business process[37].

The different classes of resources (i.e. participants) along with the actual resources that belong to each class are specified in the **Resource Model** view. A business process definition has an unique associated Resource Model view, which is used for the process instances to find the resources they need at enactment-time. Figure 3.12 shows the Resource Model related to the *registration* business process. This Resource Model specifies that there exists only one resource able to play the participant *secretary*, whereas there are no resources for the participant *patient*. This corresponds to the fact that the resource allocation policy for this kind of participant is *OnDemand*.

```
resources{
        Secretary = Ann;
        Patient;
}
```

Fig. 3.12: Resource Model of the Registration Business Process.

In this manner, the resource belonging to a certain class indicates that the resource is a *candidate* for the participant. As previously stated, a resource might belong to more than one class of participant. This means that a resource *might participate* with more than one participant within the same business process, but with only one at the same time during enactment-time. Therefore, a resource r might participate as participant p in certain business process if (and only if) r is one of the candidates of p. Using the meta-model shown in Figure 3.10, this property can be specified in OCL. Figure 3.13 shows such specification:

```
context Participant inv ResourceCandidates:
        self.candidates ->forAll (r:Resource | r.mightParticipateAs -> includes(self))

context Resource inv ParticipantMightPlay:
        self.mightParticipateAs ->forAll (p:Participant | p.candidates -> includes(self))
```

Fig. 3.13: Consistency between resource candidates and potential players of a participant.

3.4.1.4 Data

An activity might require some data to execute. It is also possible that this data is generated by an activity performed earlier. For example, in the running example, the doctor needs to know the temperature and blood pressure of the patient to be able to perform the "consultation" activity. This information is the outcome provided by the "examination" activity. This means that the data required by or produced by an activity is also part of the business process definition, and therefore it must be modelled. Hence, concepts that allow for the representation and utilisation of data to be modelled within a business process are required.

[37] This can also be denoted by using the **alloc()** primitive without argument.

As the data required by a business process depends strongly on the domain that is being targeted, the characteristics of the data may be very simple, or quite complex. A modelling language then should include a means to allow domain specific data entities to be defined. The abstract datatype principle is the chosen means to satisfy this requirement. This means that specific datatypes that match with the domain of the problem being modelling can be specified.

The notion of *type* is used to introduce a new datatype. A datatype defines a collection of data items. These data items are referred to as *objects* of the corresponding datatype. The concrete syntax that allows the modeller to define a new datatype is the following: **type** *typeName*. Attributes may be part of a datatype definition. An attribute has a name and a datatype. Attributes within the same datatype must have distinct names, however, an attribute may have the same name as its enclosing datatype. The attribute's datatype specifies the kind of value it may hold. The following concrete syntax allows modellers to define attributes within a datatype: $typeName\{typeName_1\ attr_1,...,typeName_n\ attr_n\}$). A datatype may also be defined as an ordered sequence of strings. This kind of datatype is referred to as an *enumerated* datatype. The concrete syntax used to define enumerated datatypes is the following: **type** *typeName*= **enum{**$string_1, ..., string_n$**}**.

Basic *primitive* types as *String, Boolean, Float* or *Integer* exist by default. The semantics of each primitive type is the following:

- *Integer:* represents mathematical natural numbers,
- *Float:* represents mathematical real numbers,
- *String:* represents a sequence of (*latin-1* [Int98]) characters, and
- *Boolean:* represents the values *true* and *false*.

It might be necessary to impose certain conditions on a particular datatype such that the possible values that it can represent is constrained. These conditions are known as *invariants*. Invariants are described using first order logic formulas written in OCL. The keyword **where** is used to introduce an invariant within a datatype definition.

It is worth mentioning that the collection of datatypes used within a particular business process are defined in **Data Model** view. Each business process has one associated Data Model. Conversely, objects are declared within the Process Model view of the business process. The name of the object is its identifier, and it must be unique within the context of the Process Model where it is declared.

The scope of an object is determined by the place where it is defined. An object can be defined either within the context of a participant, which is referred to as a *local object*, or as a parameter of a business process in which case it is referred to as a *parameter object*. A parameter object is accessible to every enclosed participant within the same business process, whereas a local object is only accessible for the activities enclosed within the same participant where it is defined. At enactment-time, every time a process instance is initiated, a new object instance is created whether it is a parameter or a local object.

Activities may carry *arguments* (i.e. $act(\textbf{\textit{in}}\ arg_{in_1}, ..., arg_{in_n}; \textbf{\textit{out}}\ arg_{out_1}, ..., arg_{out_m})$). An argument is used either to pass information to the activity (listed in the ***in*** part) or to get side effects of the activity execution (listed in the ***out*** part). Both local and parameters objects are eligible to be used as arguments in an activity. Arguments are passed by value.

The parameters of a business process are also divided between *input* and *output*. Input parameters are used to receive information from the enclosing context and output parameters to return to the enclosing context any useful side effect produced by the execution of the process. Input parameters in the outer most business process are used to capture the information belonging to the environment where the business process operates, and which is required for its execution.

It is worth recalling that an object represents an entity that contains certain information, which is determined by its datatype. The way of assigning information to an object depends on whether it is a parameter or a local object. In the case of parameter objects, the information is assigned upon the business process call. This means that the information held by the arguments is passed over the parameters of the business process once it is called. The case for local objects is different. Assigning information to a local object is part of the duty of an activity. Thus, to assign information to a local object, it has to be passed as an *output* parameter to an activity. Notice that the declaration of a local object only represents the instantiation of the object, i.e. no information is assigned upon the declaration of a local object. When an object is instantiated, it is assumed that its value is "undefined". An object is considered as undefined when the value it holds is not in the domain defined by its datatype. The fact that an object is undefined, thus, means that no value has yet been assigned to it.

The formalisation in terms of metamodelling of the previously introduced concepts is shown in Figure 3.14. This part of the meta-model shows that none or many datatypes are associated with a particular business process. This is modelled by the association *dttps*. Each of these datatypes is either:

- a datatype that has none or many attributes, and may or may not have an invariant (modelled with the classes *DataType* and *Attribute*, and the composite relationship *attrs*),
- an enumerated datatype (in whose case it does not have any attributes) and may or may not have an invariant (modelled with the classes *DataType* and *EnumerationLiteral*, and the ordered composite relationship *enum*), or
- a primitive datatype (modelled with the classes *DInteger, DFloat, DString* and *DBoolean* [38])

The classes *Parameter* and *LocalObject* indicate that an object is either a parameter or a local object. The name of the object is modelled with the attribute *name* owned by the abstract class *Object*. A parameter object is owned by the business process (modelled with the ordered composite relationship *params*), whereas a local object is owned by the participant where it is declared. Notice that a local object is declared with the statement *Objdecl*. This statement allows the modeller to declare one (and only one) local object, and maintains a reference to the object (modelled with the association *var*) to allow further use of the object. Both local and parameter objects can be used as arguments in the activities enclosed by the participants of the business process. This is the reason why the class *Argument* is associated by *obj* with the abstract class *Object*. The composite relationship named *args* contains an ordered list of arguments of the activity. Whether the object is an input or output argument is determined by the attribute *kind*. The type *IOType* is an enumerated type that allows the following values: *in, out* and *in_out*.

Conditions written in OCL are placed over (some) the concepts shown in Figure 3.14 to check whether a particular *DT4BP* model is valid or not. The first condition placed over the meta-model specifies that every valid *DT4BP* model must have every data type defined by either

[38] To avoid name collisions between *DT4BP* and the languages used for its specification, the *DT4BP* primitive types are preceded by a 'D'.

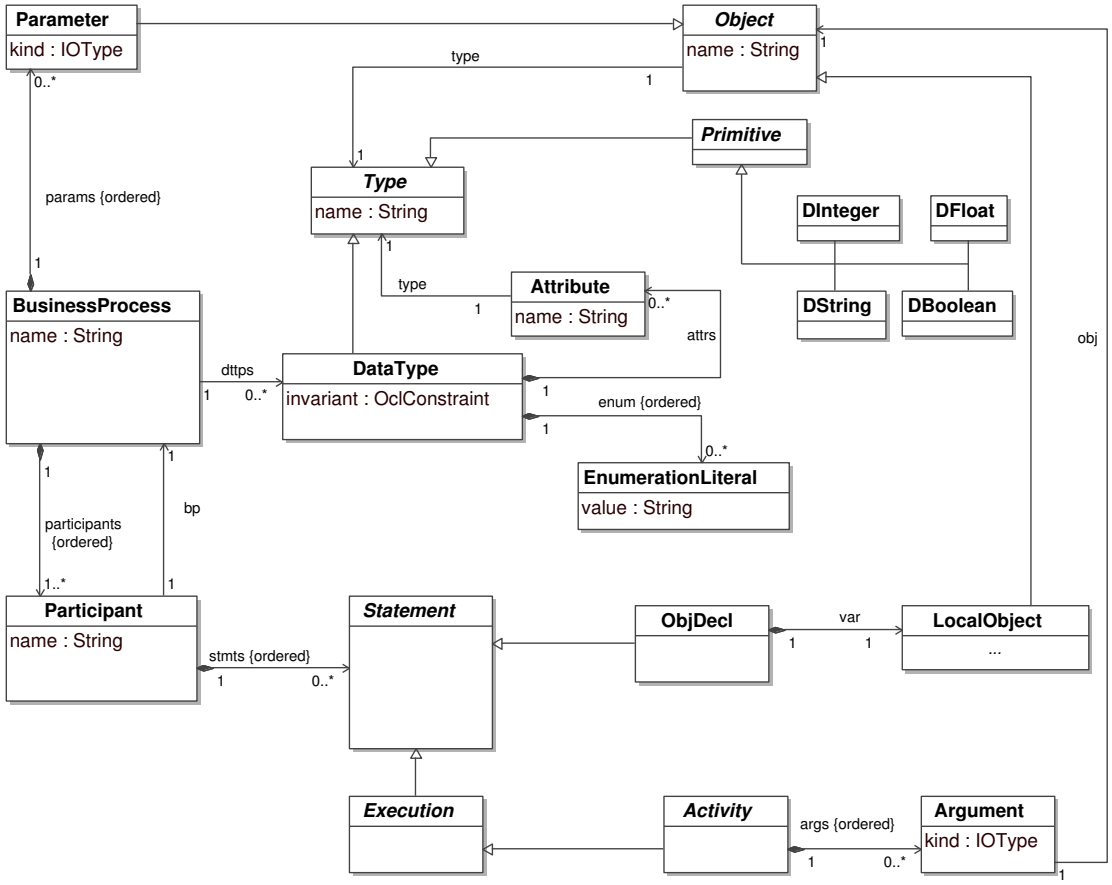

Fig. 3.14: Data and Object aspects in DT4BP.

combining multiple elements of other types, or enumerating an ordered sequence of strings without mixing them. Figure 3.15 shows how this exclusive (aka. *XOR*) manner of defining data types is specified in OCL.

```
context DataType inv XorOnEnumAttr:
    not self.attrs->isEmpty() implies self.enum->isEmpty()
```

Fig. 3.15: OCL invariant over the DataType concept.

It is also possible to determine statically whether the arguments of a *composite* activity are well-typed with respect to the parameters expected by the inner business process the composite activity refers to. The OCL condition that performs this check is shown in Figure 3.16. Notice that this constraint relies on concepts that have been formalised both in Figure 3.14 and Figure 3.6 (regarding to the *composite* activity notion).

Figure 3.17 shows part of the data model associated with the "diagnosis" business process of the running example. This figures shows the definition of the *Temperature* data type: this data type holds an integer attribute *t*. The predicate shown in line 2 defines the allowed values for the attribute *t*.

Figure 3.18 shows some of the local objects defined in the *Diagnosis* process such as *prs*,*t* and *bp*

```
context Composite inv CheckTypesOnParams:
        let
           args: Sequence(Argument) = self.args->asSequence(),
           prms: Sequence(Parameter) = self.call.params->asSequence()
        in
           args->collect(e| e.kind) = prms->collect(e| e.kind) and
           args->collect(e| e.obj.type.oclAsType(DataType).name) =
           prms->collect(e| e.oclAsType(Object).type.oclAsType(DataType).name)
```

Fig. 3.16: OCL invariant for the type-checking.

```
1 type Temperature { Integer t where
2         (30 <= t) and (t <= 50)}
```

Fig. 3.17: Part of the the Diagnosis Business Process Data Model.

(lines 5, 8 and 9; respectively) that are used as arguments in the "registration", "examination" and "makeDocument" activities (lines 6, 11, and 12; respectively). In the same figure, it can be seen that parameter objects are defined in a business process. The "diagnosis" business process has an output parameter object name *ps*, which is of type *PatientSheet* (line 1), whereas the "makeDocument" business process has an input parameter object of type *Person* called *prs* (line 17) and an output parameter object of type *PatientSheet* called *ps* (line 18). Notice that the arguments being passed to the "makeDocument" business process when it is called from "diagnosis" (line 12) are well-typed with respect to the parameter objects specified in its definition (lines 17-18).

```
1  business process diagnosis(out PatientSheet ps){
2
3    participant DiagnosisUnit{
4
5       Person prs;
6       composite registration[_,p = alloc(new Patient)](out prs)
7
8       Temperature t;
9       BloodPressure bp;
10
11      split composite examination[p,_,_](out t,bp),
12            composite makeDocument(in prs; out ps);
13      ...
14   }
15 }
16
17 business process makeDocument(in Person prs;
18                                out PatientSheet ps){...}
```

Fig. 3.18: Parameters, local objects and arguments in business processes.

3.4.1.5 Instantiation

A *process instance* is a particular situation of a business process that takes place at enactment-time. This means that for the same business process *bp* many different process instances $i_1^{bp}, .., i_n^{bp}$ can be created. Each process instance i_j^{bp} is created according to the pattern defined

by the business process *bp*, but each of them represents a different enactment of the business process.

The creation of a process instance (i.e. the instantiation of a business process) happens every time the business process is called. Non-root business processes, i.e. business processes that are not at the top level of the hierarchical structure determined by the process definition, are called by composite activities. Whereas root business processes are called when certain events takes place in its enclosing environment. The event that (at enactment-time) leads to the instantiation of the business process has to be modelled within the business process definition as this is information relevant for the people involved in the management (i.e. design, monitoring and re-design) of such business process. Thus, a root business process must be associated with the event that produces its execution. This is formalised by the meta-model shown in Figure 3.19 along with the OCL constraint described in Figure 3.20.

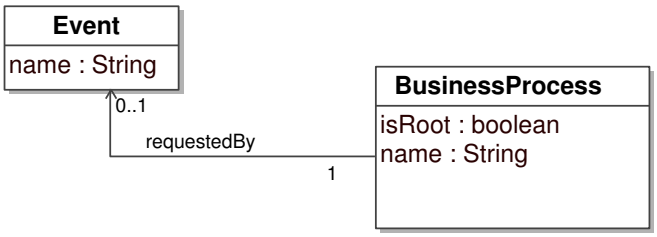

Fig. 3.19: Events in *DT4BP*.

```
context BusinessProcess inv eventForRootBP:
        if (self.isRoot) then
                not(self.requestedBy.oclIsUndefined())
        else
                self.requestedBy.oclIsUndefined()
        endif
```

Fig. 3.20: Root business process must be associated with an event.

The keyword **when** is used to allow modellers to specify the event that lets the business process be instantiated. Figure 3.21 shows how this keyword is used in the running example to model the fact that every time a patient arrives (denoted by the event *patientArrives* -line 2) at the diagnosis unit, a new instance of the *diagnosis* business process is created.

```
1 business process diagnosis(out PatientSheet ps)
2                                when(patientArrives) {...}
```

Fig. 3.21: Event-trigger business process.

Time-related events are also considered as part of those events that can trigger a business process. A time-related event is defined as one that occurs when the *clock* that measures the passage of physical time reaches a certain value. Then, the expression ***when(clock(**timeExp**))*** is used to indicate that an event is triggered when the clock reaches the time value indicated by *timeExp*. The clock is assumed to measure the passage of physical time in the calendar format YYYY-MM-DD:hh-mm, where each component is an integer value such that: (1) the years (i.e.

YYYY) ranges between 1970 and 9999, the months (i.e. MM) between 01 and 12, the days (i.e. DD) between 01 and 31, the hours (i.e. hh) between 00 and 23, and the minutes (i.e. mm) between 00 and 59. Hence, the time value to be used with the primitive **clock** must follow the same calendar format. The clock also provides operators to check whether a certain point in time has been reached by specifying only a sub-part of the calendar format. In this manner, the operators *clock.time(hh-mm)* check whether the clock has reach the time *hh-mm* regardless of the year, month and day. This means that this operator can be used to trigger a daily event. Other operators that work in a similar way are *clock.minute(mm)*, *clock.date(YYYY-MM-DD*, *clock.month(MM*, and *clock.day(DD*. Since every part of the calendar format is an integer, integer expressions obtained by performing certain operations can be also used to specify each of them (e.g. *clock.date((2006+4)-12-15*).

It is worth explaining that the triggering of the event does not imply the actual starting of the process instance as it also depends on the availability of the resources it requires to be performed. Thus, a business process will start its execution not only when the event is fired, but also when the required resources for its execution are available.

3.4.2 Collaboration between participants

As stated earlier, a collaborative business process is defined as one that requires the interaction of multiple participants to reach the process's goal. Therefore, it is expected that definitions of the targeted business processes will contain more than one participant. In the running example, every business process involved in its definition has more than one participant, except "makeDocument" and "diagnosis", which is the outer most business process that encloses all the others[39]. In the targeted business processes, the interaction among participants is expected to be high in the sense that their progress depends on another's progress. Therefore, operational dependencies between collaborative participants are expected to exist.

3.4.2.1 Message exchange

The interaction between participants is achieved by sending and receiving messages. Both sending and receiving a message are considered atomic statements performed within the context of a participant. The reasons why a participant sends a message are:

1. to forward a message to another participant (aka send-no-wait),
2. to synchronise with another participant (aka send-wait), or
3. to request an action from another participant (aka send-receive).

The first case requires sending a message without needing to wait for the recipient to receive the message (i.e send-no-wait)[40]. The second case requires sending a message and then blocking activity until the recipient receives the message (i.e send-wait). The third case requires sending

[39] As already mentioned, it is good modelling practice that the outer most business process includes only one participant, since this allows the reader to have a quick aand simple overview of the business process being modelled.

[40] It is assumed that for any two different participants P_i and P_j, the messages sent from P_i to P_j are received in the order they are sent. It is also assumed that every sent message is eventually received.

a message and then blocking activity only when a reply is needed to continue its execution rather than immediately after sending the message (i.e. send-no-wait-receive).

To address these cases, communication primitives are needed to allow sending and receiving messages along with some optional blocking mechanism that can be attached to them. Obviously, the notions of *send* and *receive* are the concepts that overcome these requirements.

- **Send:** the notion of *send* is used for the sending a message to a particular participant. The sending of a message may or may not be synchronous. In case the sending is synchronous, then the sender participant gets blocked until the receiver participant receives the message. It is worth noting that messages can also be used to exchange information between two different participants. So a message should be allowed to carry data. Thus, *local objects* can be passed as arguments to the message being sent. The only kind of information that is allowed to be passed as an argument with the message is a local object. This is because this is the information that a participant may want to share with their peers (other participants) that are enclosed with the same business process as all of them have access to the business process's parameters (the other kind of information available within a business process). Last, but not least, a message can be sent to more than one participant at once. In this case, it is said that the sender is broadcasting the message to a predefined list of participants.

  The concrete syntax for the sending primitive is:

  send *msg* ***to*** *participant [**block**]*,

  where the optional clause ***block*** is used to determine whether the sending is synchronous or not. When the *send* primitive is used to broadcast a message, more than one participant can be specified as receiver. In case a message conveys local objects, the notation to be used is $msg(lObj_1, ..., lObj_n)$.

- **Receive:** the notion of *receive* is used for retrieving a message sent by a peer participant. A Conversely to send, receive is always synchronous, so that it gets blocked until the message arrives.

  The concrete syntax for the sending primitive is:

  receive *msg* ***from*** *participant*

Therefore, these concepts allow for achieving the (1) "send-no-wait" by using the *send* primitive without the optional clause *block*, (2) the "send-wait" by using the *send* primitive along with the clause *block*, and (3) the "send-received" by using the *send* primitive without the optional clause *block* followed by a *receive* primitive.

As done thus far, the formalisation of the concepts *send* and *receive* as well as with their related aspects is given by means of a meta-model. This formalisation is shown in Figure 3.22 and it describes that the concepts of send and receive (modelled by the classes *Send* and *Receive*, respectively). These kinds of statements a participant may own as a part of all the statements it must execute. That is the reason why the classes *Send* and *Receive* extend from the abstract class *Execution*, which extends from the abstract class *Statement*. The attribute *msg* (of type *String*) is used to model the message to be sent or received, whereas the ordered association named *args* is used to capture the local objects to pass as arguments when either sending or receiving a message. Thus, the same message can be sent to multiple participants. That is the reason why a *Send* statement is linked (by means of the association *to* to one or more

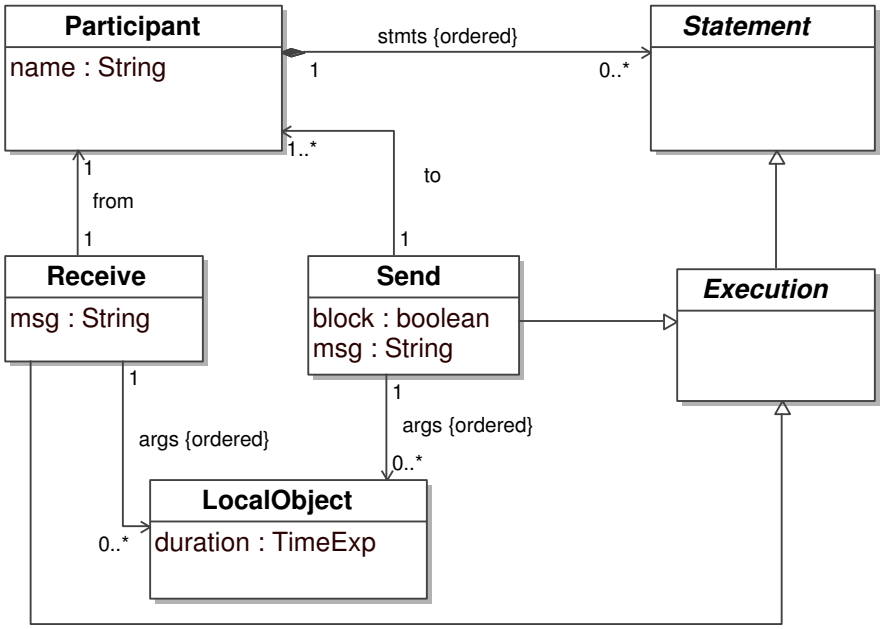

Fig. 3.22: Sending and Receiving messages in DT4BP.

```
context BusinessProcess inv ParticipantsForSendReceive:

  if(self.participants->size() = 1) then
        self.participants-> forAll (p: Participant |
                p.stmts->select(stmt: Statement |
                        stmt.oclIsTypeOf(Send))->size() = 0
        and
                p.stmts->select(stmt: Statement |
                        stmt.oclIsTypeOf(Receive))->size() = 0
        )
  else true endif
```

Fig. 3.23: No send or receive statements in a business process with an unique participant.

participants). Conversely, a *Receive* statement is linked to only the participant that sends the message (modelled by the association *from*).

This part of the meta-model is also used as the context to set OCL conditions that let the modeller know whether the *send* and *receive* concepts are being properly implemented. Since the send and receive concepts are used to exchange messages between peer participants, it must be required that the business process owns at least two participants when such concepts are used. Conversely, when a business process has only one participant, then it is forbidden that the participant includes a send or receive clause in its statements, which is the OCL rule shown in Figure 3.23. Since the context of this rule is the *business process* the reader may need to consult the meta-model components described both in Figure 3.22 and Figure 3.14.

Another obvious control to be performed over any DT4BP model with respect to message exchange is that every participant sends or receives a message from a different peer participant. In other words, a participant cannot send or receive messages from itself. The OCL rule that checks this condition in shown in Figure 3.24.

```
context BusinessProcess inv NotSelfMsgExchange:

  if(self.participants->size()>1) then
        self.participants->
                forAll (p: Participant |
                        (let
                          sends: Sequence(Send) = p.stmts
                                ->select(stmt | stmt.oclIsTypeOf(Send))
                                ->collect(stmt| stmt.oclAsType(Send)),
                          receives: Sequence(Receive) = p.stmts
                                ->select(stmt | stmt.oclIsTypeOf(Receive))
                                ->collect(stmt| stmt.oclAsType(Receive))
                        in
                          sends->forAll( s | s.to -> forAll(p1 | p1<>p)) and
                          receives->forAll( r | r.from <> p)
                        )
                )
  else true endif
```

Fig. 3.24: No participant can send or receive messages from itself.

It must also be checked that the arguments carried by a *sent message* are well-typed with respect to the parameters expected by the receiver of the message. The OCL condition shown in Figure 3.25 implements such rule.

```
context BusinessProcess inv TypeCheckingMsgs:

  if(self.participants->size()>1) then
    self.participants-> forAll (p1,p2: Participant | p1<>p2 and
                (let sends: Sequence(Send) = p1.stmts
                        ->select(stmt | stmt.oclIsTypeOf(Send))
                        ->collect(stmt| stmt.oclAsType(Send)),
                  receives: Sequence(Receive) = p2.stmts
                        ->select(stmt | stmt.oclIsTypeOf(Receive))
                        ->collect(stmt| stmt.oclAsType(Receive))
                in sends->
                        forAll(s | receives->exists(r |
                                (let
                                  type_send: Sequence(String) =
                                        r.args->collect(a | a.type.name),
                                  type_receive: Sequence(String) =
                                        s.args->collect(a | a.type.name)
                                in
                                  type_send = type_receive)))))
  else true  endif
```

Fig. 3.25: OCL invariant for the type-checking of messages with data.

Going over the running example, Figure 3.26 shows how the *send* and *receive* clauses are used to model the interaction between the secretary and the patient during the "registration" process. This process starts by the secretary requesting the patient to provide his social security card. In order to be sure that the patient receives the request, the actions are performed in a synchronous manner. That is the reason why the *send* has a *block* clause at the end (line 3). Once the patient has received the request (line 9), he provides his social security card to the secretary (line 12), who is waiting for that information (line 5).

```
1  business process registration(out Person p)
2    participant Secretary{
3      send reqSSCard to Patient block;
4      SocialSecurityCard ssCard;
5      receive reqSSCard(ssCard) from Patient;
6      ...
7    }
8    participant Patient{
9      receive reqSSCard from Secretary;
10     SocialSecurityCard ssCard;
11     searchSScard(out ssCard);
12     send reqSSCard(ssCard) to Secretary;
13     ...
14   }
15 }
```

Fig. 3.26: Message exchange in the Registration process.

3.4.2.2 Nested activities

It might be the case that a sub-set of the participants involved in a collaborative business process need to perform some sub-collaborative work, seen as an "atomic action" (i.e. single logical unit of work) by any other participant being left out of such sub-collaboration, but who is still enclosed within the same business process. This means that those activities that are required to be performed by every participant engaged in the sub-collaboration have to be scoped and hidden such that participants not engaged in the sub-collaboration see them as a single activity that provides a consistent outcome.

The solution is then to move those activities that define the sub-collaboration to another collaborative business process. Since this new collaborative business process encloses those activities that determine the sub-collaboration, the kind of participants that perform such activities must be also included in its definition. The notion of a *nested* activity is introduced to allow a participant within a collaborative business process to "call" another participant of the same class (i.e. both participant must have the same name), but defined in a different collaborative business process. The call performed by a *nested* activity is semantically equivalent to the call performed by a *composite* activity. Hence, when formalising the notion of *nested* activity by means of the metamodelling principle (which is shown in Figure 3.27), an association named *called* is placed between the class *Nested* and *BusinessProcess* in the same manner as was done between *Composite* and *BusinessProcess*.

The called participant will start performing its activities once every other participant within the same collaborative business process has also been called. Since the kind of called participant must be the same (i.e. to have the same name) as the caller participant, an OCL invariant is placed over the meta-model to ensure such condition (see Figure 3.28).

The resources that play each of the participants in the called collaborative business process are the same as those engaged at the level of the caller. In this manner the availability of the resources to run the nested collaborative business process is guaranteed. Furthermore, it does make sense to use the same resource, since the called sub-collaborative business process behaves as a context dependent sub-part of the enclosing collaborative business process (i.e. the enclosing collaborative must have at least enough participants and of the same class to be able to perform the nesting).

The caller participant will continue performing its activities once the called collaborative busi-

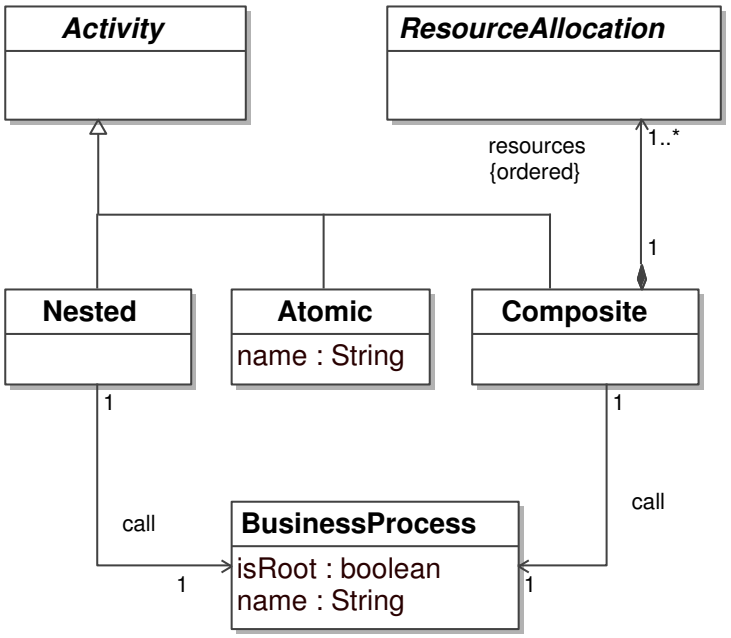

Fig. 3.27: Nested activities in DT4BP.

```
context BusinessProcess inv NestingSameNameCallerAndCalled4:

        self.participants->
                forAll (p1: Participant |
                        let
                          nestings: Bag(Nested) = p1.stmts
                          ->select(stmt:Statement | stmt.oclIsTypeOf(Nested))
                        in
                          nestings-> forAll(n | n.call.participants
                          ->exists(p2:Participant | p2 <> p1 and
                                   p2.name = p1.name))
                )
```

Fig. 3.28: In the nesting, the names between caller and called participants must be the same.

ness process has completed its execution (i.e. every participant within the called collaborative business process has performed all its activities). Therefore, those participants that are called indirectly, using the concept of nested activity, synchronise their execution upon entry and exit to the inner collaborative business process. It is also important to emphasise that a *nested* business process must guarantee the achievement of a consistent outcome even when faced with situations that lead to its goal not being reached. This is required since the overall behaviour of the enclosing business process is dependent upon the outcome. Thus, *nested* business processes are required to provide the isolation and consistency properties.

Such requirements can be fulfilled by making these kinds of business process behave as transactions. As a *nested* business process is a just particular kind of collaborative business process, the same principle (i.e. behave as transactions) can be applied to any collaborative business process. It is worth noting that while the consistency property can be provided in the same manner as is done in ACID-transactions, the isolation property can be achieved only partially due to the long-lived nature of the targeted business processes. The isolation property can be

applied only to those local elements that are not visible from outside of the business process. A deeper analysis of considering business processes as transactions is deferred until Section 3.4.4 when dependability-related concepts are introduced. This is because transaction processing is a well-know approach used to develop fault-tolerant software [GR92].

3.4.3 Time-related aspects

Considering both the requirements and the global time model defined in Section 3.1.3, it is necessary to determine all possible time constraints that should be specified when modelling a business process. The strategy for determining these time constraints consists of considering every concept introduced thus far and deciding the kind of time-related information that can be associated with it. Identifying the possible time constraints allows for establishing which modelling concepts are required to capture this information. In addition, this process highlights which concrete modelling elements should be provided to the language's user such that they may describe time constraints in an easy and direct manner.

3.4.3.1 Timing constraints over a business process

As explained in Section 3.4.1.5, a business process is initiated by a request. The request is either an external event coming from the environment where it operates[41] or a simple call (like the one made by *composite* or *nested* activities).

The point in time when the request to a business process is performed (t_r in Figure 3.29) can be used as the temporal frame of reference when setting time constraints over the business process. Another point in time that can be used as a frame of reference is the actual time at which the business process begins its execution (t_s in Figure 3.29). Both frames of references then are used to analyse the time constraints that may be set over a business process.

- **Starting:** taking t_r as the frame of reference, the actual initiation of a business process could be constrained in the following way:
  - the process starts t_s time units after t_r (t_s is the *delay*),
  - the process starts at least $t_{s_{min}}$ time units after t_r ($t_{s_{min}}$ is the *minimum delay*),
  - the process starts no later than $t_{s_{max}}$ time units after t_r ($t_{s_{max}}$ is the *maximum delay*),

  The concept *start* then is used to capture such potential time constraints. The keyword **start** is proposed as the primitive to concretely specify constraints related to the start of the business process. This primitive may be followed by either a single time value t_s or a pair of time values $[t_{s_{min}}, t_{s_{max}}]$. *Start* followed by a single time value (i.e. ***start*** t_s) is used to specify the exact delay in starting the business process. Whereas if it is followed by a pair of constraints (i.e. ***start*** $[t_{s_{min}}, t_{s_{max}}]$), it is used to specify the minimum delay ($[t_{s_{min}}, \_]$), the maximum delay ($[\_, t_{s_{max}}]$), or both ($[t_{s_{min}}, t_{s_{max}}]$).
- **Periodicity:** in general, there is no information about how often the requests that initiate a business process arrive (i.e. request patterns are not known in advance). An exception to this rule is the *periodic* business processes. A periodic business process is performed

[41] Time-related events are included in this consideration. These events are launched automatically by the built-in clock that measures the passage of the time as observed in the physical world.

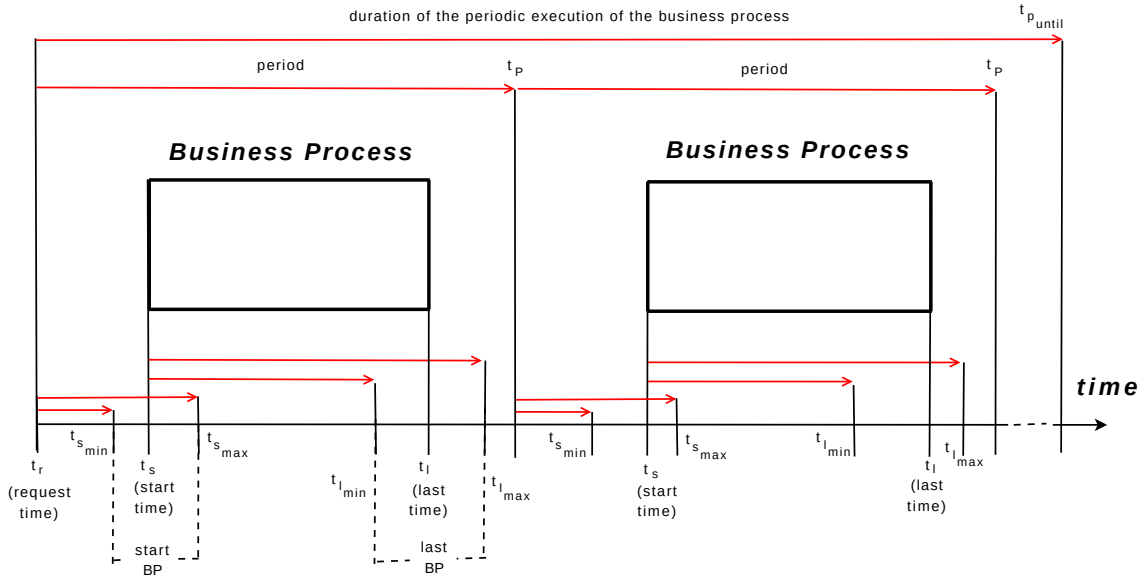

Fig. 3.29: Timing constraints over a business process.

according to a recurring pattern, which specifies how often the business process is requested and for how long the recurring patterns has to be applied (if any). However, it must be noted that a periodic business process must be initially requested by a certain (either internal or external) event. The point in time when the initial request takes place (i.e. t_r) is the frame of reference both for the periodicity (i.e. t_P in Figure 3.29) and marks when subsequent requests are made (i.e. $t_{P_{until}}$ in Figure 3.29).

The primitives **every** and **until** are used to specify information related the period of the process and the end of the periodic sequence, respectively. Thus, while ***every*** t_P is used to specify how often the business process is requested, ***until*** $t_{P_{until}}$ specifies the termination of the requests.

It is worth noting that (1) the timing constraints relative to the start of the business process are considered in the first and every subsequent periodic request of the business process; (2) periodic business process might last longer than the value specified using the **until** primitive as it constrains the duration the periodic business process is requested, and not its duration; and (3) the period of a business process (i.e. t_P) defines not only the point in time at which the next business process is going to be requested, but also the maximum elapsed time for the current business process under execution as a periodic business process must complete its execution within the period.

- **Duration:** taking as the frame of reference the actual point in time at which the business process starts its execution (i.e. t_s in Figure 3.29), then the duration of the business process can be constrained in the following way:
  - the process lasts exactly t_l time units (t_l is the *elapse time*)
  - the process lasts at least $t_{l_{min}}$ time units ($t_{l_{min}}$ is the *minimum elapse time*)
  - the process lasts no more than $t_{l_{max}}$ time units ($t_{l_{max}}$ is the *maximum elapse time*)

  In this case, the concept *last* is used to capture the constraints related to the duration of the business process. The modeller has to use the primitive **last** to specify this information.

In the same way as was done previously, this primitive followed by a single time value (t_l) specifies the exact duration of the process, whereas if it is followed by a pair ($[t_{l_{min}}, t_{l_{max}}]$) it specifies the minimum and maximum elapse times for such process.

```
1 business process diagnosis(out PatientSheet ps)
2                  when(patientArrives) last[_,2hs.] {...}
```

Fig. 3.30: Constraining the duration of the Diagnosis process.

In the running example, the duration of the "diagnosis" process has to be constrained, as it is required that "the patient should get its diagnostic in less than two hours". Figure 3.30 shows how this requirement is modelled using the concept *last*.

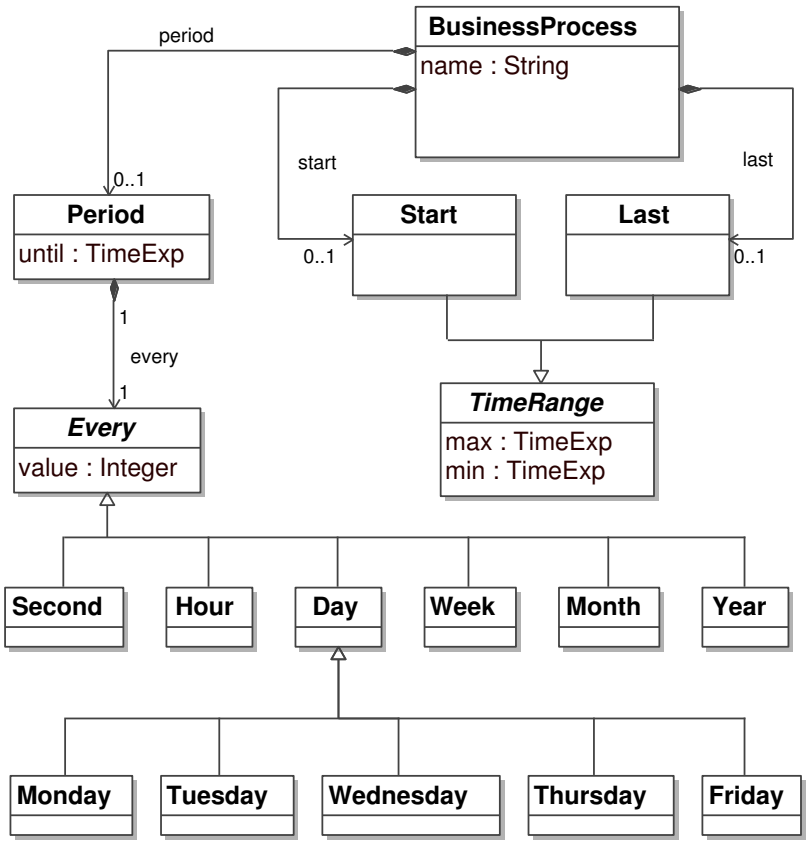

Fig. 3.31: Time concepts related to a business process.

Figure 3.31 shows the formalisation of these time-related concepts according to the metamodelling principle. This formalisation, thus, specifies that a business process (of type *BusinessProcess*) may or may not have its starting time constrained. When this constraint exists, it is held by the composite relationship *start*. Notice that only one start constraint may be associated with a business process. In addition, the constraint (since *Start* extends from the abstract class *TimeRange*) and may restrict the minimum delay (specified with the*min* attribute), the maximum delay (specified with the*max* attribute), or both. The particular case in which the attributes *min* and *max* have the same time values represents the modelling of an exact delay.

It is worth noting that the time values these attributes may have are of type *TimeExp*. This datatype defines a time value as a positive integer.

Regarding the duration of the business process, it is specified by means of the class *Last*, which also extends from the *TimeRange* and the composite relationship *last*. When a business process has its duration constrained, then the composite relationship must contain one (and only one) instance of the class *Last* that has, at least one of the attributes *min* and *max* different from zero. The *min* attribute contains the time expression that determines the minimum allowed duration of the business process, whereas *max* represents the maximum allowed duration.

The remaining part of the meta-model is concerned with the formalisation of the business process periodicity (if any). In case a business process is periodic, then it must have a period, which is modelled by the composite relationship *period*. A period (of type *Period*) has two time values. One time value, modelled by the composite relationship *every*, describes how often the business process has to be requested (once it is started). The other time value, modelled by the attribute *until*, describes the time up to which these requests have to be performed. Notice that the instance held by the composite relationship *every* is an instance of one the classes that extends from the abstract class *Every*.

```
context TimeRange inv ConsistentRange:
  (self.min > 0 or self.max > 0)
  and
  (if(self.min > 0 and self.max > 0)
        then self.min <= self.max else true endif)

context BusinessProcess inv ConsistentPeriod:
  if(self.period.every.value > 0) then
        if(self.start.min > 0) then
                if(self.last.min > 0) then
                        self.start.min + self.last.min <=  self.period.every.value
                else
                        self.start.min <=  self.period.every.value
                endif
        else
                if(self.last.min > 0) then
                        self.last.min <=  self.period.every.value
                else
                        true
                endif
        endif
  else
        true
  endif
```

Fig. 3.32: Consistency between time-related values at the level of the business process.

Figure 3.32 shows the OCL conditions that ensure the values held by the previously mentioned time-related concepts are such that there exist at least one process instance that meets these conditions. Such OCL conditions are specified over the assumption that all time-related values are in the same time unit. This assumption can be achieved by performing a normalisation of the time-related information, that takes place during the parsing of the model.

The invariant *ConsistentRange* then not only ensures that every instance of the sub-type *TimeRange* has at least one of its attributes with a value greater than zero (otherwise it does not make sense to create an instance), but also that when both attributes are different from zero that *max* is greater or equal to *min* (the case *max=min* represents an exact delay or duration of the business process).

On the other hand, the invariant *ConsistentPeriod* checks that period is greater than or equal to the minimum delay (if any), the minimum elapse time (if any), and the sum of both in the case that both minimums are greater than zero. Notice that this invariant will not hold in the case that the business process has to (1) wait for a time that goes beyond the period, (2) last longer than the period, or (3) wait and last for a time that exceeds the period. Since such cases only describe invalid process instances they should not be model.

3.4.3.2 Timing constraints over a participant

The time a participant takes for completing its enclosed activities (including waiting times between activities) may also be constrained [42] . The frame of reference for this time constraint is determined by the point in time at which the business process starts its execution, t_s in Figure 3.33. Notice that a business process starts executing only once all its required resources have been allocated. It is assumed that the time taken to allocate the resources is negligible.

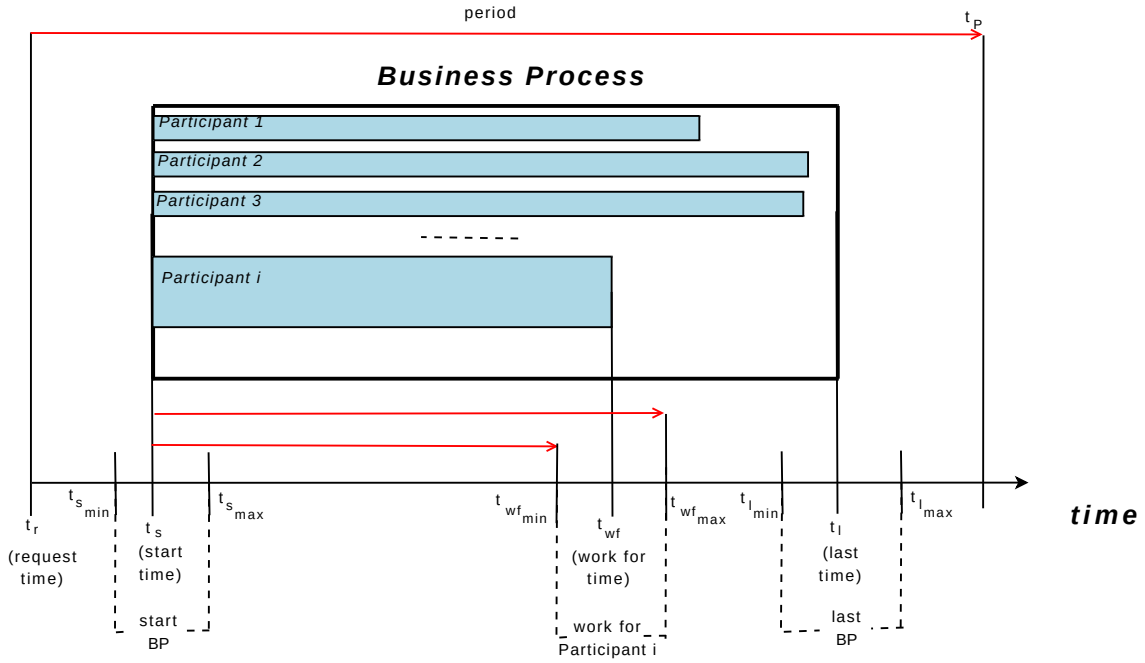

Fig. 3.33: Timing constraints over a participant.

The notion of *workFor* is used to capture the minimum time (i.e. $t_{wf_{min}}$ in Figure 3.33) and maximum time (i.e. $t_{wf_{max}}$ in Figure 3.33) a participant may be engaged in executing its enclosed activities. The primitive that allows its modelling carries the same name. This primitive followed by a pair of values $[t_{wf_{min}}, t_{wf_{max}}]$ specifies the minimum and maximum working times the participant has to execute its enclosed activities. As usual, the symbol '_' is used to leave without specifying one of the values.

Figure 3.34 shows how this primitive is used to specify the requirement that a doctor should work in the examination of the patient for at least 15 minutes in order to ensure a certain level of quality in the examination. Notice that the time information held by the **workFor** primitive is concerned with the time the resource will be engaged for completing the activities enclosed

[42] Actually, this time constraint has an effect on the resource that is assigned to the participant at runtime.

by the participant, whereas the primitive *last* is with the duration of "consultation" business process.

```
1  business process consultation(
2              in PatientSheet ps, Temperature t, BloodPreasure bp;
3              out PatientSheet ps) last[15min.,1hs.]{
4
5    participant Patient{
6          ...
7    }
8    participant Doctor {
9          ...
10   }workFor[15min.,_]
11 }
```

Fig. 3.34: Constraining the working time of Doctor participant.

Figure 3.35 shows the part of the meta-model that formalises the time-related concept *workFor*. This formalisation indicates that a participant may or may not have its working time constrained. This is modelled by the composite relationship *workFor*, which allows a particular participant (of type *Participant*) to be bound with an instance of class *WorkFor*. Every instance of class *WorkFor* has attributes *min* and *max*. These attributes own the time-related values that specify the minimum and maximum times the participant is allowed to work, respectively.

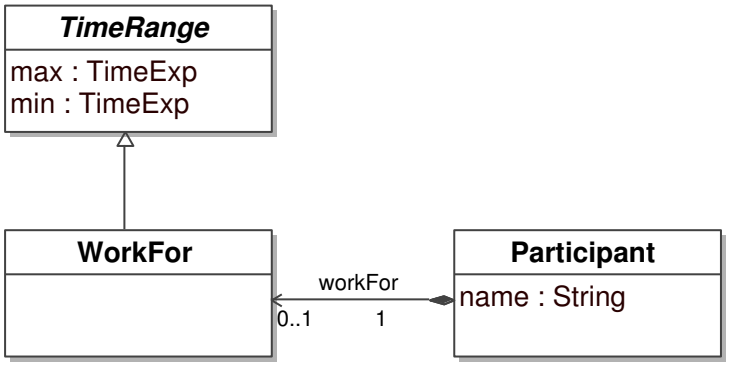

Fig. 3.35: Time concepts related to a participant.

Notice that the minimum time duration that participants may work should not exceed the maximum elapse time of the business process where they are embedded, otherwise it is impossible to create any valid process instance that meets both time constraints. In the case that the participants are part of a periodic business process, their minimum *workFor* time should not exceed the period of the business process (i.e. t_P in Figure 3.33). The OCL invariants that ensure the consistency between these time-related values are shown in Figure 3.36. The invariant *PartWorkForMinElapseMax* checks the minimum *workFor* time is always lower than or equal to the maximum elapsed time. Whereas *PartWorkForMinPeriod* does the same but for the period.

3.4.3.3 Timing constraints over an activity

Taking as the frame of reference the point in time at which an (whether *composite, nested* or *Atomic*) activity ends (finish of act_1 in Figure 3.37) or, by default, the initiation of its enclosing

```
context Participant inv PartWorkForMinElapseMax:
        if(self.workFor.min > 0 and self.bp.last.max > 0) then
                self.workFor.min < self.bp.last.max
        else true endif


context Participant inv PartWorkForMinPeriod:
        if(self.workFor.min > 0 and self.bp.period.every.value > 0) then
                self.workFor.min < self.bp.period.every.value
        else true endif
```

Fig. 3.36: Consistency between Participant and Business Process time-related values.

business process, an activity may be constrained such that it has to wait for at least $t_{act_{delay}}$ time units (aka delay) before starting its execution. It is worth explaining that both *composite* and *nested* activities are considered as starting immediately in spite of the business processes these kinds of activities refer to may not start their execution immediately (e.g. the business process holds a delay).

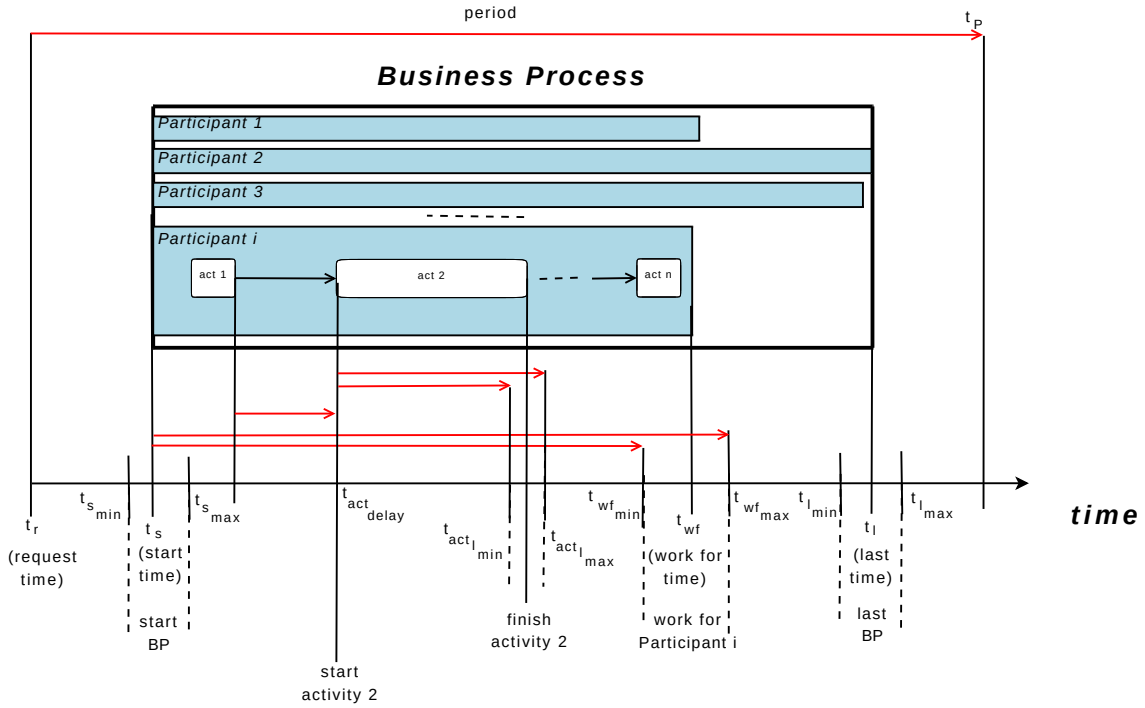

Fig. 3.37: Timing constraints over an activity.

As it is assumed that the execution of certain activity takes time, the actual time it takes to execute is called the activity duration. This duration can be explicitly constrained by specifying the minimum amount of time (i.e. $t_{act_{l_{min}}}$ in Figure 3.37) the activity must be under execution (aka earliest deadline) and/or the maximum allowed time (i.e. $t_{act_{l_{max}}}$ in Figure 3.37) to be spent performing this activity (aka latest deadline).

According to the previous analysis, it may be required to specify (1) *in* which point in time the activity has to start and (2) the point in time *within* which the activity should finish. While the former is used to capture the information related to the delay in requesting the activity, the

latter captures the activity's deadlines.

The formalisation in terms of the metamodelling of the *in* and *within* concepts is shown in Figure 3.38. The composite relationships *in* and *within* owned by the class *Activity* (by inheriting from class *Execution*) allow an activity to capture the information regarded as its delay and duration, respectively. Notice that the duration of an activity is defined by a range of time values, whereas the delay is defined by only one. Hence, the duration of an activity (modelled as an instance of class *Within*) has two time values (i.e. attributes *min* and *max*), whereas the delay (modelled as an instance of class *In*) has only one time value (i.e. attribute *delay*).

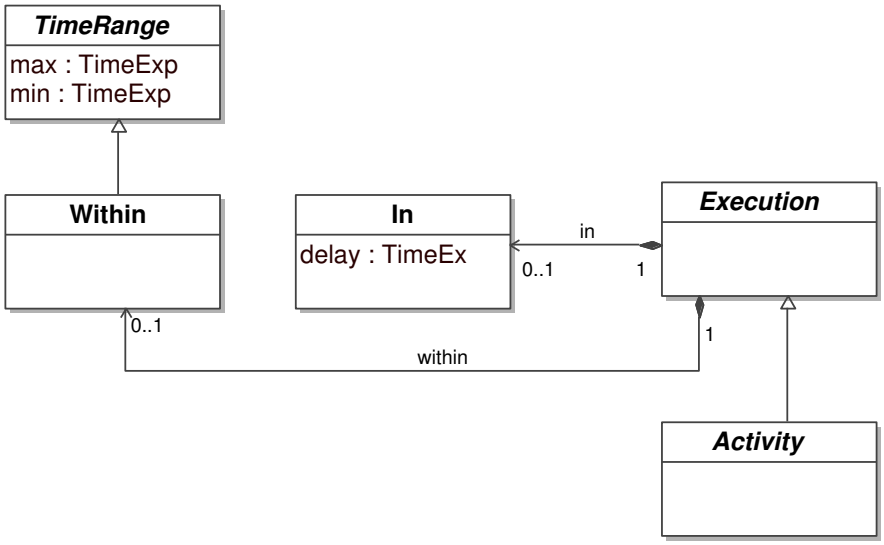

Fig. 3.38: Time concepts related with the notion of Activity.

Primitives with the same name are provided to allow the language's user to describe such information. In this manner, **in** allows the modeller to describe information related to the start of an activity. This primitive must be followed by a single time value $t_{act_{Delay}}$ which specifies the delay in starting the activity. On the other hand, the primitive **within** is used to allow the modeller to specify the time information that constrains the duration of the activity: i.e. its earliest and latest deadlines. Following the same strategy when minimums and maximums need to be specified, a pair of time values following the primitive **within** is used to constrain the duration of the activity. Thus, the pair $[t_{act_{l_{min}}}, t_{act_{l_{max}}}]$ indicates that the activity executes for at least $t_{act_{l_{min}}}$ time units and it does not take more than $t_{act_{l_{max}}}$ time units. As used so far, the '_' symbol means that no constraint is set over the time value where this symbol is used. Figure 3.39 shows how this primitive is used to model the duration of the activity "consultation" according to the given requirement: "consultation' has to take less than 1 hour, but more than 15 minutes.

It is worth noting that time constraints set over *composite* or *nested* activities, i.e. values specified with *within*, must be consistent with those time constraints that might have been set in the definition of the business processes they refer to, e.g. *last* values. Thus, in the case that a *composite* or *nested* activity has its maximum allowed duration constrained (i.e. **within**$[\_, t_{act_{l_{max}}}]$) and the business process being referred to is either (1) periodic (i.e. *business process bp* **every** t_P **until**) $t_{P_{until}}$ or (2) its minimum elapse time is constrained (i.e. *business process bp* **last** $[\_, t_{l_{max}}]$), then:

```
1 business process diagnosis(out PatientSheet ps) when(patientArrives) last[_,2hs.] {
2
3   participant DiagnosisUnit {
4       ...
5       composite consultation[p,_](in ps,t,bp; out ps) within[15min.,1hs.];
6       ...
7       }
8 }
```

Fig. 3.39: Constraining the duration of the activity *consultation*.

- for case (1): the maximum allowed duration of the *composite* or *nested* activity has to be greater than or equal to the sum of the duration of the periodic execution of the business process and its period (i.e.$t_P + t_{P_{until}} \leq t_{act_{l_{max}}}$)
- for case (2): the maximum allowed duration of the *composite* or *nested* activity has to be greater than or equal to the sum of minimum delay (i.e. **start**$[t_{s_{min}}, \_]$) set at the start of the referred business process (if any) and its minimum elapse time (i.e. $t_{s_{min}} + t_{l_{max}} \leq t_{act_{l_{max}}}$)

The OCL invariants that ensure these conditions are shown in Figure 3.40. The reader should notice that no consistency is required between the time constraints defined by the concepts *in* (at the level of *composite* or *nested* activities) and *start* (at the level of the referred business process). The concept *in* is used to constrain the request to start executing the activity, whereas *start* is used to constrain the actual starting of the business process.

```
context Composite inv ConsistencyWithinLastPeriod:

  if(self.within.max > 0 and self.call.period.every.value > 0) then
        if(self.call.period.until > 0) then
          self.call.period.until + self.call.period.every.value <= self.within.max
        else
          self.call.period.every.value <= self.within.max
        endif
  else true endif

  and

  if(self.within.max > 0 and self.call.last.min > 0) then
        if(self.call.start.min > 0) then
          self.call.start.min + self.call.last.min <= self.within.max
        else
          self.call.last.min <= self.within.max
        endif
  else true endif
```

Fig. 3.40: Consistency between *within*, *last* and *period* time-related values for a composite activity.

Another condition that may be checked at modelling time is that the sum of the activities' minimum delays and/or durations (if any have been specified) has to be lower than or equal to (1) the maximum allowed working time of the participant that encloses the activities (if any), (2) the maximum allowed elapse time of the business process (if any), and (3) the period of the business process (if any). Figure 3.41 shows the OCL conditions that perform these checks.

```
context Participant inv MinGuaranteedET:
  let
        execs:Sequence(Execution) = self.stmts->collect(s|s.oclIsKindOf(Execution)),
        minWithinET:Integer= execs->collect(e|e.within.min > 0)-> sum(),
        minInET:Integer= execs->collect(e|e.max.min > 0)->sum()
  in

        if(self.workFor.max > 0) then
                minWithinET + minInET <= self.workFor.max
        else true endif

        and

        if(self.bp.last.max > 0) then
                minwithinET + minInGET <= self.bp.last.max
        else true endif

        and

        if(self.bp.period.every.value > 0) then
                minwithinET + minInGET <= self.bp.period.every.value
        else true endif
```

Fig. 3.41: Consistency of the sum of activities' max/min delays with respect to their enclosing context.

3.4.3.4 Timing constraints when exchanging messages

The exchange of messages between participants is captured by the *send* and *receive* concepts (see Section 3.4.2 for details about these concepts). Time constraints over these concepts may be set to specify:

1. how long a (blocking) sending participant is willing to wait for the recipient to get the message,
2. how long the recipient is willing to wait for a message to arrive,
3. how long the recipient executes their activities to produce a reply after a message has been received, and
4. how soon a reply should arrive to a sending participant after a message has been sent.

The timing constraints (1) and (2) can be modelled using the concept *within* introduced in the previous section. More precisely, a participant p_{sender} holding the statement ***send*** *msg* ***to*** $p_{receiver}$ ***block within[_,t]*** describes the condition that the p_{sender} is willing to wait for the $p_{receiver}$ to get the message *msg* for t time units, maximum. On the other hand, the participant $p_{receiver}$ holding the statement ***receive*** *msg* ***from*** p_{sender} ***within[_,t]*** describes the fact that $p_{receiver}$ is willing to wait for message *msg* to arrive at t time units, maximum.

However, with the concepts introduced so far, it is not possible to model constraints (3) nor (4). For capturing such timing constraints, a new concept is required that allows for the combination of a set of activities within a common *block* such that one timing constraint can be bound to a group of activities $act_1, ..., act_n$, rather than only one. The notion of *block* is introduced to set a common timing constraint over a set of activities. Since an *activity* is a *statement* (see the part of the meta-model shown in Figure 3.8), in order to be more precise without loosing

generality, the concept of *block* is used as a structuring means to join a set of statements rather than activities. In this manner, the notion of *block* allows for the definition of an activity[43] within a *participant* that actually is composed of a a set of *statements*.

The formalisation of the timing constraints over the concepts *send* and *receive*, as well as the notion of *block* are shown in Figure 3.42. The approach taken for the formalisation of the timing aspects with regard to the *send* and *receive* concepts is exactly the same followed to formalise the time aspects of the activity concept. In this manner, the classes *Receive* and *Send* are inherited from the abstract class *Execution* such that the composite relationships *in* and *within* are part of their definition. The same is done for the concept *block* to allow a set of activities be constrained using the notions of *In* or *Within*. Hence, the class *Block* also extends from class *Execution*. It is worth noting that the ordered set of statements owned by a particular block, which is modelled by the ordered composite relationship *stmts*, may not only contain activities, but also any other kind of statement such as sends, receive, or even control-flow operators.

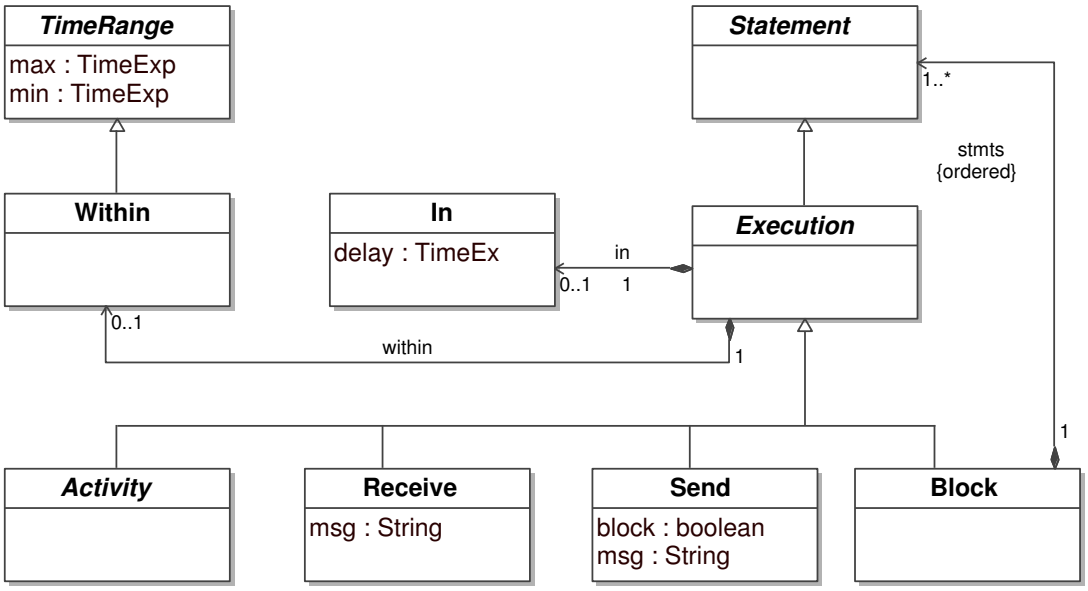

Fig. 3.42: Time concepts related to the notions of Activity, Block, Send and Receive.

The concrete representation of the *block* concept is achieved by the primitive **do{...}**. In this way, assuming that $a_1, ..., a_n$ are all the activities that a participant must execute to produce the expected reply, the statement $\boldsymbol{do}\{a_1, ..., a_n\}$ ***within[_,t]***[44] allows constraint (3) to be modelled. In the same way, assuming that $a_1, ..., a_n$[1] now represents the activities that follow a (no blocking) *send*, the constraint (4) can be modelled. Figure 3.43 shows how the primitives **do** and **within** are used within the running example to model the constraint "the parallel activities "examination" and "makeDocument" should execute in less than 30 minutes".

[43] In the BPMN language, this activity is known as an *embedded process*.
[44] This statement means that t is the maximum allowed time for executing the activities $a_1, ..., a_n$.
[1] The activity a_n must be the one in charge of receiving the message.

```
1  business process diagnosis(out PatientSheet ps) when(patientArrives) last[_,2hs.] {
2
3
4    participant DiagnosisUnit {
5       ...
6       do{split composite examination[p,_,_](out t,bp),
7          composite makeDocument(in prs; out ps)
8       }within[_,30min.];
9       ...
10      }
11 }
```

Fig. 3.43: Constraining a block of activities.

3.4.3.5 Timing constraints over data

The information carried by an object might be valid for only a limited time after its last update (e.g. the object that holds the patient's temperature). This type of time-related information associated with objects is aimed at specifying a property p that determines for how long the object's information will be valid. This time-related information then defines a deadline for the information being carried for that particular object, which is relative to the last time it was updated.

A timing constraint over an object (if given) is meant to define a time frame which determines the validity of the information placed within the object. Since this validity time frame is relative to the moment at which the object was last updated, its specification is given by providing the *duration* of the time frame. As this timing constraint is associated with an object, it must be given at the same time the object is declared using the concept *ObjDecl*. This issue points out that the only kinds of objects that may own a *duration* are local objects. In this manner, when declaring a local object its duration (captured by the attribute with the same name) can be defined as well. The formalisation of the notion of *duration* as a means to place a timing constraint over a particular local object is shown in Figure 3.44.

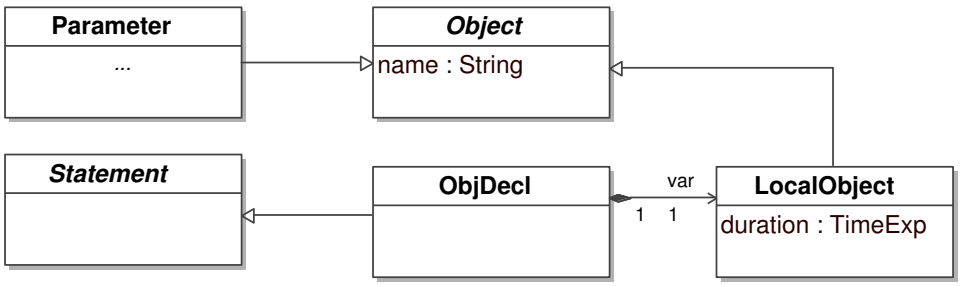

Fig. 3.44: Duration of the local object's data.

The concrete modelling of this time constraint is achieved using the primitive **expire** (i.e. *typeName localObject* ***expire(timeExpr)***). Once again, the running example is used as a means to show how the primitive is used. The patient's measured temperature and blood pressure are considered as valid until one hour after they have been taken. The modelling of this requirement using the primitive **expire** is shown in Figure 3.45.

```
1  business process diagnosis(out PatientSheet ps) when(patientArrives) last[_,2hs.] {
2
3
4    participant DiagnosisUnit {
5       ...
6       Temperature t expire(1hs.);
7       BloodPressure bp expire(1hs.);
8       ...
9       }
10 }
```

Fig. 3.45: Modelling time constraints over data elements.

3.4.3.6 Timing constraints over resources

Time constraints over resources are used to specify their availability. Absolute time constraints would be the rule rather than the exception when specifying the availability pattern for certain resources.

The last kind of time constraint that could be required to be modelled are those used to specify the availability of a resource. Such time constraints determine the availability of a resource by means of temporal patterns (e.g. Mondays and Fridays, or Week days from 8:00 to 16:00, etc.) combined with valid periods (i.e. each pattern is valid only for certain period of time (e.g. from 2009-01-01 to 2011-12-31). Since these patterns may be very different, the time constraints used to model them are complex, and not easy to read or understand. Furthermore, many different primitives should be given in order to allow such patterns to be modelled. The way to cope with this modelling issue is to rely on the notion of *calendar*[45] as the means to describe the availability of a resource. In this manner, instead of using patterns to model the time frames for which a particular resource is available, this information is now explicitly included[46] in a calendar. Therefore, the availability of a particular resource can be known by means of its associated calendar (if any). Figure 3.46 shows the part of the meta-model that formalises this association, which is modelled as a composite relationship named *availability*. It is worth noting that a resource without an associated calendar is considered as the one that is always available.

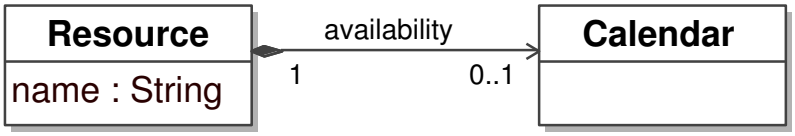

Fig. 3.46: Calendar as means to specify the availability of a resource.

3.4.3.7 Control flow operators

The remaining concepts to be analysed are: *control flow operators* and *event*. As it is assumed that a *control flow operator* does not take time to be performed, it does not make sense to set time constraints over them.

[45] Dates and times are represented using the Gregorian calendar as it is the international standard.
[46] The way in which information is set in a calendar is beyond the scope of this work.

3.4.3.8 Events

An *event* is used to specify the circumstances that make a business process start its execution. Thus, a time constraint set over an event can be see as a constraint over the start of the business process that such an event triggers. Therefore, rather than attaching time constraints to an event, the approach is to use such events as frames of reference for the time constraints that can be set over the start of a business process (i.e. exact, minimum, and maximum starting delay).

3.4.4 Dependability

A *dependable business process* has been defined as one that either (preferably) does not miss its goal unacceptably frequently or, if it does, the consequences are not unacceptably severe. This section introduces those concepts that allow modellers to derive explicit dependability when modelling business processes.

3.4.4.1 Pre- and post-conditions

A business process that misses its goal is considered as one that has failed. Without having a precise definition of the business process's goal, to judge whether it has failed or not may lead to ambiguous answers. Since precision is needed, a business process goal has to be described using certain formal notation. As done so far, OCL is the chosen notation to meet this requirement. It is also important to state clearly under which conditions such goal is expected to be reached by the business process. These conditions are expected to be fulfilled by the "client" that requests the business process. The relationship between the business process and its clients can be considered as a contract in the same sense as Meyer does in the "Design by Contract" approach [Mey97]. The *pre-condition* of the contract defines what every client must satisfy to achieve the business process' goal. Thus, the goal is the *post-condition* of the contract. A process instance that satisfies the business process pre-condition is expected to achieve its post-condition (or business process' goal), eventually.

The notion of contract also helps in separating the responsibilities in case the contract is broken (i.e. either the pre-condition or the post-condition is not satisfied). A business process that is called without satisfying its pre-condition represents a fault of the client. Conversely, a business process that, once started, does not achieve its post-condition represents a fault of the business process provider. It is worth noting that a process instance is created only if the business process pre-condition is met. For the case when the contract is broken due to a fault of the provider, represents a process instance failure as the goal (i.e. the post-condition) is not achieved.

Figure 3.47 shows the part of the meta-model that formalises the notions of pre- and post-condition. Both a pre- and a post-condition are considered as first-order logical formulas written in OCL. Each condition then is modelled as a class that contains an attribute called *predicate* of type *OclConstraint*, which are aimed at storing the logical formula the condition represents. The composite relationships *pre* and *post* model the fact that each business process must have both a pre- and post condition. As already stated, the post-condition describes the business process goal. Hence, this post-condition is what the business process has to achieve to be considered as having had provided a *normal* outcome to the client that requests such business process. That is the reason why the business process post-condition has been modelled with a class termed *Normal* that extends from the abstract class *Outcome*. Further explanations regarding the business process outcome are given in the next section.

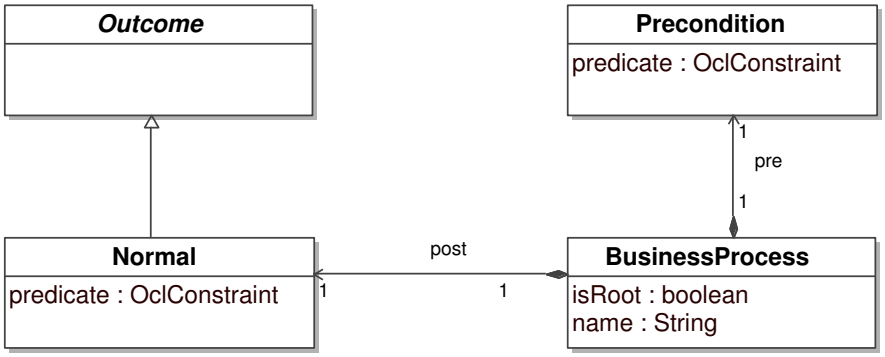

Fig. 3.47: Pre- and post-condition in a business process.

The context for pre- and post-conditions is determined by the input and output parameter objects of the business process. The pre-condition expression may only refer to input parameter objects as it specifies the applicability of the business process, whereas the post-condition expression may only refer to output parameters as it describes the effects upon the completion of the business process.

Figure 3.48 shows, through the running example, how the notion of pre- and post-condition are put in practice. The client is the patient that arrives at the diagnosis unit with the intention of being diagnosed and getting a treatment to deal with the reported disease. This, as already explained in Section 3.4.1.5, is the event (modelled in line 1) that triggers the business process. Since the *diagnosis* business process has been engineered to satisfy the client's demands, its goal (or post-condition) then is to diagnose the patient (attribute d of the output parameter ps) and provide him with a treatment (attribute p of the output parameter ps). This post-condition then (modelled in OCL as shown in line 21) specifies that the attributes d and p should be defined to consider the process instance achieving the business process goal. It is worth mentioning that the diagnosis unit personnel assume that a person would only come to the diagnosis unit because he believes he is ill. Hence, there is not any special pre-condition to be fulfilled by the person to be diagnosed at the unit. That is the reason why the business process pre-condition is always true (line 2).

A business process instance is assumed to meet its post-condition if the (1) activities are executed according to the order prescribed in the process definition and (2) each activity performs in a correct manner. The order in which the activities of a business process must be executed is modelled using the control flow operators introduced in Section 3.4.1.2. Having specified the activities' execution order, then it is possible to judge whether a process instance has adhered or not to the allowed execution paths prescribed in the process definition. Conversely, there is not a means to model what a particular activity has to perform. Since this is not modelled, then it cannot be judged whether the activity has been properly performed by a particular process instance.

This gap is bridged by using the notion of pre- and post-condition. Thus, a post-condition is used to describe what are the expected results of having executed the activity. The pre-condition describes what is assumed by the activity in order to achieve the expected results. An activity for which its pre-condition does not hold cannot be executed. An activity that ends its execution without holding its associated post-condition is considered as one that has not been performed properly (i.e. it has failed). A failing activity produces a deviation in the process instance

```
1  business process diagnosis(out PatientSheet ps) when(patientArrives) last[_,2hs.]
2    pre[true]{
3
4      participant DiagnosisUnit {
5        ...
6        Person prs;
7        composite registration[_,p = alloc(new Patient)](out prs) within[_,15min.];
8
9        Temperature t expire(1hs.);
10       BloodPressure bp expire(1hs.);
11
12       do{split composite examination[p,_,_](out t,bp),
13                 composite makeDocument[_](in prs; out ps);
14       }within[_,30min.];
15
16       composite consultation[p,_](in ps,t,bp; out ps) within[15min.,1hs.];
17       ...
18       composite giveInformation[p,_](in ps) within[_,15min.];
19       ...
20     }
21 } post[not ps.d.oclIsUndefined() and not ps.p.oclIsUndefined()]
```

Fig. 3.48: Pre- and post-conditions for the Diagnosis Business Process.

execution that leads it to miss its goal if no corrective actions are taken. Therefore, a deviation in the process instance execution can be detected when a post-condition associated with one of its prescribed activities evaluates to false. Associating a post-condition to an activity not only eases its judgement about correctness, but also provides a means to detect a deviation in the process instance execution. The context for pre- and post-conditions are the input and output parameter objects of the activity, respectively. Notice that *composite* and *nested* activities do not need to have pre- and post-conditions as these kinds of activities are actually references to business processes, which have their own pre- and post-conditions. This means that the pre- and post-condition of a *composite* or *nested* activity are held by the business process to which they refer. In this manner, pre-conditions and post-conditions must be provided for *atomic* activities, only. This is formalised by the part of the meta-model shown in Figure 3.49. The composite relationships *pre* and *post* model the fact that an *atomic* activity (modelled by the class *Atomic*) has a pre- and a post-condition, respectively. Conversely, *nested* and *composite* activities do not have a pre- nor a post-condition.

Once again, the running example is used to show how pre- and post-conditions at the level of *atomic* activities are put into practice. Figure 3.50 shows part of the *consultation* business process definition. This business process specifies that the *atomic* activities *diagnosePatient, prescribeTreatment* and *fillPatientSheet* need to be performed in sequence by the participant *Doctor*. The atomic activity *diagnosePatient* is considered as properly executed when a diagnosis (here modelled by the local object *d*) is obtained. Obtaining a diagnosis means that the local object contains a valid value as defined by its datatype. The OCL condition described in line 12 specifies this condition. Similar OCL conditions are specified for the *atomic* activities *prescribeTreatment* (line 16) and *fillPatientSheet* (line 20).

Recall that, in a business process, pre- and post-conditions are being used as a means to assess the execution of a particular process instance under the assumption that a certain initial condition holds. This means that if this initial condition (aka *pre-conditions*) holds, then the process instance is allowed to be executed. However, the actual execution of the process instance also depends on the existence and availability of the resources the instance requires to be performed. This underlines the fact that, from the perspective of a state transition system, the state of the

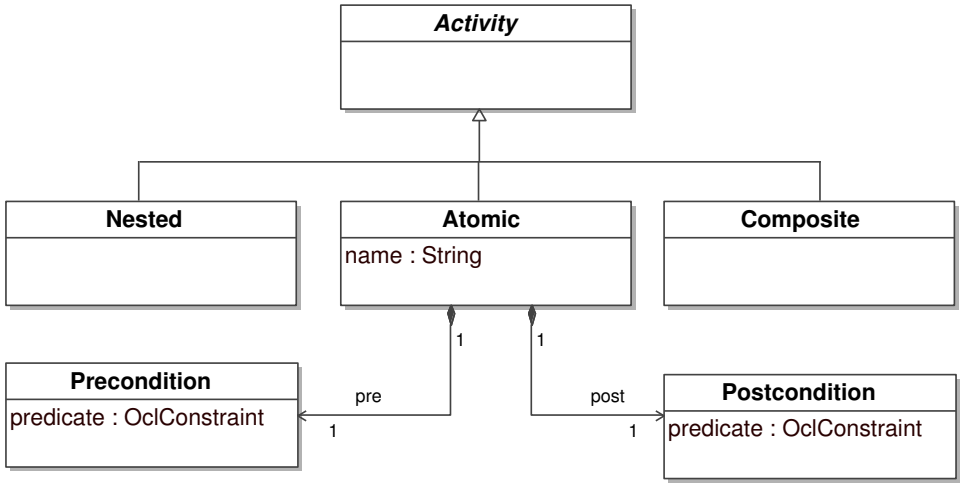

Fig. 3.49: Associating pre- and post-conditions with an activity.

mechanism used to execute a particular process instance is not only defined by the information held in the pre- or post-conditions, but also for the status of the resources that allow it to be played. Moreover, the state must also hold information regarding the life cycle of the activities such that they are executed in the specified order.

Pre- and post-conditions on *atomic* activities and business processes then provide clarity regarding the circumstances in which a particular business process is expected to reach its goal, and what conditions must be met to achieve the goal. It can, thus, be concluded that their inclusion in a process definition is a way to increase its dependability since either a deviation or failure of the process can be assessed without ambiguities.

3.4.4.2 Transactional behaviour

A *business transaction* [Pv07] is defined as a consistent change in the state of the business process that is driven by a well-defined activity. Business processes usually are composed of several business transactions. Business transactions exhibit the following characteristics:

1. they provided a result that is critical to the overall success of the business process in which they are embedded,

2. they involve two or more participants, which interact in a coordinated manner to achieve a mutually desired outcome

3. they run over long periods of time, behaving as open nested transactions [GR92] (i.e. sub-business transactions can abort or commit independently of the status of the final outcome of the parent business transaction),

4. they provide consistency and durability as ACID-transactions, but relax isolation (limited to those parts of the business process' state that are local to the business transaction) and atomicity (limited to guarantee consistency during the business transaction progress).

```
1  business process consultation(
2              in PatientSheet ps, Temperature t, BloodPreasure bp;
3              out PatientSheet ps) last[15min.,1hs.]{
4
5    participant Patient{
6          ...
7    }
8    participant Doctor{
9          ...
10     Diagnosis d;
11     diagnosePatient(out d)
12         post[not d.oclIsUndefined()];
13
14     Prescription p;
15     prescribeTreatment(out p)
16         post[not p.oclIsUndefined()];
17
18     ...
19     fillPatientSheet(in ps,p,d; out ps)
20         post[ps.d = d and ps.p = p]
21   }
22 }post[not ps.d.oclIsUndefined() and not ps.p.oclIsUndefined()]
```

Fig. 3.50: Pre- and post-conditions on *Atomic* activities within the *Consultation* Business Process .

Business processes belonging to the domain of interest (i.e. DCTC) share the following properties: they include multiple collaborative participants; they are expected to last for hours, days or longer; and, the special interest in making them dependable, is evidence of the importance in providing the expected business result. Furthermore, as explained in Section 3.4.2 when the notion of *nested* business process was introduced, (reduced) isolation and consistency are transactional properties that any business process should hold.

It can be concluded then that the assumption of considering a business process as a business transaction does not over-constrain the domain of interest. The motivation to consider business processes as business transactions (or transactions, for short) resides in exploiting the features provided by the transaction paradigm to increase the dependability of business processes. In other words, the nice features (i.e. ACID properties) that make transactions an approach to achieve fault-tolerant software can be borrowed and adapted to the context and needs driven in this thesis. Next, the tuned transactional properties that a DCTC business process provides are detailed.

- **Outcomes**
  There are four kind of outcomes a business process may produce: ***normal, degraded, aborted*** and ***failed***. The meta-model depicted in Figure 3.51 formalises these outcomes by means of the classes with the same names. The classes are specialisations of the class *Outcome*.

  A *Normal* outcome is produced when the business process reaches its goal, i.e. its associated post-condition holds. When the business process' goal cannot be reached, a dependable business process should strive to provide a partial service that potentially satisfies every participant [MKB08]. This partial service represents a *Degraded* outcome with respect to the initially agreed upon business objective. In the same manner as used for the normal outcome, the degraded outcome must specify the condition (modelled by the attributed *predicate*) that makes the business process outcome to be considered as degraded with respect to the original expected outcome.

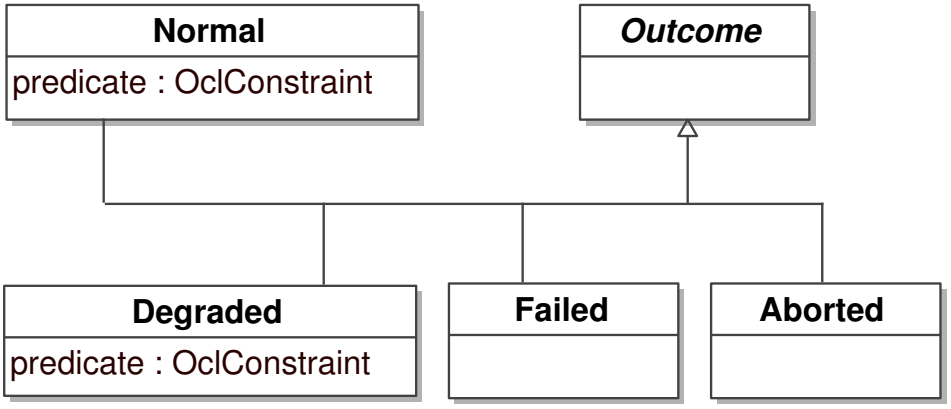

Fig. 3.51: Business process outcomes.

In the running example, a degraded outcome arises when the doctor requests that the patient remains in the hospital for some time because the patient's current health status does not allow the doctor to determine a clear diagnostic. The doctor then prescribes a preliminary treatment to be followed by the patient during his stay in the hospital such that he can eventually be diagnosed. Figure 3.52[47], (line 5) shows how the *degraded* outcome is modelled. Notice that what makes it degraded with respected to the expected *normal* outcome (Figure 3.52, line 3) 1) the patient has to remain at the hospital longer than expected and 2) the business process ends without providing a diagnostic to the patient, i.e. *ps.d.oclIsUndefined()*. Nonetheless, the patient does receive a preliminary treatment (i.e. ***not*** *ps.p.t.oclIsUndefined()*).

```
1 business process diagnosis(out PatientSheet ps) when(patientArrives) last[_,2hs.]{
2         ...
3 } post[not ps.d.oclIsUndefined() and not ps.p.oclIsUndefined()]
4 ...
5 }degraded[ps.d.oclIsUndefined() and not ps.p.t.oclIsUndefined()]
```

Fig. 3.52: Degraded outcome in the *Diagnosis* Business Process .

In the case that it is not possible to provide any partial service (i.e. *Degraded* outcome), the business process should be *Aborted*. Aborting a business process might imply performing business-specific activities to clean-up the effects of having executed non-reversible activities. Thus, an *abort* outcome specifies that the effects of executing a business process are undone, i.e. the pre-condition of the business process holds, regardless of the side effects remaining at the end of the business process.

All objects are considered as manually recoverable, which implies that specific knowledge has to be given for objects to recover their status when aborting the business process. Objects for which there is no such information, are considered as automatically recoverable. This means that "somehow"[48] they are rolled-back to the status they had before starting the business process.

[47] The reader is encouraged to review Section C.1 of the Appendix C to see the complete example.
[48] The recovery may be carried out for some underlying transactional system on which the business process relies on.

A *Failed* outcome occurs when the business process faced with problems that made it impossible not only to achieve its expected outcome or a degraded one, but also to be aborted.

A business process then relaxes the *all-or-nothing* outcome, i.e. the *atomicity* property, that (ordinary) transaction processing offers. This is because it is allowed to obtain partial results from its execution. It is worth noting that while *normal* and *failed* outcomes represent the best and worst cases that a business process can produce, respectively, *degraded* outcomes can vary from very close to the optimal expected result, to very far off. An *aborted* outcome can be considered as an extremely degraded outcome. However, the fact that it is the only outcome that is guaranteed to satisfy the business process pre-condition, makes it different.

- **Consistency**
  A business process is guaranteed to produce a consistent result if it satisfies any of the following conditions: it satisfies the business process' post-condition, a *normal* outcome; it satisfies one of the degraded post-conditions, a *degraded* outcome; or it satisfies its associated pre-condition when it aborts, an *aborted* outcome. The only situation where the business process does not offer a guarantee is when the business process fails entirely, a*failed* outcome.

  It is important to emphasise here that while a process definition always includes a *normal* outcome, it may or may not include *degraded, aborted or failed* outcomes. Including *degraded, aborted* or *failed* outcomes in the business process definition, implies extending the results it provides to its enclosing context.

- **Isolation**
  Due to the long-lived nature of a business process, only local information (i.e. local objects) can be isolated from the outside. This means that any change made over a non-local object would be seen from the outside. It is important to emphasise here that changes on both local and parameter objects are made by *atomic* activities only, since *composite* and *nested* activities are just references to business processes.

- **Locking**
  An *object*, regardless of whether it is a *parameter* or a *local* object, can only be accessed, i.e. read or written, within an *atomic* activity. As different *atomic* activities might attempt to access the same *object* concurrently, a locking mechanism is required to avoid smuggling information. The locking mechanism proposed is one that works at the level of *atomic* activities by locking those objects that are being passed as output arguments (as these variables are the ones being modified, only). These objects remain locked for the time the atomic activity performs, which is expected to be much shorter than the time required by a composite activity (i.e. long-running business process). An *atomic* activity needs to gain access over all objects being passed as output arguments before starting its execution, otherwise it remains waiting, i.e. it is blocked, until such a condition is met.

3.4.4.3 Exception handling

It is assumed that a process instance that has deviated from its prescribed paths, i.e. the process definition, eventually misses its goal if no corrective activities are executed. Thus, to avoid this ultimate failure the process must first *detect* that it has deviated from the prescribed path defined in the process definition. Second, the process must perform the necessary corrective activities that will *recover* return the execution to one of its prescribed paths.

Since detection and recovery are the bases of *fault tolerance* (see Section 2.2), techniques involved in fault tolerance should be considered as a means to model dependable business process. One of the techniques that allows fault tolerance to be attained, and has strong and direct connection with the features[49] that lead to the business process failure, is exception handling. Therefore, in order to attain dependable business processes, the concepts of *exception* and *handler* embodied by such a technique are borrowed and adapted in the following way: an *exception* denotes the notification of a deviation; a *handler* represents the set of corrective activities required to recover from such a deviation.

- **Deviations and Exceptions**
  It has been said that a deviation during the execution of a process instance can be detected when the post-condition associated with one of its prescribed activities evaluates to false. Since an exception is used as a means to signal the occurrence of a deviation, that is precisely what has to happen when a post-condition evaluates to false.

  An activity (either *composite, nested* or *atomic*) that ends its execution without holding its associated post-condition[50] represents a failed activity. While it is simple to know when an activity fails (i.e. the post-condition is false), it may not be obvious to realise what was the cause of the failure. There may be multiple reasons that the post-condition becomes false. Knowing the details of the activity failure helps to define the recovery measures that need to be put in place in order to mitigate the negative effects of these situations.

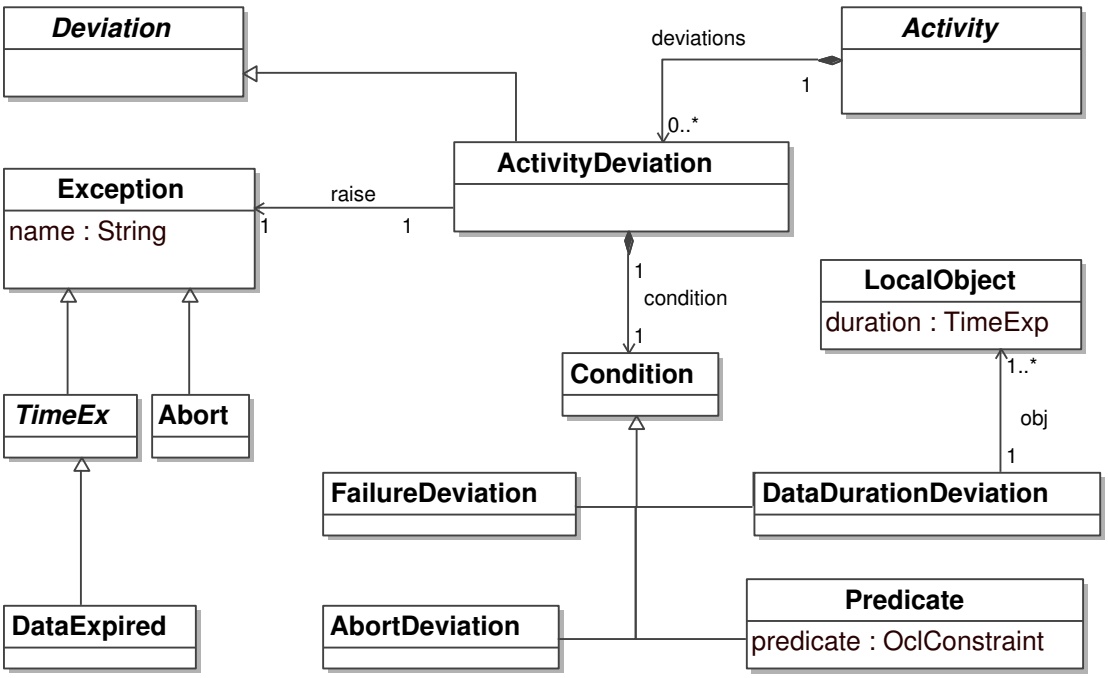

Fig. 3.53: Using deviation as means to detect activity failures.

The proposal to cope with this issue is that each activity has information about every expected deviation that may occur during its execution. In this manner, while the activity

[49] These features are: (1) detection of the deviation during the business process execution, and (2) handling the deviation after the deviation has been detected.

[50] For *composite* and *nested* the post-condition is the one held by the business process they refer to.

post-condition describes the expected result to be achieved when it performs successfully, an expected *deviation* describes the condition that allows detecting the failure of such an activity. An expected deviation then is modelled by providing (1) the condition that allows it to be detected and (2) the *exception* being raised in order to notify the actual occurrence of the deviation.

The meta-model shown in Figure 3.53 shows the formalisation of these concepts. The composite association *deviations* captures the different ways in which an activity (of type *Activity*) may deviate. Each of these deviations, *ActivityDeviation*, must have a condition of type *Condition*. The deviation is affiliated with the exception through the association named *raise*.

The keyword **deviation** is the element that lets modellers specify deviations in a process definition. The syntax for using this keyword is:

deviation*[exName|deviationCondition]*),

where *deviationCondition* is the condition (written using the OCL language) that allows the deviation to be detected. *exName* is the exception raised as soon as the condition exists. This means that a deviation might be detected not only upon completion of the activity, but also during its execution.

The conditions that make a deviation detectable may be very specific. These kinds of conditions are described as a *predicate* written in OCL. The class *Predicate*, which is a specialisation of the class *Condition*, along with the attribute of the same name captures this concept within the meta-model. It is worth noting that there may exist multiple deviations for the same activity such that their condition is of type *Predicate*. In that case, the conditions owned by each deviation have to be mutually exclusive with each other and with respect to the post-condition of the activity.

In the running example, deviations that may arise during the examination of the patient are shown in Figure 3.54. At this step in the process, a nurse and an assistant check the temperature (line 6) and blood pressure of the patient (line 14), respectively. With regard to the activity of checking the temperature, a value over 40 (line 8) or a problem in the thermometer (line 9) represent deviations with respect to the expected result of this activity (line 7). Regarding the assistant activity of checking the patient blood pressure, deviations with respect to the expected result (line 15) arise when the value of the blood pressure is over 200 (line 16) or the blood pressure monitor does not work properly (line 17).

A deviation that occurs because either (1) at least one of the activity's input parameters has expired, i.e. its associated duration time has passed, (2) the activity has been aborted, or (3) the activity has failed deserves special attention. These particular deviations are modelled using special conditions and exceptions.

The notion of the *DataDurationDeviation* is used to detect a late access to at least one of the input parameter objects of the activity that has its data value duration constrained by time (see Section 3.4.3 for a review of the time constraints over data). This condition must go along with the exception *DataExpired*, which is the means for signaling the occurrence of this type of deviation. It must be noted that since an object, regardless of whether it is a *parameter* or a *local* object, may only be accessed within an *atomic* activity (see 3.4.4.2 for more information), this kind of deviation may only take place during the execution of this kind of activity. The OCL invariants that ensure that both a *DataDurationDeviation* is associated only with an *AtomicActivity* (invariant *DataDurationDev*). The exception

```
1  business process examination(out Temperature temp, BloodPressure bp)
2                                  last[_,30min.]{
3
4    participant Patient{...}
5    participant Nurse{
6          ...
7          checkTemp()
8             post[temp|temp.oclIsNew() and temp < 40];
9             deviation[EX_HighTemp|(temp|temp.oclIsNew() and temp > 40)]
10            deviation[EX_MalfunctionThermometer|temp.oclIsUndefined()]
11         ...
12   }
13   participant Assistant{
14         ...
15         checkBP()
16            post[bp|bp.oclIsNew() and bp < 200];
17            deviation[EX_HighBP|(bp|bp.oclIsNew() and bp > 200)]
18            deviation[EX_MalfunctionBPMonitor|bp.oclIsUndefined()]
19         ...
20   }
21 }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
```

Fig. 3.54: Deviations that may arise in the *Examination* Business Process.

raised when this occurs is *DataExpired* (invariant *DataExpiredException*) is shown in Figure 3.55. Notice on the same Figure that the OCL invariant *DurationGTZero* ensures that the duration of each object being referred to by *DataDurationDeviation* is greater than zero, otherwise the deviation would never take place.

```
context Activity inv DataDurationDev:
        Activity.allInstances()->forAll(act|
          act.deviations->exists(dev|dev.oclIsTypeOf(DataDurationDeviation)
                implies act.oclIsTypeOf(Atomic)))

context ActivityDeviation inv DataExpiredException:
        if(self.condition.oclIsTypeOf(DataDurationDeviation)) then
                self.raise.oclIsTypeOf(DataExpired)
        else true endif

context DataDurationDeviation inv DurationGTZero:
        self.obj->forAll(o|o.duration > 0)
```

Fig. 3.55: Invariants related with the *DataDurationDeviation* concept.

The concrete modelling of a *DataDurationDeviation* is achieved using the same primitive **deviation** with the special condition **dataExpired(**$obj_1, ..., obj_n$**)**[51], and an exception of type *DataExpired*.

Regarding a deviation that occurs because the activity has been aborted or failed, the modelling distinguishes between *atomic*, *composite* and *nested* activities. Both the abortion or failing of an *atomic* activity is modelled by a deviation (*ActivityDeviation*) that has a condition of type *Predicate*. This means that a condition (written using the OCL language) and an exception has to be provided. However, it must be noted that an activity is considered as having deviated due to an abortion only if the exception used to notify its occurrence is of type *Abort*.

[51] The objects $obj_1, ..., obj_n$ are the *atomic* activity input parameters that have their data duration constrained by time.

The case is different for *composite* or *nested* activities, as they must capture the different deviations the business process may produce. For the sake or simplicity, the analysis here is performed only for *composite* activities, however it is also valid for *nested* activities. As explained in Section 3.4.4.2, a business process produces four different kinds of outcomes: *normal, degraded, aborted* and *failed*. A *composite* activity captures the *degraded* outcome (i.e. ***degraded****[cond]*) by means of a deviation *dev* that has a condition of type *Predicate* (i.e. *dev.condition.isTypeOf*(*Predicate*)) such that the condition of the deviation is the same as the degraded outcome *dev.condition. predicate* = *cond*. However special conditions are required to capture the outcomes *aborted* and *failed*[52]. The particular concepts *AbortDeviation* and *FailureDeviation* are introduced to achieve their modelling. In this manner, a *composite* activity relies on the notion of *AbortDeviation* to model the condition that captures the *aborted* outcome of the referred business process, whereas *FailureDeviations* is the condition that captures the *failed* outcome. The OCL invariants that ensure that the special conditions *Abortdeviation* and *FailureDeviation* are only used in deviations associated with *composite* or *nested* activities are shown in Figure 3.56.

```
context Activity inv AbortDev:
        Activity.allInstances()->forAll(act|
          act.deviations->exists(dev|dev.oclIsTypeOf(AbortDeviation)
            implies (act.oclIsTypeOf(Composite) or (act.oclIsTypeOf(Nested)) )))

context Activity inv FailureDev:
        Activity.allInstances()->forAll(act|
          act.deviations->exists(dev|dev.oclIsTypeOf(FailureDeviation)
            implies (act.oclIsTypeOf(Composite) or (act.oclIsTypeOf(Nested)) )))
```

Fig. 3.56: Invariants related with the *AbortDeviation* and *FailureDeviation* concepts.

The keywords **aborted** and **failed** let the modeller capture the deviation conditions of the *composite* activity that represent the abortion and failure of the business process, respectively.

An activity may also fail when one of its time constraints is not met. The time constraints that can be associated with an activity are those determined by the concepts *In* and *Within*. The concept *In* is used to constrain the start of the activity, whereas *Within* determines the duration of the activity (see Section 3.4.3.3 for more details about these time-related concepts).

Before going further with the explanation of missing time constraints there is a point with regard to the concept *In* that needs to be clarified. The notion of *In* can be used to constrain the start time of an activity. This means that the activity has to be delayed for a certain time before starting its execution. It is assumed that the delay set by using the concept *In* is always met. Therefore, such time constraints cannot be missed. Hence, the time constraints that can be missed regarding an activity are those modelled with the concept *Within*.

The fact that one of the time constraints modelled with the concept *Within* is not met represents a deviation in the execution of the activity. Time-related deviations are formalised in terms of metamodelling as instances of the class *TimeDeviation*, which extends from the abstract class *Deviation*. The composite relationship named *timeout* is aimed at modelling the time-related deviations that may be associated with the time constraints

[52] More details about the relationship between the *composite*'s outcome and the business process' outcomes is given at the end of this Section.

set by means of the concept *Within*. Figure 3.57 shows the part of the meta-model that formalises these concepts.

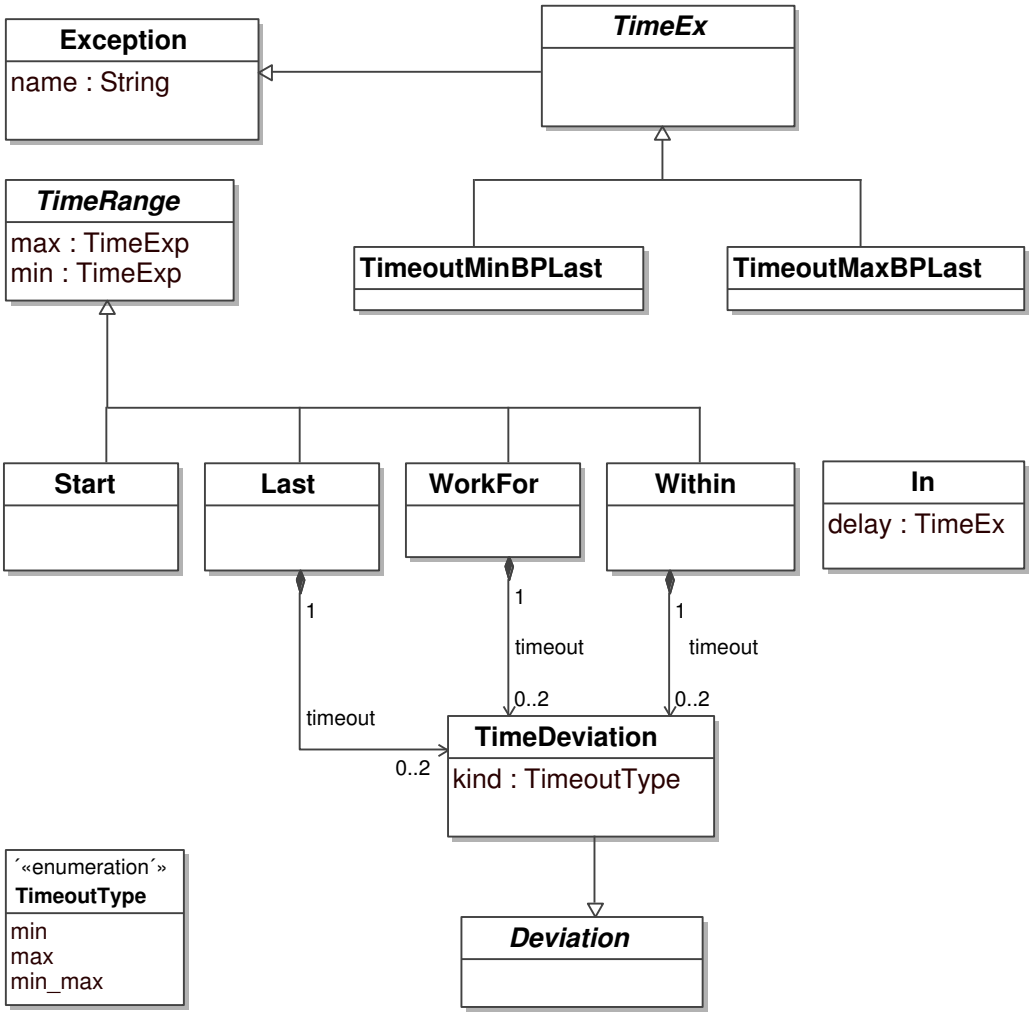

Fig. 3.57: Time-related deviations as a means to detect timeouts.

As time-related deviations are a specialisation of a *Deviation*, it must be used to explicitly describe (1) the condition that detects a missed time constraint, and (2) the exception that is then raised to signal its occurrence.

The missed time constraint may be due to either an early or late completion of the activity with respect to the imposed time limit. An early finish of an activity is considered as having the lower-bound missing, whereas late completion of the activity as having the upper-bound missing. This points out that time-related deviations can be categorised with respect to the way in which the time constraint is missed. Notice that knowledge of whether it is the upper or the lower bound constraint that has been missed may be of interest, in particular when the handling associated with each deviation is not the same[53]. The class *TimeDeviation* has the attribute *kind*, which is of type *TimeoutType*. This attribute is aimed at capturing the condition, i.e. point (1), that allows the deviation to be detected. Hence, a time-related deviation for which this attribute has the value *min*

[53] More information about the handling of a deviation is given in the next section.

models the deviation of the activity when its lower bound is missed, whereas the value *max* when its upper bound is missed. The case in which the attribute has the value *min_max* models the deviation of the activity regardless of whether it is the lower or upper bound that is missed.

Regarding the concrete modelling of the conditions associated with time-deviation, special predicates, which are not OCL expressions are used. The special predicate ***timeout.min*** denotes that a lower-bound was missed whereas ***timeout.max*** indicates that an upper-bound was missed. The special predicate ***timeout*** denotes that a time constraint was missed without making any distinction as to whether the missed bound was the upper or the lower one.

Regarding the signaling of a time-related deviation, i.e. point (2), an exception named *timeout* is assumed to be raised. Hence, whenever a time-related deviation occurs (whatever the time-related deviation is), the *timeout* exception is raised. Time-related deviations, thus, do not require the modelling of an exception to signal its occurrence.

The same principle of using a time-related deviation to denote that a time constraint has been missed is extended to those time-related constraints that might be set over a participant and/or a business process. However, there are certain points regarding the exception of a deviation that merit further discussion.

As explained in Section 3.4.3.2, the notion of *WorkFor* can be used to place a time-constraint over the duration of the actual participant, i.e. the *resource*, performing its enclosing activities. A time-related deviation associated with the concept *WorkFor* then represents that the maximum or minimum elapsed time constraints have been missed. The occurrence of any of these deviations produces the implicit raising of a *timeout* exception, in the same way as it is done when a time-constraint associated with *Within* is missed.

Regarding the business process, timing constraints can be set using the concepts *Start* and *Last* (see Section 3.4.3.1 for more details). Timing constraints set with the concept *Start* deserve special attention. First, the lower time bound, i.e. minimum delay, set over a business process using the concept *Start* is assumed as always being met. Second, the missing of the upper time bound defined by using the *Start* concept, represents the case in which the process instance has not been able to start its execution on time. Since no execution has happened, the behaviour of the process instance is equivalent to the one that has aborted its execution. Therefore, the enclosing context of a process instance that misses its starting time gets signaled by a deviation of type *AbortDeviation*. In this manner, time-related deviations can be associated with the concept *Last*, only.

Time-related deviations are associated with the concept *Last* in the same manner as considered so far. When one of the deviations associated with this concept occurs, an implicit exception is also raised. Conversely to deviations associated with *Within* and *WorkFor*, the implicit exception raised when a deviation associated with *Last* occurs is dependent of the time-constraint missed. This is due to the fact that the administration of such deviations requires the involvement of every business process participant. Further, (as the reader will see in the uncoming sections) the modelling of this of kind of handling needs precise information regarding the exception to be handled. In this manner, the notion of *TimeoutMinBPLast* is introduced to signal that the lower time bound associated with the concept *Last* has been missed. Whereas *TimeoutMaxBPLast* is used to indicate that the upper time bound has been missed.

- **Handling model**
  When an exception is raised, indicating that a deviation has been detected, corrective

actions have to be undertaken to avoid that the business process misses its goal. The set of corrective actions defines a *handler*. A handler is meant to deal with one or more exceptions. The mechanism that determines how to find a handler for a particular exception is known as a *propagation mechanism*. The propagation mechanism adopted in this work adheres to the termination model [BM00]: once the handler completes, the control flow is expected to continue as if the exception would not have occurred. However, the continuation of the business process control flow will depend on the severity of the exception to be administered. When the severity of the exception makes continuation of the execution of the business process as originally planned impossible, its handling has to be designed to terminate the business process execution in a safe way, i.e. the negative effects of missing the goal should be mitigated as much as possible.

A raised exception can be handled either at the level where it has occurred, or at any other higher level that can be reached as a result of the propagation of the exception. During the execution of a business process, an exception is initially raised when an activity deviation is detected. Unless handled, this raised exception is propagated up to the level of the business process. Thus, an activity deviation defines the lowest of hierarchical levels that an exception can go through once it is raised. The business process is the highest level. A handler meant to deal with an exception at the level of the activity deviation is termed *participant handler*. A hander that deals with an exception at the level of the business process is termed *business process handler*. In the following, both kinds of handlers are explained:

1. **Participant handler**
   The scope of an exception being raised due to the deviation of an activity is the participant that encloses such activity. Hence, once the exception is raised, the propagation mechanism searches for a handler at the level of the participant.

   The aim of a *participant handler*, simplified to *p-handler* from here on out, is to deal with the exception that indicates the detection of a deviation. The existence of a p-handler is determined by the activity to which it is associated in order to deal one of its possible deviations. This existential relationship is formalised by means of metamodelling using the class *PHandler* and the composite relationships named *localHandlers* and *handlers* (see Figure 3.58 for the description of the meta-model that shows this formalisation). The class *PHandler* captures the modelling of the different p-handlers that can be created, whereas the composite relationships bind the p-handlers with the activities they belong to. The composite relationship *localHandlers* contains those p-handlers aimed at dealing with logic-related deviations, i.e. deviations of type *ActivityDeviation*, whereas *handlers* contain p-handlers aimed at dealing with time-related deviations, i.e. deviation of type *TimeDeviation*.

   Thus, whenever a p-handler exists, it must be associated with one deviation. The relationship between a p-handler and a deviation is formalised by the association named *handles*. Notice that a same p-handler may be used to deal with more than one deviation. This fact makes the *handles* association have a cardinality $1..*$.

   Notice that each of the corrective activities enclosed within a p-handler, captured as the statements owned by the p-handler - ordered composite relationship named *stmts*, may have associated deviations and/or time-constraints. Thus, attaching p-handlers to these activities in order to deal with deviations is also permitted. Therefore, p-handler may contain nested p-handlers. Once a p-handler has completed its execution, the control flow of the participant continues as the activity would not have faced a deviation.

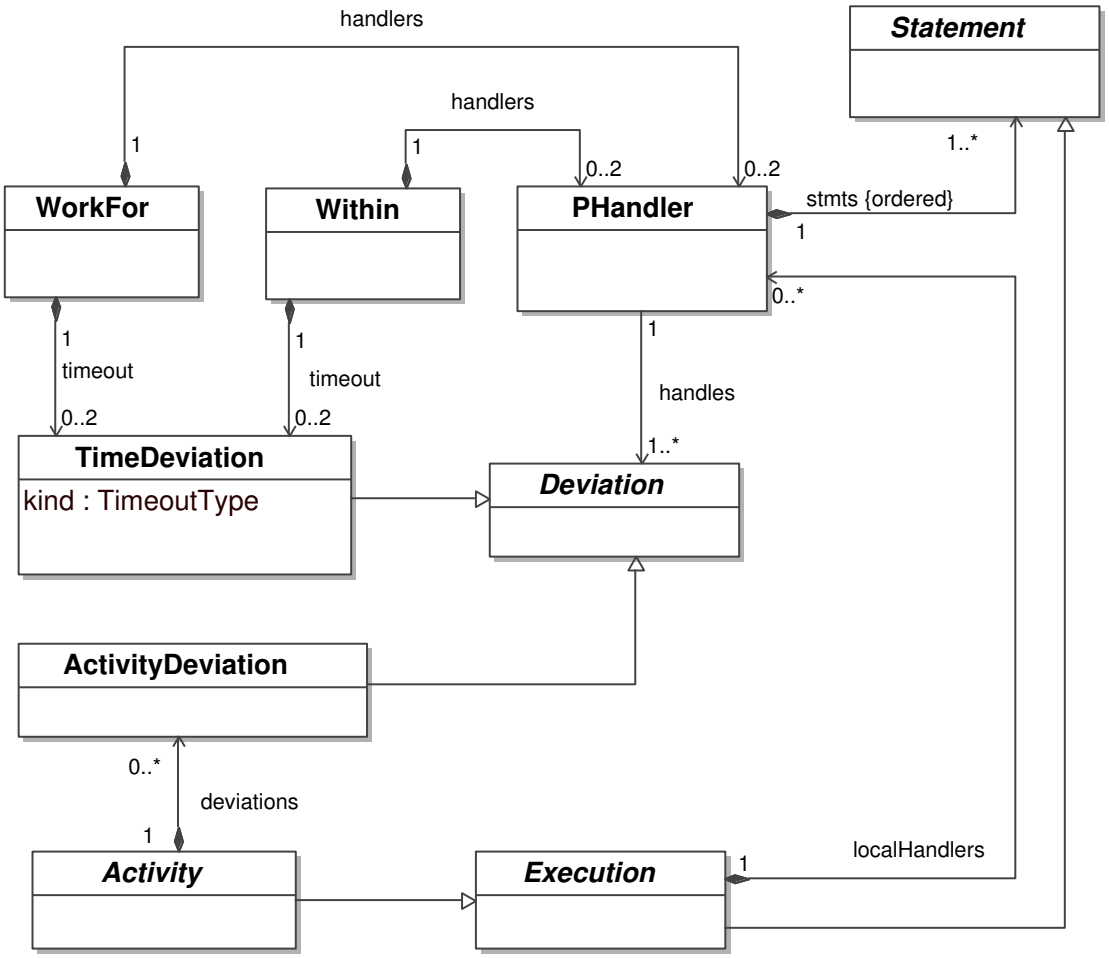

Fig. 3.58: Handling deviations within a *Participant*.

The concrete modelling of a p-handler is achieved by (1) enclosing the corrective activities it performs between the "**{**" and "**}**" brackets, and (2) placing them next to the exception they are meant to handle (i.e. ***deviation***$[ex \mid condition]\{act_{hnd_1}, ..., act_{hnd_n}\}$). Figure 3.59 shows the p-handlers defined within the *Examination* Business Process (lines 10-12 and 20-22) to deal with the deviations "malfunction of the thermometer" (line 10) and "malfunction of the blood pressure monitor" (line 20), respectively. In the case of a malfunction of the thermometer, the corrective action to be undertaken is to find a new thermometer (line 11). To find a new blood pressure monitor is the corrective action prescribed when a the malfunction is detected in that kind of device (line 21).

It has been stated that the same p-handler can be used to handle different deviations occurring within the same activity. The concrete modelling of this is achieved by using the disjunction operator ***or***, which allows for the joining of either (1) the different deviations, i.e. ***deviation***$[ex_1 \mid condition_1]$ ***or...or deviation***$[ex_i \mid condition_i]$ $\{act_{hnd_1}, ..., act_{hnd_n}\}$, or (2) the different conditions that produce each deviation, i.e. ***deviation***$[ex \mid condition_1$ ***or*** ... ***or*** $condition_i]$ $\{act_{hnd_1}, ..., act_{hnd_n}\}$.

Notice that a p-handler is considered as executing within the context of the activity it is associated with. Hence, a p-handler has access to the same input parameters of

```
1  business process examination(out Temperature temp, BloodPressure bp)
2                                                  take[_,30min.]{
3
4    participant Patient{...}
5    participant Nurse{
6           ...
7           checkTemp(out temp)
8              post[temp|temp.oclIsNew() and temp < 40];
9              ...
10             deviation[EX_MalfunctionThermometer|temp.oclIsUndefined()]{
11                        findNewThermometer();
12             }
13          ...
14   }
15   participant Assistant{
16          ...
17          checkBP(out bp)
18             post[bp|bp.oclIsNew() and bp < 200];
19             ...
20             deviation[EX_MalfunctionBPMonitor|bp.oclIsUndefined()]{
21                        findNewBPMonitor();
22                  }
23          ...
24   }
25 }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
```

Fig. 3.59: Deviations being handled locally by *p-handlers* in the *Examination* Business Process.

its associated activity.

For a deviation that does not hold an associated p-handler, the exception being raised when the deviation occurs is propagated to the p-handler's enclosing context. This context is either another p-handler, or the *participant* that owns the deviated activity. Once the exception has reached the context of the participant, it is propagated to the level of the business process. Eventually, the exception is expected to be handled. This means that, at a minimum, a handler at the level of the business process to deal with the deviation has to be defined.

2. **Business Process handler**
   As just stated, when there is not p-handler associated with an activity deviation, the exception is propagated to the level of the business process. Propagating the exception to the entire collaborative business process has the effect that (1) every participant stops executing its current activities and (2) the propagation mechanism starts searching for a handler at the level of the business process to deal with the exception.

   The aim of a *business process handler* (*bp-handler* from here on out for simplicity) is to administer an exception in a collaborative manner. A bp-handler involves all the participants enclosed within the same collaborative business process. Hence, each participant taking part in a particular business process, is also involved within a bp-handler. In this manner, a bp-handler is formalised by means of metamodelling (see Figure 3.60) as an instance of class *BPHandler*, such that (1) it is composed of a set of participants (modelled by the composite relationship *hndParticipants*), and (2) once it completes its execution it produces an outcome modelled by the composite relationship *outcome*.

   The activities performed by a participant when dealing with a particular deviation within the context of a bp-handler are those contained by the ordered composite relationship named *stmts*.

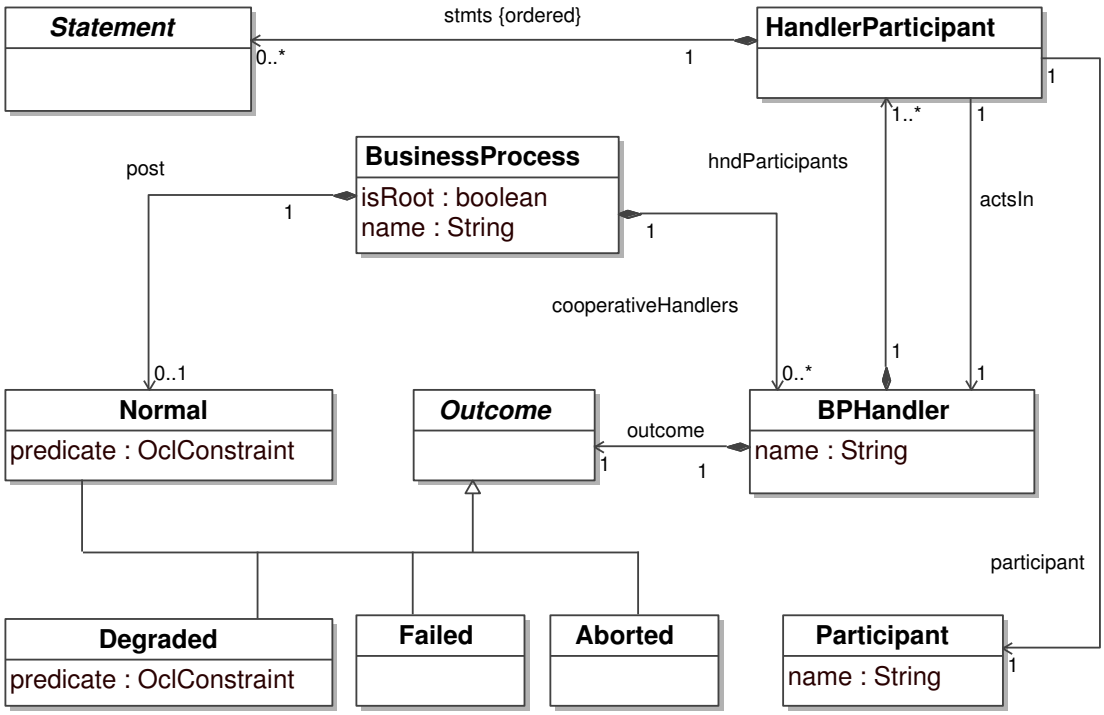

Fig. 3.60: Handling deviations in a collaborative way.

Note that there must exist one participant of type *HandlerParticipant* within a bp-handler for each participant of type *Participant* held by the *BusinessProcess* [54]. The association named *participant* between classes *HandlerParticipant* and *Participant*, along with the OCL condition shown in Figure 3.61 ensures this property. The association ensures that a handler participant is associated with one and only one participant, whereas the OCL condition checks that the number of participants within the bp-handler and the business process are the same.

```
context BusinessProcess inv OneHandlerParticipantForEachBPParticipant:
        self.cooperativeHandlers->
                forAll (bpHandler: BPHandler |
                        bpHandler.hndParticipants->size() = self.participants->size()
                )
```

Fig. 3.61: Consistency between the number of *HandlerParticipant*'s and the number of *Participant*'s.

The fact that a business process may own none or many bp-handlers is modelled by the composite relationship named *cooperativeHandlers*.

A bp-handler produces an outcome (of type *Outcome*). This outcome indicates that the business process has either (1) fully reached its goal. i.e. *normal* outcome, (2) partially reached its goal, i.e. *degraded* outcome, (3) aborted its execution, i.e. *aborted* outcome, or (4) failed in a graceful way, i.e. *failed* outcome. The kind of outcome to be provided depends in great part on the severity of the exception being handled.

[54] See Figure 3.10 for the relationship *BusinessProcess-Participant*.

However, it must be noticed that a bp-handler always produces an outcome as it is assumed to fully complete its execution.

When the severity of the deviation does not allow the business process goal to be reached either fully or partially, the business process, as a last alternative, should be aborted, i.e. *abort* outcome. By default, the abortion of a business process is achieved by simply raising the exception *Abort*. This points out that it is assumed the implicit execution of a *bp-handler* whenever a deviation notifies its occurrence by raising the *Abort* exception. However, it must be noticed that the modeller still has the chance to explicitly model the required activities that make a business process abort. This may be usually the case when dealing with long-live business processes in which objects are modified by non-reversible effects.

To avoid overloading the *Process Model* with information related to the collaborative handling of exceptions, a new model called *Dependability Model* is defined to include such information. Therefore, the concrete modelling of a bp-handler is defined within the *Dependability Model*, in the **recovery** section. Figure 3.62 shows part of the *Dependability Model* related to the *Examination* business process. The ***recovery*** section of this model includes the handlers *Undress* (lines 2-11) and *CardiacUnit* (lines 14-27). These handlers are meant to deal with the exceptions *EX_HighTemp* and *EX_HighBP*, which are used to notify the occurrence of the deviations "temperature over 40" and "blood pressure over 200", respectively (the raising of these exceptions can be found in Figure 3.54, on lines 9 and 17, respectively). Both bp-handlers are expected to deal with these exceptions such that the business process fully reaches its expected goal. Hence, the outcome produced for each of these bp-handlers is ***Normal*** (lines 11 and 27).

```
1  recovery{
2     Undress(out Temperature temp, BloodPressure bp) {
3          participant Patient{
4                  receive reqUndress from Nurse;
5                  Undress();
6          }
7          participant Nurse{
8                  send reqUndress to Patient block;
9          }
10         participant Assistant{}
11    }Normal
12
13
14    CardiacUnit(out Temperature temp, BloodPressure bp) {
15         participant Patient{
16                 receive reqChangeRoom from Assistant;
17                 goToSpecialUnit();
18         }
19         participant Nurse{
20                 receive reqChangeRoom from Assistant;
21                 takePatientToSpecialUnit();
22         }
23         participant Assistant{
24                 send reqChangeRoom to Patient,Nurse;
25                 notifyNewRoomPatient();
26         }
27    }Normal
28 }
```

Fig. 3.62: *Business Process handlers* in the *Dependability Model* of *Examination* Business Process .

A bp-handler executes within the context of the business process it is attached to. Hence, participants collaborating in a *bp-handler* not only have access to the business process input data (i.e. *parameter* objects), but also to the local data (i.e. *local* objects) produced up until the moment the exception was raised in the case of the deviated participant or received (or participants being notified about the occurrence of the deviation. This semantics allows every participant to use, in the handling of the exception, the work performed during its normal execution within the business process until the occurrence or notification of such exception.

Thus, a handler, whether it is a p-handler or a bp-handler, is the feature that allows a business process to produce a consistent outcome. i.e. *normal, degraded* or *abort* even when confronted with deviations. When the deviation is very severe and the failure of the business process is unavoidable, the last thing that can be done is try to reduce the harm as much as possible before notifying the process users of the failure. A dependable business process is assumed to include a handler for each defined deviation. This handler can be at the level of the participant, by providing a p-handler, or at the level of the overall business process by a bp-handler.

- **Parallel exceptions**
  Multiple exceptions may arise simultaneously. This is possible not only because participants execute their activities in parallel, but also because the same participant may execute multiple activities in parallel: the *split* and *spawn* control flow operators allow the parallel execution of activities by the same participant. Therefore, a participant that performs in parallel at least two *composite* activities[55] may face with this situation. Another source of multiple parallel exceptions takes place when an activity has a time-related constraint defined by the primitive *within*: in this case, the time-related constraint could be missed along with an activity deviation. When multiple parallel exceptions take place, whatever the case is, the *propagation mechanism* must determine what is the exception to be handled before it starts searching for an appropriate handler.

  The resolution mechanism proposed in this thesis to determine what is the chosen exception to be handled when multiple exceptions are raised in parallel is based on the approach described by Campbell and Randell in [CR86]. The principle followed by the authors is that all multiple exceptions being raised simultaneously, in conjunction, constitute a new exception that is symptomatic of a different and more complicated deviation than those reported by each exception individually. Therefore, when multiple exceptions are raised in parallel (i.e. $ex_1, ..ex_n$), the *propagation mechanism* will look for a handler (whether p-handler or bp-handler) with the capabilities to deal with the exception that the conjunction of all raised exceptions (i.e. $ex_1 \textbf{\textit{and}} ... ex_{n-1} \textbf{\textit{and}} ex_n$) defines.

  In large and complex collaborative business process, such as those being targeted, the number of potential exceptions that may arise in parallel grows exponentially. In line with the principle of explicit modelling dependability adopted in the context of this thesis, every potential combination of parallel exceptions that could be raised and that need to be handled, *must* be explicitly determined in the business process definition. Parallel exceptions that are not explicitly included in the business process definition are assumed (by the stakeholders) as impossible to occur[56]

[55] It is impossible to get raised multiple exceptions when a participant is prescribed to perform in parallel *atomic* activities only, since all these activities are to be performed by the same actual resource. In this scenario, the resource performs the parallel activities in a time-sharing fashion.

[56] Of course, the occurrence of any of these exceptions will make the business process fail.

Both the definition of a set of parallel exceptions and its binding with a handler can be done at the participant and business process level. Parallel exceptions at the level of the participant are modelled in a concrete manner using the conjunction operator ***and***. This operator allows several deviations to be combined into a joint deviation, i.e. ***deviation***$[ex_1 \mid condition_1]$***and...and deviation***$[ex_i \mid condition_i]$, such that this joint deviation is detected if and only if the conjunction of all the conditions holds. The handling of a set of parallel exceptions within the context of the participant is achieved by defining a *p-handler* next to the definition of such set of parallel exceptions. As already explained, parallel and ordinar) exceptions for which no *p-handler* has been defined are propagated to the level of the business process.

Parallel exceptions at the level of the business process along with the handler in charge of dealing with it are defined using the concept *resolution*. This concept is meant to map a set of parallel exceptions with a *bp-handler*. Since a single raised exception can be considered as a particular case of parallel exceptions, i.e. a set with only one element, the concept *resolution* is used as a means to statically define the binding between an exception and its associated handler, regardless of whether the exception is defined as the conjunction of parallel exceptions or it is just a simple exception. The formalisation of this concept is shown in Figure 3.63, where it is specified that a resolution, *Resolution*, must be created within a business process every time a particular set of exceptions modelled by the association of name *left* wants to be handled by a bp-handler, captured by the association of name *right*. Since a bp-handler is defined to deal with at least one exception, there should exist at least one resolution that refers to it. Figure 3.64 shows the OCL condition that ensures this property over *DT4BP* models.

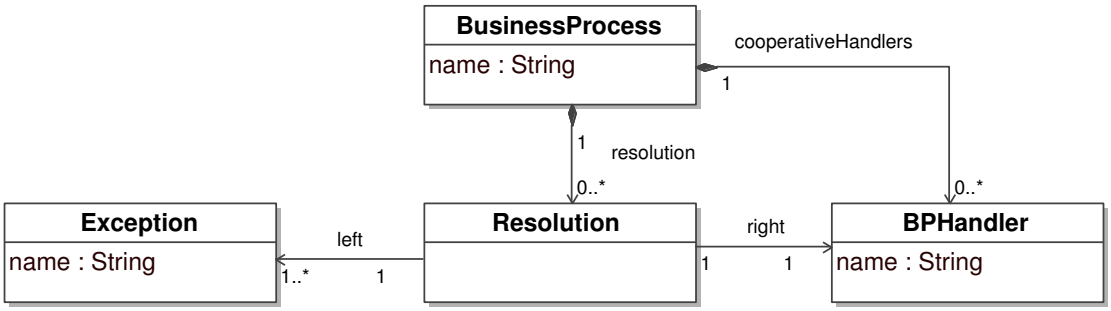

Fig. 3.63: Definition and handling of parallel exceptions.

```
context BusinessProcess inv BPHandlerInResolition:
      self.cooperativeHandlers->
        forAll (bpHandler: BPHandler |
                self.resolution -> exists(res:Resolution| res.right = bpHandler)
        )
```

Fig. 3.64: A *BPHandler* is referred to at least once for a *Resolution*.

The primitive **resolution** is the concrete element introduced to define a section within the *Dependability Model* where all the bindings between a set of exceptions and the *bp-handler* in charge of its handling are defined. An arrow $(->)$ is used to define the binding between a set of parallel exceptions (left part) and the respective *bp-handler* (right part). Figure

3.65 shows how these concrete primitives are used in the *Examination* business process of the running example.

A parallel exception that may arise during the examination of the patient is that both his temperature and blood pressure are over the expected values, i.e. over 40 and 200, respectively. A specific handler called *UndressANDCardiacUnit* is defined to deal with such parallel exception (lines 16 and 18). The binding between this parallel exception and its associated handler is shown on line 4. The bindings *EX_HighTemp-Undress* (line 2) and *EX_HighBP-CardiacUnit* (line 3), which are considered when each exception is raised alone, are also included in this section.

```
1  resolution{
2          EX_HighTemp -> Undress;
3          EX_HighBP -> CardiacUnit;
4          EX_HighTemp and EX_HighBP -> UndressANDCardiacUnit;
5  }
6
7  recovery{
8     Undress(out Temperature temp, BloodPressure bp) {
9      ...
10    }Normal
11
12    CardiacUnit(out Temperature temp, BloodPressure bp) {
13     ...
14    }Normal
15
16    UndressANDCardiacUnit(out Temperature temp, BloodPressure bp) {
17     ...
18    }Normal
19 }
```

Fig. 3.65: Binding single and parallel exceptions with their respective *bp-handlers* in the *Dependability Model* of *Examination* Business Process .

- **Binding between possible business process' outcomes and composite activity's results**
  Since a *composite* activity[57] is a container for a business process, there must exist a clear connection between the different results returned by the *composite* activity, i.e. normal and deviated, and the outcomes produced by the referring business process (i.e. normal, degraded, aborted and failed).

  Since a *composite* activity is considered as having executed successfully when the business process it refers to has correctly performed, the normal result achieved by a composite activity is determined by the correctness of the referred business process.

  A deviated *composite* activity is one whose execution is considered as improperly done. This means that the business process encapsulated by the *composite* activity has produced an outcome that is not *normal*. A *degraded* outcome produced by a business process, ***degraded****[condition]*, becomes a *deviation* at the level of the *composite* activity. Note that the condition described by the deviation at the level of the *composite* activity must be the one used to model the degraded outcome at the level of the business process.

  The *aborted* and *failed* outcomes become special kinds of deviations at the level of the *composite* activity. What make them special is that their names are used to describe the

[57] It is worth noticing that the following analysis is also valid for *nested* activities.

deviation condition since their implicit semantics are enough to understand the kind of outcome produced by the encapsulated business process.

Figure 3.66 relies on the BPMN-like notation to depict the relationship between the possible *composite* activity's results and the possible business process' outcomes in a graphic manner.

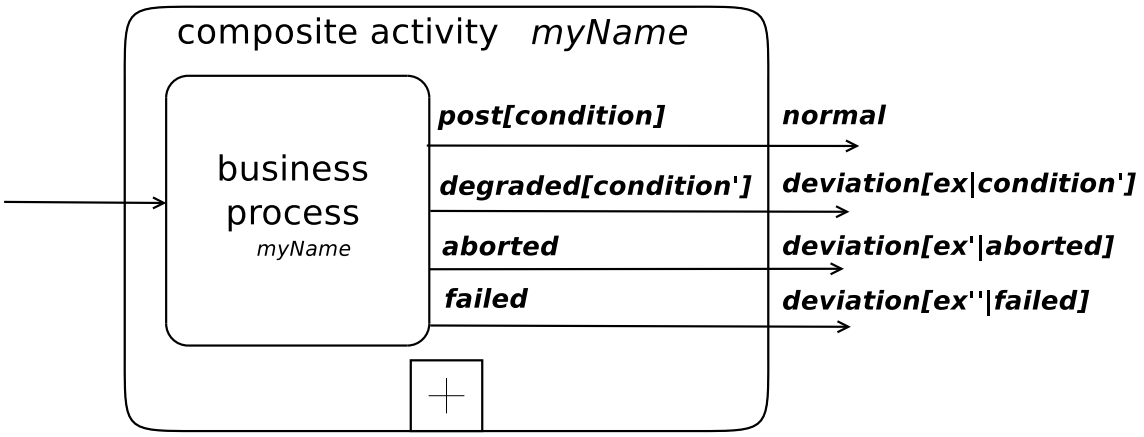

Fig. 3.66: Relationships between the outcomes of a business process and its view as composite activity.

3.5 Comparison with existing business process modelling languages

This section compares *DT4BP* with the existing business process modelling languages described in Section 3.2. The comparison criteria is based on those key concepts that govern the targeted business processes (i.e. collaboration, time and dependability) including the relevant factors that should be considered when modelling a business enterprise (i.e. function, organisation, information and resource [Toh99, LZRT08]). This comparison allows the reader to identify, in a compact manner, not only the modelling concepts that *DT4BP* supports, but to compare the current support of some existing business process languages with respect to these concepts.

The metric used to evaluate the support a particular business process modelling language offers for a certain concept is defined as follows: (+) indicates *direct support* for the concept, (+/−) indicates *partial support*, and (−) indicates that the concept is *not supported* at all. *Direct support* means that there exist constructs in the language that allow certain concepts to be captured effectively in accordance with its semantics. *Partial support* means that a modelling language is able to capture a concept by combining or extending (if possible) some of its constructs, despite the fact that these constructs are aimed at capturing different concepts. It is important to underline that a modelling language may provide partial support for a concept for which it was originally not targeted. The proposed workaround should correlate relatively closely with the concept being addressed. A modelling language that requires either complex workarounds to capture a particular concept or significant conceptual reorganisations of its constructs is considered as one that does not provide support to capture such concept.

Table 3.5.1 lists the results of applying the assessment criteria to both the existing business process modelling languages as well as to *DT4BP*. This table also displays the individual aspects

Concept	**UML-AD**	**BPMN**	**YAWL**	**EPC**	**DT4BP**
Modelling viewpoint					
Control-flow	+	+	+	+	+
Data	−	−	+	−	+
Resource	−	−	+	−	+
Exception handling	−	−	+	−	+
Collaboration					
Multi-participants	+/−	+	−	−	+
Choreography	+/−	+	−	−	+
Time					
Business process min. delay	+	+	−	−	+
Business process max. delay	−	−	−	−	+
Business process min. duration	−	−	−	−	+
Business process max. duration	−	+	−	−	+
Periodic business process	−	+	−	−	+
Participant working time	−	−	−	−	+
Activity min. delay	+	+	+/−[a]	−	+
Activity max. delay	−	−	−	−	+
Activity min. duration	−	−	−	−	+
Activity max. duration	−	+	+/−[b]	−	+
Object duration	−	−	−	−	+
Dependability					
Pre-condition	−	−	+	−	+
Post-condition	−	−	+	−	+
Transaction	−	+	−	−	+
Deviation	+	+	+	−	+
Deviation handling	+	+	+	−	+
Multiple parallel deviations	−	−	−	−	+
Cooperative handling	+/−	+/−	−	−	+

Tab. 3.5.1: Comparison between existing business process modelling languages and *DT4BP*.

[a]Only for Atomic Automatic Activities.
[b]Only for Atomic Manual Activities.

that determine each concept. Regarding the concept *modelling viewpoint*, the attributes that determine this concept are the different types of perspectives oriented toward the description of a business process. Russell [Rus07] identifies *control-flow, data, resource* and *exception handling* as the four dimensions that provide the basis for modelling the business processes. *DT4BP*, by its *Process, Data, Resource* and *Dependability* models provides explicit support that allows the modeller to address each of these perspectives. The only comparable existing business process modelling language that also provides explicit support for these perspectives is YAWL.

The features related to the collaboration concept are those that define not only the different participants required to take part in the business process, but also the interactions (collaborative efforts) between such participants (aka choreography) required for achieving the business process goal. *DT4BP*, by its built-in constructs *participant, send and receive* provides direct support for these aspects.

One of the characteristics that represents an advancement of *DT4BP* with respect to existing business process modelling languages, is its powerful set of time-related primitives for constraining the temporal behaviour of the business processes. The fact that the primitives for constraining the business process are different from those used to constrain activities, results in clearer semantics, thus, avoiding misconceptions about the time frame of reference associated with each primitive. Furthermore, time primitives for constraining (1) the time a participant is able to work, as well as (2) the duration of the value stored by a object, are novel means that have never been considered by any other notation.

Another aspect that makes *DT4BP* a superior language, is the feature that allows for considering the occurrence of multiple parallel exceptions as a new exception combined with its respective handling so as to avoid the failure of the business process. This feature combined with the ability to consider business processes as long-lived transactions, and the explicit handling of deviations, whether at the level of the participant or the business process, represent support for the dependability concept that goes beyond the one given by existing languages.

Last, but not least is the fact that *DT4BP* is a business process modelling language that provides support for every listed aspect in a integrated manner. The *Process Model* works as a pivot element where the data types defined in the *Data Model* are used to determine the data objects required to perform the different activities that compose the business process. The same principle applies between the resources defined in the *Resource Model* and the allocation policies defined in the *Process Model* used to select the actual resources in charge of performing the business process activities. On the other hand, the *Dependability Model* represents a complementary dimension for the *Process Model* that is used to drive the collaborative handling of simple and parallel deviations that take place during the enactment of the business process.

4. THE TIMED-CAAFWRK CONCEPTUAL FRAMEWORK

Abstract

This chapter describes a new version of the Coordinated Atomic Actions conceptual framework that includes real-time extensions, Timed-CaaFWrk. Here is also presented an implementation framework meant to support the development phase aimed at achieving an implementation that is compliant with certain designs given in terms of the Timed-CaaFWrk. Section 4.1 introduces the role of the Time-CaaFWrk conceptual framework within this thesis, and explains why it was chosen. Section 4.2 describes the ideas that characterise the conceptual framework considered in this thesis. The description of these concepts is done in terms of metamodelling, and thus contributing to the formalisation of the conceptual framework. Section 4.3 presents the real-time extensions given to the conceptual framework to make it possible to design dependable distributed software systems that contain time constraints. In Section 4.4, an implementation framework aimed at supporting the programmers' tasks during the development phase is described. The chapter closes with Section 4.5, describing the use of Model Driven Engineering techniques to obtain automatically generated code from a software system design adhering to the Timed-CaaFWrk conceptual framework.

4.1 Motivation

The DT4BP modelling language introduced and described in Chapter 3 has been defined to ease the description of DCTC business processes. Let M_{DT4BP} be a business process definition modelled using the DT4BP language, and let $< s_1 \xrightarrow{act_1} s_2 \rightarrow ... \xrightarrow{act_k} s_n >$ be a trace generated by the execution of a process instance of M_{DT4BP}. Then the *semantics* of M_{DT4BP} is defined as the set of all possible traces (named $Traces_{M_{DT4BP}}$) that can be generated for the model. These traces are generated by executing a Java program, M_{Java}, obtained by *translating* the M_{DT4BP} into Java. Hence, the translation that maps expressions and statements of the DT4BP language into sequences of Java instructions is part[1] of the *semantic mapping* required when defining a language. In this case, the *semantic domain* is the Java Virtual Machine, which must also be provided when defining a language (see Chapter 2, section 2.3.2 for more information about the elements of a language definition).

[1] The other part of the mapping translates Java instructions into a binary Java class file suitable for a Java runtime system. Since this translation is provided by any Java compiler, it is considered as implicit and thus not considered further.

It is important to note that the Java program M_{Java} is not a general program. M_{Java} complies with the principles defined by the *Timed Coordinated Atomic Actions conceptual framework (Timed-CaaFWrk)*, which was used as the reference to define the DT4BP modelling language. In this manner, since there exists a close relationship between DT4BP and Timed-CaaFWrk concepts, the latter is used as a pivot in the translation process to ensure that the resulting Java program explicitly captures the concepts introduced in the DT4BP models. Figure 4.1 sketches out how this translation process takes places. Reaching a M_{Java} model from a M_{DT4BP} model is a two step process: 1) the M_{DT4BP} model is translated into a model, M_{TCaa}, that is compliant with Timed-CaaFWrk; 2) the M_{TCaa} model is then translated to obtain the Java program M_{Java}.

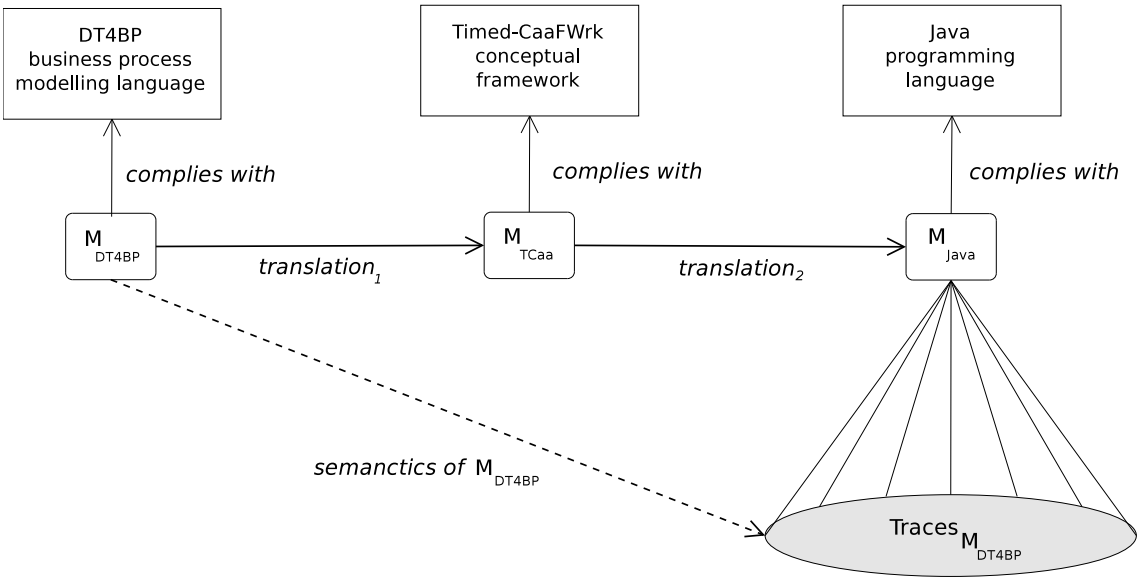

Fig. 4.1: Translation process of DT4BP models into Timed-CaaFWrk-compliant Java programs.

This chapter describes Timed-CaaFWrk and the facilities to obtain, both manually and semi-automatically. Java programs that comply with this conceptual framework. The translation that results in a M_{TCaa} model, compliant with Timed-CaaFWrk, from a given M_{DT4BP} model is explained in Chapter 5.

4.2 The CaaFWrk

The Coordinated atomic actions conceptual framework (CaaFWrk) was defined to encircle the concepts of *conversations* [Ran75], *transaction processing* [GR92], and *exception handling* [Cri89] around the notion of *atomic action*[2] ([BW01], pages 321-322). This results in a new abstraction, *Coordinated Atomic Action (CAA)*, which represents the CaaFWrk's central concept and thus gives the name to the conceptual framework.

The CaaFWrk has been evolving since its first definition [XRR+95], leading to different schemes with their own semantics. (For a complete list of previous works on CAAs, please to see [Rom07]). This section covers not only how the CaaFWrk is considered in the context of this thesis, but

[2] Atomic actions have been also called *multiparty interactions* [EFK89].

also what are the new extensions that have been introduced to fulfil the usage of the CaaFWrk. As has been done thus far, the formalisation of the concepts is given by the metamodelling principle.

4.2.1 Fundamentals

A CAA specifies the orchestration of a set of instructions[3] spread over a group of (one or more) *roles*. The instructions owned by a particular role will be executed concurrently with the instructions owned by the other peer roles enclosed within the same CAA. The actual execution of a particular role is made by a *participant*. Conceptually, this means that a participant is any entity that has the capacity of executing a set of instructions. Thus, a role specifies the instructions to be executed, while a participant is the entity in charge of performing the instructions at run-time. A participant can be a process, a thread, an active object or any mechanism with processing power [CRR09]. Each participant is assumed to have its own features (e.g. number or processors or size of the memory, in the case the participant is a process). Figure 4.2 shows the standard informal way of graphically representing a CAA along with its constitutive parts, which are detailed in the next subsections.

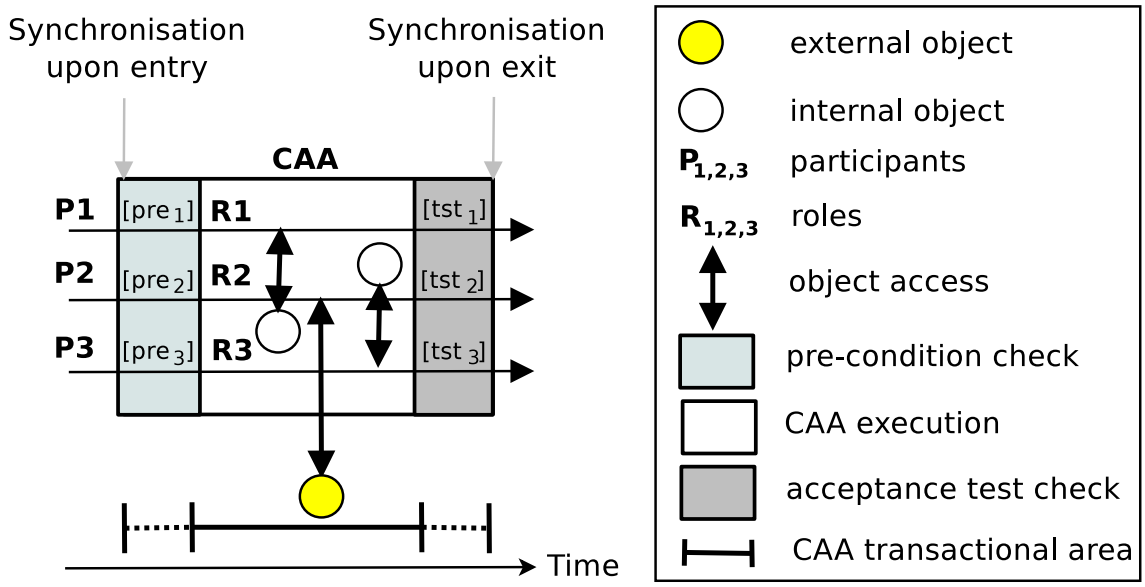

Fig. 4.2: A simple CAA

Figure 4.3 shows the part of the meta-model that formalises the concepts introduced above. The starting point of this formalisation is the class *CAA*, which states that a CAA is composed of an ordered set of roles, captured by the *roles* composite relationship. A *Role* has a name, which is its identifier within the context of the CAA where it belongs. Hence, roles within the same CAA must have different names. The OCL invariant shown in Figure 4.4 ensures this condition. Furthermore, a role is associated with one and only one CAA, which is ensured by the named *caa*. The instructions owned by each role are captured by the *instrs* composite relationship. Note that a role may be modeled with no instructions. The characteristics each participant has (which range from one to many) are modelled with the composite relationship *features*. Each characteristic (of type *Feature*) is modelled as a pair (name, value), where the name is the

[3] An instruction is also known as an *operation, statement* or a *method call*.

identifier of the participant's characteristic, which is unique, and the value is an integer number that quantifies the characteristic for that participant.

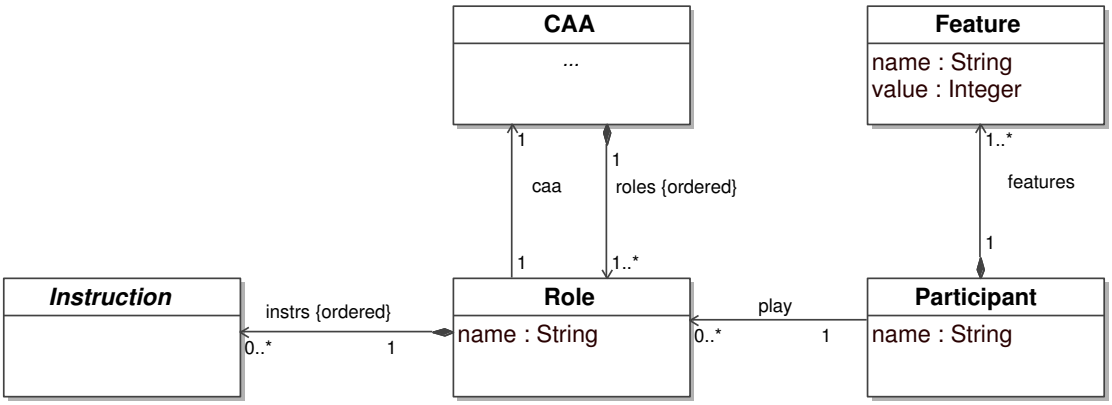

Fig. 4.3: Fundamental parts of a CAA.

```
context Role inv Uniqueness:
        Role.allInstances()->forAll(r1,r2|r1<>r2 implies r1.name<>r2.name)
```

Fig. 4.4: Uniqueness of the role's name.

As already indicated, a role is expected to be played by a participant. Every participant is considered to be unique, so that its name represents the participant's identifier within the CAA design where it is being used. The OCL invariant that ensures this condition is very similar to the one shown in Figure 4.4 and thus it is obviated. Those number of roles that a certain participant is ready to *play* may range from none[4] to many. However, a participant is allowed to play only one role at a time.

The instructions prescribed for a role within certain CAA are meant to collaborate with the instructions prescribed for the other roles of the same CAA in order to achieve a common goal. This common goal can be seen as the service (or functionality) the CAA offers to the enclosing context where it is embedded and computed atomically, i.e. from the CAA user's viewpoint, there is not an intermediate state. The following principles govern the manner in which the service offered for certain CAA is provided at run-time:

4.2.1.1 Agreement upon entry

To execute a CAA, there must exist a group of participants such that each of them agrees to perform one (and only one) role of the CAA. The agreement to take on a certain role is defined as the satisfaction of the pre-condition owned by the role. Using metamoddelling, this is formalised in Figure 4.5. The composite relationship named *pre* captures the role's condition the participant that wants to play this role has to satisfy. These conditions are assumed to be first-order logical expressions written using the *OCL* language [WK03], which are both decidable[5]

[4] The fact that a participant is not required to be associated with at least one role upon its definition allows for the association between them to be defined later.

[5] There exists a Turing machine that always halts given any finite input string.

and computed within a reasonable amount of time. The type *OclConstraint* captures this kind of OCL expression. For the case in which the role does not hold a condition, the participant is assumed to satisfy the trivial condition, i.e. *true*.

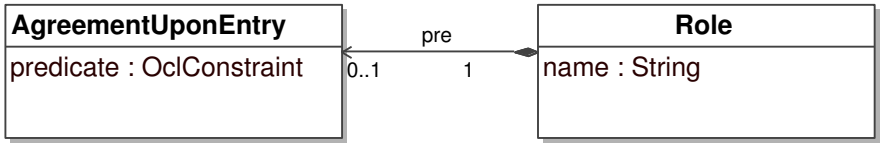

Fig. 4.5: A role may have an associated pre-condition.

Once every participant has agreed to take on a certain role, the CAA is said to be ready to start its execution. Participants in charge of executing certain CAA are assumed to enter and leave such CAA synchronously [XRR+02]. In Figure 4.2, the pre-condition area represents the checking of the pre-conditions owned by the CAA's roles. This pre-condition checking is conceptually considered as happening without any delay. It is worth mentioning that the conjunction of the pre-conditions owned for each role is considered as the *pre-condition* of the CAAs, which is expected to be met by those participants that come together to execute the CAA. It is assumed that a CAA executes only if its pre-condition is satisfied.

4.2.1.2 Internal execution

A CAA starts its execution when every participant starts its associated role. Playing a role means that the participant executes the set of instructions prescribed to the role. A role may contain three kinds of instructions: *execution, control* or *declaration* (see formalisation in Figure 4.6).

- *execution:* instructions of this kind allow a role to (1) execute and operate, e.g. perform a computation or assign a value to a variable, (2) send/receive information to/from a different role, (3) call another CAA, or (4) enclose instructions into a block such that they are considered as a single instruction. The concept *Execute* allows the execution of an operation to be modelled . The operation, captured by the attribute of the same name, is described in a declarative manner using the notions of pre-condition and post-condition, captured by the attributes *pre* and *post*, respectively. Roles belonging to the same CAA are allowed to freely communicate amongst themselves, but not with any other roles not owned by the CAA. The concepts *Send* and *Receive* are the means to model the communication (by information exchange) between roles. A role may communicate with one or more peer roles. This is modelled by the association *to*. Whereas a role is allowed to receive information from only one peer. This modelled by the association *from*. The instruction that allows a role to call other CAA is captured by the concept *CallCAA*. The *Block* instruction encloses an ordered sequence of none or many instructions as if they were a single instruction.

- *control:* instructions of this kind are those that allow a role to control the execution flow of instructions. The flow can be controlled by the concept of *If*. *If* defines an alternative between the branches *then* and *else* and is based on the condition captured by the attribute *cond* of type *OclConstraint*. The other control instruction are *While* and *Repeat*, which

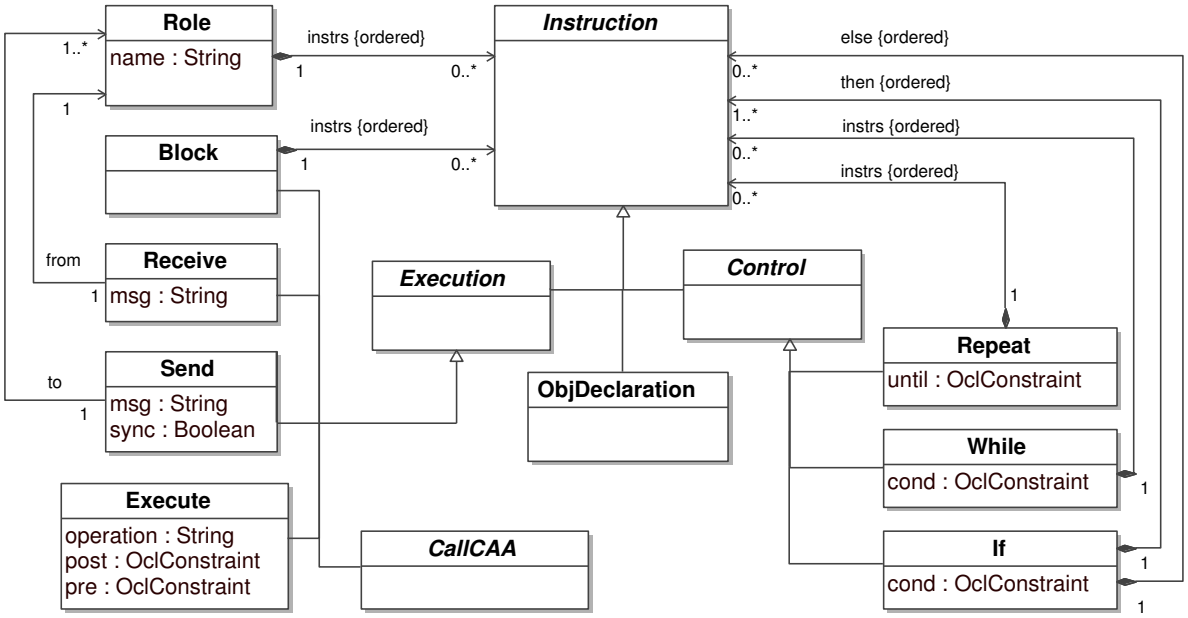

Fig. 4.6: Instruction set.

loop the execution of an ordered sequence of instructions (captured by the composite relationship *instrs*) only while their respective conditions hold.

- *declaration:* there exists only one instruction of this kind, *ObjDeclaration*. It is meant to capture the modelling of an object declaration. Objects declared using this instruction are referred to as *internal or local* objects. The concept that captures this kind of object is an *InternalObj* (see Figure 4.7). Notice (1) that the instruction *ObjDeclaration* may declare one or more internal objects, captured by the composite relationship *internalObjs*), and (2) that when declaring an internal object its type must be provided since every object is associated with a type. This is captured by the association *type* that links the classes *Type* and *Object* from which *InternalObj* extends.

A *Type* is either a *Primitive* datatype (i.e. *DString, DInteger, DFloat,* or *DBoolean*) or a structured *DataType*. A *DataType* is defined as a set of attributes (described by the composite relationship *attrs*) or as an ordered sequence of literals (described by the ordered composite relationship *enum*) such that it defines an enumerated datatype. *DataType* may also include an invariant, which is captured by the attribute of the same name. Invariants are first-order logical formulas written in OCL under the previously states assumptions. Regarding the attributes, each attribute is also associated with a *Type*, which is either *Primitive* or *DataType*.

Please recall that roles enclosed within a same CAA are allowed to freely communicate amongst themselves by exchanging information. The information they are allowed to exchange are internal objects previously declared in each role. As shown in the formalisation of Figure 4.7, a role may send a message, *msg*, to another role along with one or more internal objects. Whereas a role is allowed to receive a message along with only one internal object. Regardless of whether a role sends a message or not, the real communication between peer roles is achieved through the information carried by the internal objects sent. That is the reason why it is claimed that

roles cooperate amongst themselves by *sharing information*.

Other objects that may take place within the CaaFWrk are those declared outside the boundaries of a particular CAA. These objects are referred to as *external* objects. The external objects, *ExternalObj*, that are required by a CAA to attain its goal are passed as parameters upon its call. The ordered composite relationship named *params* captures those external objects to be used by the CAA to fulfill its execution. It must be noted that (1) every CAA's external object is also associated with a *Type*, either *Primitive* or *DataType*, that is associated with the CAA by the association *dttps*.

The CAA's external objects are reachable from every CAA's role. The instructions these roles prescribe over the CAA's external objects are executed according to the transaction processing principles: 1) either all instructions are completed or none are completed (atomicity), 2) concurrent instructions are free from interference[6](isolation), once all the instructions have been executed the external objects 3) are in the correct state (consistency) and 4) they will survive failures (durability). Therefore, a CAA guarantees the ACID properties over its external objects.

It is worth noting that certain external objects can be accessed concurrently (aka competitive concurrency) by different CAAs at the same time. Hence, depending on the kind of access (i.e. read only, or read and write) the CAA's roles allow over a particular external object, the CAA may gain exclusive access over the external object. The attribute *access* owned by each external object determines how the object is accessed. However, assigning a value to this attribute, which changes from one CAA to another, is managed by the transactional system on which the CaaFWrk relies on to achieve the ACID properties. Therefore this topic is not considered further in this thesis.

4.2.1.3 Agreement upon exit

To determine whether a CAA has performed acceptably, the participants in playing the CAA's roles have to agree that the result is acceptable. This agreement is defined by an acceptance test, i.e. a first-order logical expression without side effects, owned by each role and captured by the composite relationship named *post* - see Figure 4.8). The acceptance must be satisfied by the participant before leaving the CAA. The conjunction of all role acceptance tests determines the global acceptance of the CAA (aka CAA's post-condition). A role that is not assigned an acceptance test, is assumed has its trivial logical expression as *true*.

As was assumed for the pre-condition checking, the evaluation of the acceptance tests are considered to happen without any delay. A global acceptance test that evaluates to false, i.e. at least one of the role's acceptance test fails, represents an erroneous condition. As will be explained in the following discussion, the detection of an error condition initiates the CAA recovery mechanism.

4.2.1.4 Exception handling

As explained in Chapter 2, Section 2.2.2, fault-tolerance is achieved by error detection and recovery. The CaaFWrk makes use of the principles coming from exception handling [Goo75, Cri89, BM00] to achieve error detection and recovery, and thus to include fault tolerance as the means to attain dependability.

[6] Their execution is equivalent to some serial order of execution. Thus, it is said that they satisfy the serialisability property.

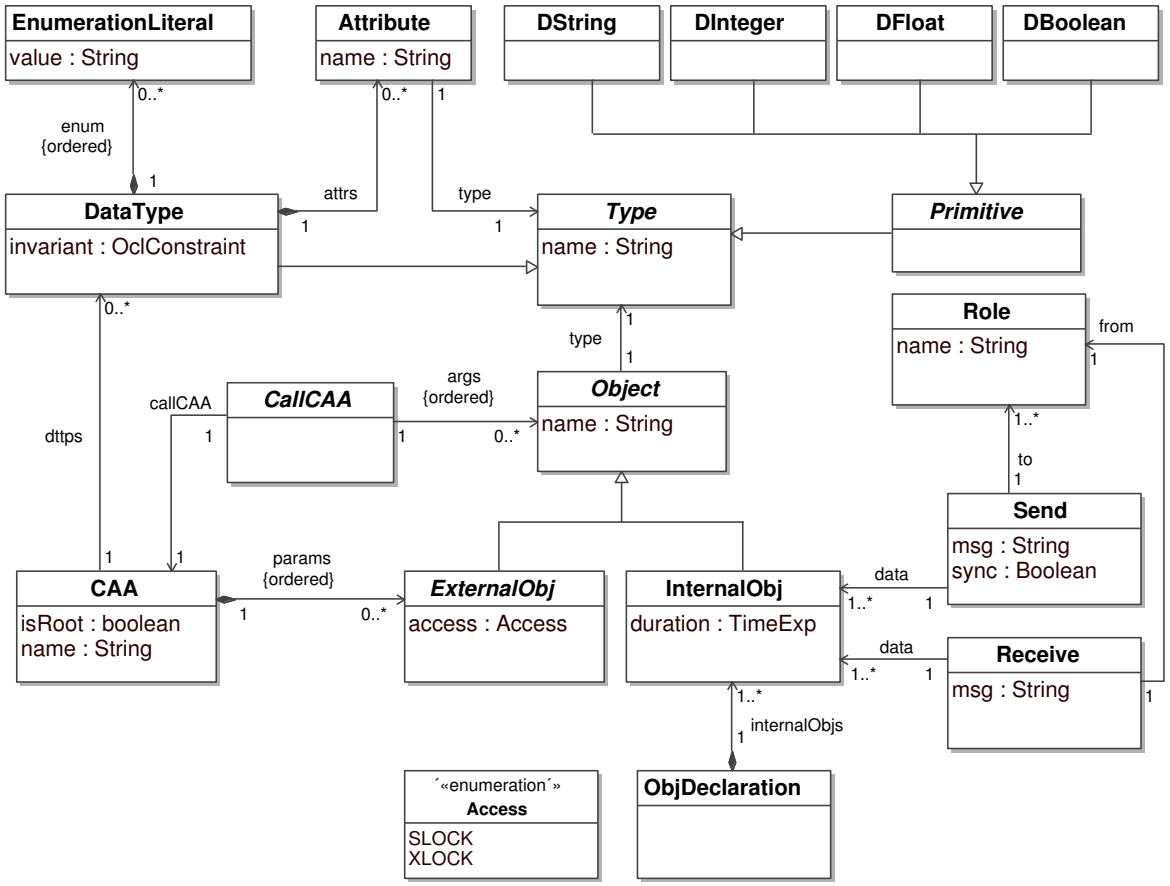

Fig. 4.7: Kinds of objects.

1. **Detection:** within the CaaFWrk, an exception is raised to indicate that an error condition occurred. An exception, e.g. a division by zero, can be raised by any of the CAA software or hardware layers that execute at run-time, or by the particular CAA itself when a certain condition is detected, e.g. when an acceptance test does not hold. In any case, the semantics of a raised exception is always the same: an error has been detected. The error signaled by anexception is always the same regardless of where it takes place. For example, the semantics of the division by zero exception is always the same whether it is raised in one CAA or in an other. Each exception has a *name* (see Figure 4.9), which is unique within the CAAs (the OCL invariant named *Uniqueness*, which is shown in Figure 4.10 ensures this condition). The name of a particular exception thus is its identifier within the entire CAA design being carried out.

   The an exception in a CAA always occurs in one of its roles. The case in which a role detects an error condition deserves special attention, since the subsequent procedure it is to notify the peer roles of the error. In order to fulfill this requirement, the special instruction *Raise* is introduced. This instruction is used in all instances to model an exception by a role. It is worth noticing that this instruction may only be used within a role. The OCL invariant named *onlyInRoles*, shown in Figure 4.10, ensures this condition.

2. **Concurrent exceptions:** because participants play out their roles concurrently within a

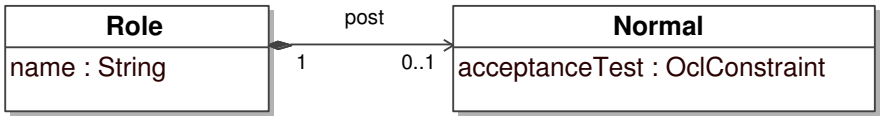

Fig. 4.8: A role may have an associated post-condition.

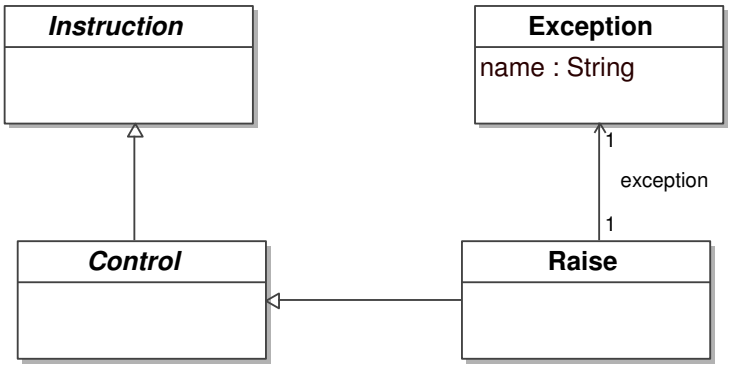

Fig. 4.9: Raise of an exception.

CAA, multiple exceptions might be raised simultaneously. This set of concurrently raised exceptions represents a different and more complicated exception.

Xu et al. [XRR98] propose the notion of *exception graph* to define at design time the potential concurrent exceptions that may be raised within a CAA and that must be handled. This exception graph is defined as follows:

- there exists a node e_i for each single exception that (1) may be raised within the CAA and (2) requires handling
- there exists a node e_j for each set of ES_j exceptions e_{k_i} ($i = 1..n$, with $n \geqslant 2$) such that may be concurrently raised within the CAA and need to be handled,
- there exists a directed edge from e_j to e_{k_i} for each $e_{k_i} \in ES_j$, which represents an exception e_j raised when exceptions e_{k_i} ($i = 1..n$) are raised concurrently,
- there exists a unique node μ, and
- there exists a directed edge from μ to e_j for each node e_j with $degree_{in}(e_j) = 0$, where $degree_{in}(e_j)$ is defined as the number of edges entering to the node e_j.

Hence, considering the number of edges that enter into a node (i.e $degree_{in}(e_i)$) and the number of edges that leave from it (i.e. $degree_{out}(e_i)$), an exception graph is composed of three kinds of nodes: (1) nodes with $degree_{out}(e_i) = 0$, which represent primitive exceptions that cover no other exceptions; 2) nodes with $degree_{in}(e_i) \neq 0$ and $degree_{out}(e_i) \neq 0$, which resolving exceptions and cover other exceptions, and (3) one node with $degree_{in}(e_i) = 0$ (i.e. node μ), named the root of the exception graph and representing the so-called *universal* exception. This universal exception is assumed to cover every concurrent or single exception that is not modelled at design-time in the exception graph but, that may occur at run-time. Hence, any raised exception, regardless of whether it is a single or

```
context Exception inv Uniqueness:
        Exception.allInstances()->forAll(e1,e2|e1<>e2 implies e1.name<>e2.name)

context Raise inv onlyInRoles:
        Raise.allInstances()->forAll(raiseInstr|Role.allInstances()->
            exists(role|role.instrs->includes(raiseInstr)))
```

Fig. 4.10: Uniqueness of the exception's name, and the correct use of the instruction *Raise*.

a concurrent exception, that has not been explicitly modelled in the exception graph is resolved with the universal exception.

Figure 4.11 shows an example of a simple exception graph for a certain CAA. In this example, the CAA may raise the single exceptions e_i ($i = 1..5$), and the concurrent exceptions $e_1 \wedge e_2 \wedge e_3$, $e_3 \wedge e_4$ and $e_1 \wedge e_2 \wedge e_3 \wedge e_4$. The figure also shows the node that represents the*universal* exception along with the directed edges that make it the root of the graph and indicating that it resolves every exception that is not part of the exception graph. Hence, the concurrent exceptions $e_1 \wedge e_2$, $e_2 \wedge e_3$, $e_1 \wedge e_3$, $e_1 \wedge e_4$, or $e_4 \wedge e_5$ (among others) are resolved as the universal exception.

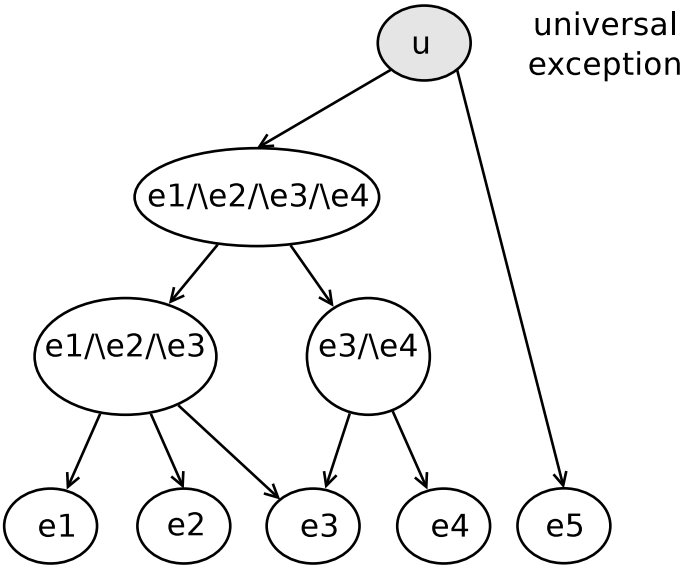

Fig. 4.11: Exception graph.

The part of the meta-model that allows for the capture of the exception graph is shown in Figure 4.12. The relationship named *madeOf* is devised to capture the directed edges that associate an exception e_j with the exceptions e_{k_i} (with $i = 0..*$). This information specifies that the exception e_j is the resolving exception for the concurrently raised exceptions e_{k_i}. The particular case in which an exception e_j is not made of any exception (i.e. $e_j.madeOf.isEmpty()$ in OCL terms) models the fact that e_j is a single exception.

The exception graph for each CAA is used at run-time by the *exception resolution mechanism* to identify the exception raised and subsequently handled by the *recovery mechanism* of the CAA. The recovery mechanisms is explained in the following discussion.

To handle multiple concurrent exceptions [CR86, CR86] and [XRR98] propose that the exception resolution mechanism will raise the exception at the root of the smallest subtree

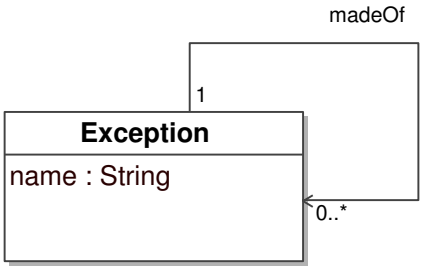

Fig. 4.12: Meta-model component for capturing the representation of an exception graph.

containing all the exceptions as the one to be handled by the recovery mechanism. Notice that the root must exactly cover the exceptions raised concurrently. Thus, for the exception graph shown in Figure 4.11, the exception resolution mechanism will signal the exception $e_1 \wedge e_2 \wedge e_3$ for the case when exceptions e_1, e_2 and e_3 are raised concurrently. However, the exception resolution mechanism will raise the universal exception μ in the case that the exceptions e_1 and e_2 are raised concurrently as there is no node in the exception graph that exactly covers all of these exceptions. It is worth mentioning that the case when only one exception is raised represents a particular case, since every single exception is considered as covered by itself. Thus, the exception resolution mechanism works when multiple exceptions are raised concurrently as well as when only one exception is raised.

Due to the way in which the exception graph is modelled, the exception resolution mechanism used in the context of this thesis is different. Instead of finding the root of the smallest subtree containing all the exceptions raised concurrently, the mechanism looks for an exception such that its *madeOf* association contains exactly those exceptions that were raised concurrently. In the case that no exception exists the mechanism has to raise the universal exception. The OCL condition shown in Figure 4.13 ensures that every resolving exception e_j (i.e. $e_j.madeOf->size() > 0$) is made of different exceptions. Otherwise, the graph is ambiguous since there exist two different resolving exceptions for the same set of concurrent exceptions.

```
context Exception inv UniqueResolvingException:
        Exception.allInstances()->forAll(e1,e2|e1<>e2 and
                                    e1.madeOf->size()>0 and e2.madeOf->size()>0 implies
                                    e1.madeOf<>e2.madeOf)
```

Fig. 4.13: Uniqueness of the resolving exceptions.

3. **Recovery:** once an exception has been raised, a recovery mechanism should be used to mask or mitigate the impact of the situation on the CAA. The recovery mechanism provided by the CaaFWrk is based on the notion of exception handling. This means that the CaaFWrk recovery mechanism includes support to replace the normal CAA execution by an exceptional execution when an exception is raised. The exceptional execution that is put in place when an exception occurs is known as the *Handler* associated with such an exception (see formalisation in Figure 4.14). The general aim of a handler is to restore the normal computation by putting the software system into a correct, i.e. error-free state. Whether a handler is able to restore the normal computation or not, depends on the

severity of the raised exception. As shown in the formalisation by the association named *handles*, the same handler can be used to deal with different exceptions.

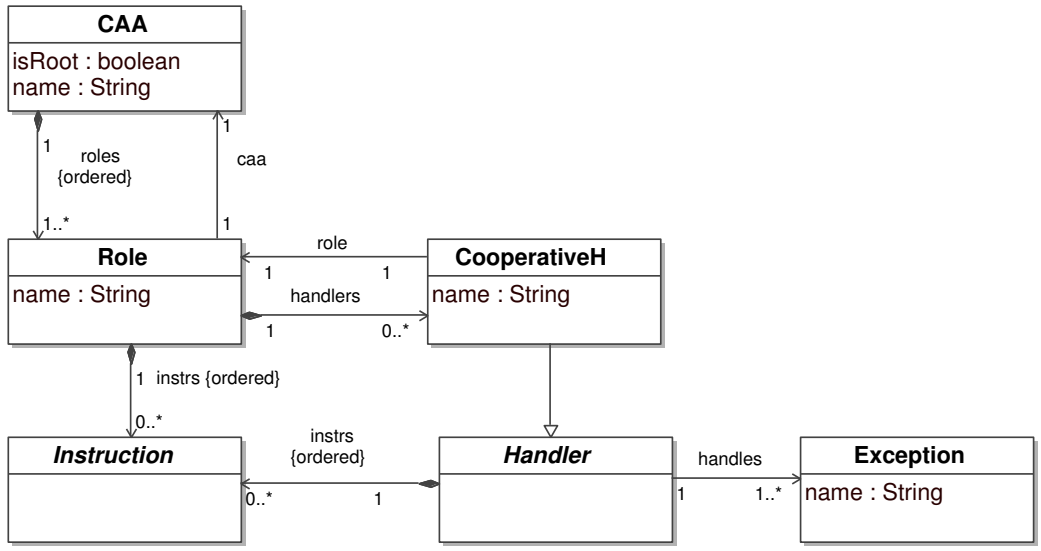

Fig. 4.14: Cooperative handling of an exception.

Depending on what the correct state is, handlers can be classified as performing *forward error recovery (FER)* or *backward error recovery (BER)*. A handler is said to perform FER when it puts the software system into a usually non-visited correct state. This is done by analysing the detected exception, e.g. looking at its type or the parameters it carries, and using redundant information. Conversely, a handler performs BER when the correct state is that which the software system was executing before the exception occurrence. Hence, BER is a particular case of FER.

In the context of the CaaFWrk, performing a BER over a CAA is equivalent to restoring every CAA external object to the state they had before execution of CAA began. The operations necessary to undo the effects of the CAA onits external objects, i.e. the nothing part in the all-or-nothing property provided by Atomicity, are provided by the external transactional support on which the CaaFWrk relies to ensure the ACID properties on the external objects. Therefore, BER is considered as a built-in component that every CAA has, and, thus, its modelling is not required.

Conversely, the designer of the CAA must define the instructions executed by a handler that performs a FER. This behaviour (captured in the meta-model shown in Figure 4.14 by the ordered composite relationship *instrs*) depends on the specific characteristics of the software system under development, and the severity of the exceptions to be handled. The FER of a particular CAA is defined as a set of handlers (i.e. $FER = \{hnd_1, ..., hnd_n\}$). Each handler hnd_i is meant to administer at least one of the exceptions the CAA is supposed to deal with, i.e. each exception that belongs to the exception graph, excluding the universal exception.

Each handler $hnd_i \in FER$ is defined as a set of cooperative handlers (i.e. $hnd_i = \{coopHnd_1, ..., cooprHnd_k\}$). More exactly, there must exist one cooperative handler $coopHnd_i$ for each role $role_i$ $(i = 1..k)$ in the CAA. This is because of the cooperation principle: if a set of roles cooperate to reach the CAA's goal, then they also have to cooperate when

handling an exception. Thus, when an exception e (of type *Exception*, according to the formalisation shown in Figure 4.14) is is detected by one of the roles $role_i$ (of type *Role*) in the CAA, the other roles $role_j$ ($j = 1..k$ and $j \neq i$) are *informed*. Subsequently, each role $role_i$ starts executing its cooperative handler $coopHnd_i$ (of type *CooperativeH*). Therefore, for each exception e_i the CAA is supposed to deal with, a $hnd_i \in FER$ must be defined, such that the number of cooperative handlers, hnd_i, is equal to the number of roles in the CAA. The OCL condition shown in Figure 4.15 checks that this condition holds for every CAA.

```
context CAA inv bindingRoleCooperativeHnd:
  let
    exceptions:Set(Exception)= Exception.allInstances()->select(e|
      let
        coopHnds:Set(CooperativeH)= CooperativeH.allInstances()->
                                    select(coopH|coopH.role.caa=self)
      in
        coopHnds->exists(coopHnd|coopHnd.handles=e)
    )
  in
    exceptions->forAll(e|
        let
          nRoles:Integer=self.roles->size(),
          nCoopHnds:Integer=CooperativeH.allInstances()->
            select(coopH|coopH.role.caa=self and coopH.handles=e)->size()
        in
          nRoles=nCoopHnds)
```

Fig. 4.15: Soundness between the number of roles and cooperative handlers for each exception handled within a CAA.

Note that the binding between a role and its respective handler is captured by the association *role*, whereas the composite relationship named *handlers* captures the different cooperative handlers owned by a role to administer each of the different exceptions.

The CaaFWrk recovery mechanism combines FER and BER when handling an exception. The statecharts [Har87, HN96] shown in Figure 4.16 describe the life cycle of the recovery mechanism at run-time. The statecharts initially set the CAA in the state *Service*, which represents the execution of its service. It may happen that either:

(a) a set *ES* of one or more exceptions are raised while the CAA is still executing (represented by the event *Exception(ES)*), or

(b) the CAA completes its execution, which is represented by the event *caaDone*. However, its post-condition, i.e. global acceptance test, does not hold and is represented by he predicate *not(postCond)*. A non-holding post-condition is an erroneous situation. A CAA post-condition does not hold when at least one of the role acceptance tests is false. Each role $role_i$ for which its acceptance test does not hold raises an exception $ExAcceptanceTest_{role_i}$ such that $ES = \{ExAcceptanceTest_{role_i}\}$.

Both cases activate the resolution mechanism to initiate the search for the resolving exception e in the set of raised exceptions *ES*. Depending of the exception found, the recovery will be either FER or BER.

(a) if the resolving exception e found by the resolution mechanism is not the universal exception, represented by the predicate *Resolution(ES)=e*, then the FER to handle exception e is initiated (represented by the state *FER(e)*). In case that either:

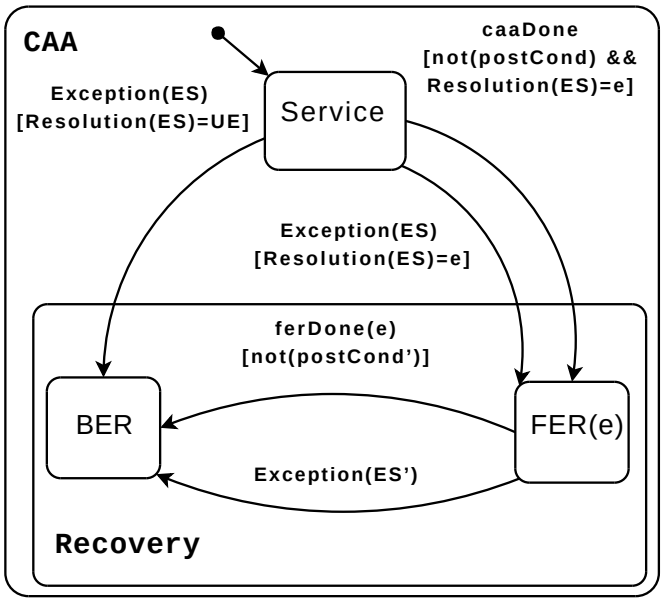

Fig. 4.16: CaaFWrk recovery mechanism life cycle

i. one (or more) exceptions are raised during the execution of the FER (represented by the event *Exception(ES')*), or

ii. the FER completes its execution (represented by the event *ferDone(e)*), but its post-condition, which may or not be equal to the CAA post-condition, does not hold (represented by the predicate *not(PostCond')*),

then the BER is started.

(b) if the resolving exception e found by the resolution mechanism is the universal exception (represented by the predicate *Resolution(ES)=UE*), then the BER (represented by the state *BER*) is started.

4.2.1.5 Outcomes

It has been said that (1) the CAA's goal is the service or functionality it offers to the enclosing context where the CAA is embedded, and (2) from the CAA's user viewpoint this service is provided atomically, i.e. there is not any visible intermediate state between a CAA request and its returned outcome. However, the outcome a CAA returns to its user depends on not only whether the CAA is faced or not with an exception, but also whether the recovery mechanism succeeds or not in dealing with such an exception. Thus, the raising of an exception and the behaviour of the recovery mechanism in dealing with it are the key elements that determine the outcome the CAA returns to its enclosing context.

In the same manner as done in the previous section, a statecharts is used to describe the CAA life cycle and the different outcomes that are returned to the enclosing context depending on the final state that is reached. The statecharts that describes the CAA life cycle is shown in Figure 4.17. Notice that this statecharts is an extension of the one shown in Figure 4.16, which describes the life cycle of the recovery mechanism. This is because, as was previously stated, the behaviour of the recovery mechanism is one of the key elements that determines the overall internal behaviour of the CAA.

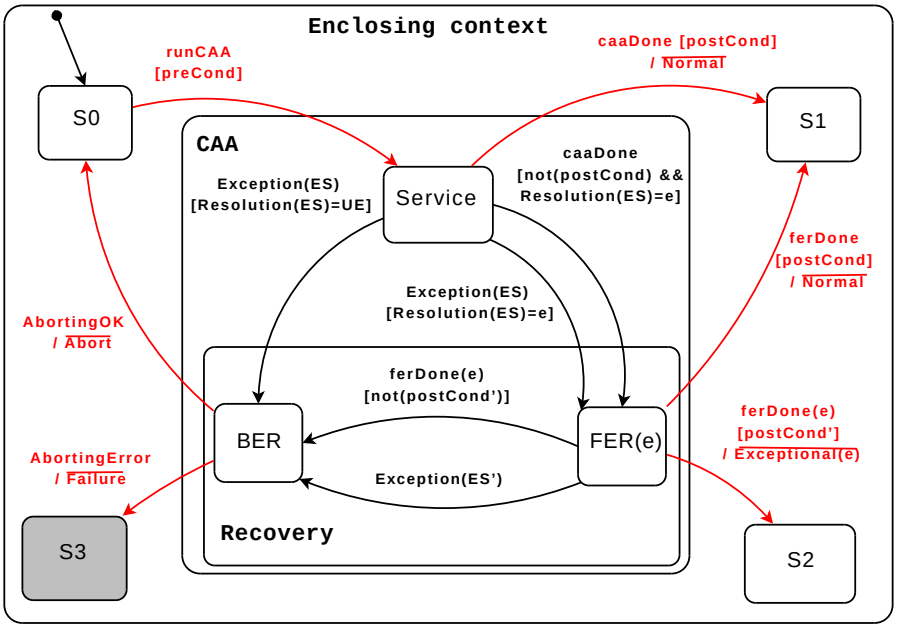

Fig. 4.17: CAA life cycle and outcomes

The enclosing context is assumed to be initially in a certain state S_0, from which a request to the CAA is performed (represented by the event *runCAA*. The CAA will start executing its service, state *Service*, only if the pre-condition holds, the predicate *preCond*.

The specification says that the state S_1 is reached when the event *caaDone* occurs and the predicate *postCond* holds. This state transition represents the best possible scenario in the CAA life cycle. This scenario corresponds to the completion of the CAA without any exceptions and holding its associated post-condition. In this case, the outcome provided by the CAA to the enclosing context is referred to as *normal*. This outcome is described in the specification by the broadcast event[7] *Normal*, which is generated due to the transition from state *Service* to S_1.

The other possible scenarios arise when an exception occurs. As explained in the recovery part of Section 4.2.1.4, when an exception is raised the resolution and recovery mechanisms have to be activated, in that order. In this situation, the life cycle of the CAA is defined by the behaviour of these two mechanisms. However, depending of the success of the exception handler, different outcomes can be produced.

When the recovery is carried out by means of the FER, the outcomes are either *normal* or *exceptional*. The outcome *normal* is produced when the FER is able to both mask the raised exception and to meet the expected post-condition. Thus, the CAA provides to its enclosing context the service as originally expected, returning a *normal* outcome. The case in which the FER is able to mask the raised exception, but a degraded version of the originally expected service (specified by the logical formula *postCond'*) is achieved, produces the *exceptional* outcome. This outcome is described in the specification by the broadcast event *Exceptional(e)*, where e is the exception that carries information about the degraded outcome having been achieved. This event is generated due to the transition from state *FER(e)* to S_2. It is worth noting that despite the fact that the state S_2 is not the state originally expected, it is still sufficient to allow the software system to continue executing.

[7] Event with the line above.

In case the recovery is carried out by means of BER, either because the resolution mechanism resolved to the universal exception (specified in the statecharts by the event *Exception(ES)[Resolution(ES)=UE]*), or an exception was raised during execution (specified in the statecharts by the event *Exception(ES')*), the produced outcomes are either *abort* or *failure*. The CAA returns *abort* when the BER succeeded in undoing the effects over the CAA's external objects and moving the software system to the same state it had before the CAA was requested, i.e. state S_0. In case the BER fails in reaching such a state, the CAA is considered as having failed. This is the worst case scenario, since the software system has been left in a corrupted state (in the statecharts, the grey-colour state S_3 represents this kind of state).

Having explained the different possible outcomes a CAA may produce, their formalisation in terms of the metamodelling principle is given in the Figure 4.18. This formalisation describes the four possible outcomes a CAA may reach by specialising the abstract class *Outcome* in *Normal, Exceptional, Aborted*, and *Failed*. Notice that both *Normal* and *Exceptional* include an attribute that is meant to capture the first-order logical expression that describes the acceptance test to be passed such that the respective outcome can be returned. It is worth noticing that while a role may only produce *Normal* outcomes, i.e. what the composite relationship *post* models, a cooperative handler is allowed to return one of the four different outcomes, i.e. what the composite relationship *outcome* models.

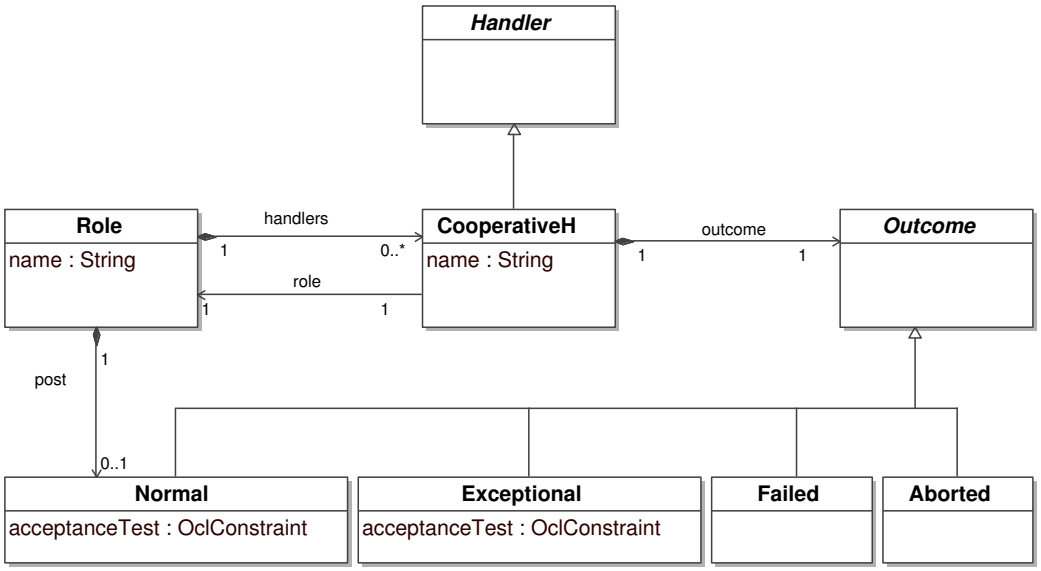

Fig. 4.18: Metamodelling formalisation of the CAA outcomes.

There is an important point that deserves special attention. An *Exceptional* outcome, which determines a degraded response from the CAA to its caller, is achieved by *signalling* an exception. This points out that part of the internal behaviour of the FER may include an instruction to signal the exception. The special instruction *Signal* allows an exception to be signalled to the CAA enclosing context. The formalisation of this instruction is shown in Figure 4.19. The *Signal* instruction is considered as a kind of *Control* instruction since its execution not only signals the exception (that is captured by the association with of the same name), but also transfers the flow control from the CAA to its caller. The instruction *Signal* is allowed to be used only within a cooperative handler $coopHnd_i$ of type *CooperativeH*. The OCL condition shown in Figure 4.20 checks this constraint.

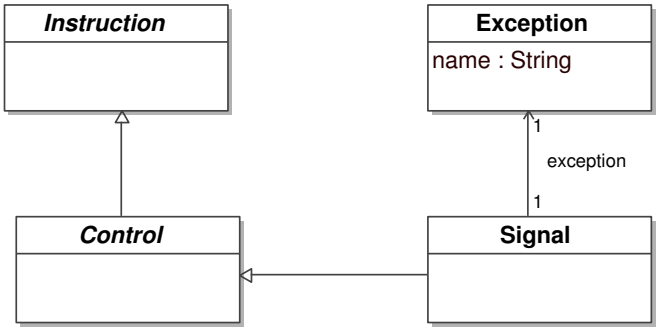

Fig. 4.19: Instruction to signal an exception.

```
context Signal inv onlyInCooperativeHnds:
        Signal.allInstances()->
          forAll(signalInstr|CooperativeH.allInstances()->
            exists(coopHnd|coopHnd.instrs->includes(signalInstr)
          )
        )
```

Fig. 4.20: The *Signal* instruction may only be used within a cooperative handler.

4.2.1.6 Nested and composite CAAs

Another important characteristic of a CAA is that it can be designed in a structured way using the notion of **nested** and **composite** CAA. Using the classic informal notation introduced at the beginning of this section, Figure 4.21 depicts a CAA named CAA_1 that is structured by one nested CAA named CAA_2, and one composite CAA named CAA_3. This figure can be used to explain the characteristics of these structuring facilities.

Both nested and composite CAA's are designed to be embedded into another CAA. Their differences are (1) in the participants that play each of the roles they are made of, and (2) the way in which they are invoked. A nested CAA, e.g. CAA_2, is played by a subset of the participants (e.g. P_1 and P_2) that carry out the roles of its enclosing CAA, e.g. CAA_1. A nested CAA starts its execution once every role in the enclosing CAA (e.g. R_1 and R_2) has passed control over to one of the roles of the nested CAA (e.g. $R_1^{'}$ and $R_2^{'}$) . This distinction points out that each role in the nested CAA is explicitly called by a role of the enclosing CAA. Once all these calls have been performed, the nested CAA starts. The activities carried out by each of the participants within the nested CAA are hidden to both the other roles being played by participants that that did not enter into the nesting (e.g. R_3), and to any other nested or composite CAA (e.g. CAA_3) being executed by the enclosing CAA at the same time.

Composite CAAs [TLIR] (e.g. CAA_3) are more flexible than CAAs, since their use does not require any participant already engaged in the execution of the roles of the enclosing CAA. The participants required to play a composite CAA role (e.g. $P_1^{'}, P_2^{'}$ and $P_3^{'}$) are assumed to be assigned upon the call of the composite CAA. A composite CAA starts executing its roles (e.g. $R_1^{''}, R_2^{''}$, and $R_3^{''}$) once it is invoked by a role in the enclosing CAA (e.g. R_3). This role synchronously waits for the outcome of the composite CAA. Then, the calling role resumes its execution according to the outcome of CAA_3. The execution being performed by the composite CAA is hidden from the enclosing CAA. The effects of having executed a composite CAA are

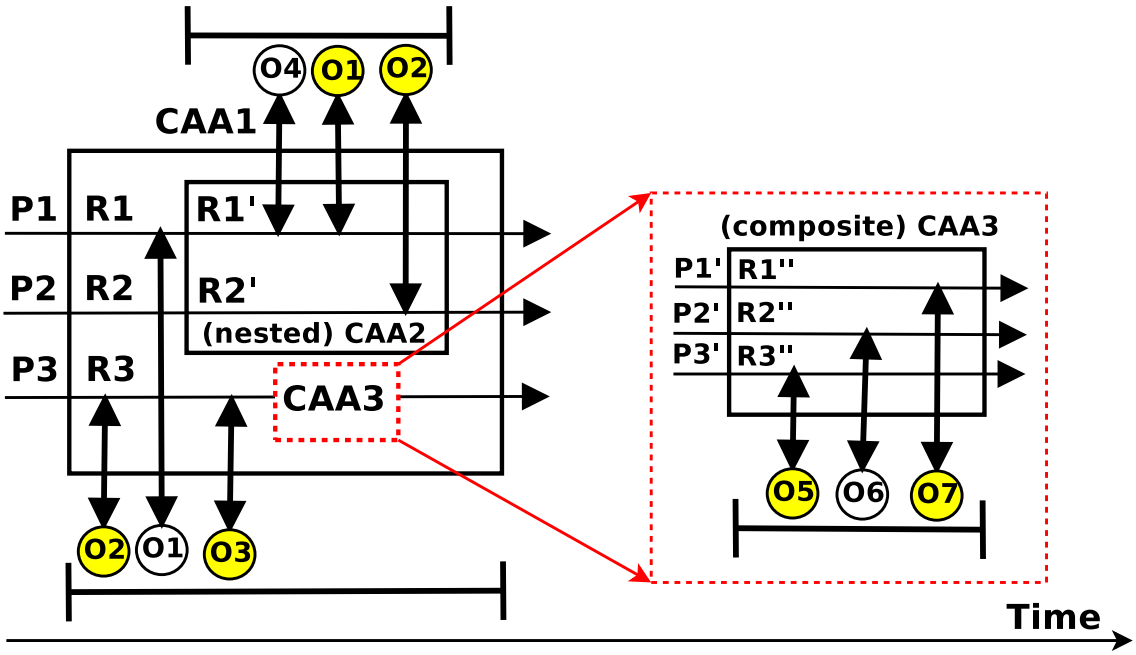

Fig. 4.21: Nested and composite CAAs.

known only by the changes it makes to those objects that were passed to it as parameters, e.g. O_5 and O_7.

The a nested CAA differs from a composite CAA by the participants that play them as well as by the way they are invoked. Thus, there is nothing preventing the use of the CAA in one instance as nested and at another as composite within the same design. It is the caller of the CAA who decides whether it is nested or composed. When nesting a CAA, roles of the enclosing CAA must use the instruction *CallNested* to invoke the nested CAA (see the part of the meta-model shown in Figure 4.22). This instruction indicates which CAA is to be nested. (This is captured by the association named *callCAA* that is inherited from the abstract class *CallCAA*.) In addition, the instruction also indicates the role to be played by the participant carrying out the invocation. (This is captured by the association named *callRole*.) The case in which a role invokes a composite CAA is achieved by using the instruction *CallComposite*. This instruction only requires knowledge of which CAA is to be invoked. Each CAA has a unique name within the design. The OCL invariant named *Uniqueness* that ensures this condition is shown in Figure 4.23.

It is assumed that in each CAA design there exists an outer most CAA, which defines the boundaries between the software system structured in terms of CAAs, and the environment where the software system is meant to be deployed [ZRX+99]. This outer most CAA, which is referred to as *root CAA*, is assumed to be (1) unique (the invariant checks this condition), and (2) started by an event coming from the environment, whereas the other CAAs (i.e. nested and composite CAAs) are started by explicit (either *CallNested* or *CallComposite*) calls. It must be noted that recursive calls are not allowed (the invariant *nonSelfReference* checks this condition).

An important consideration regards the objects that are passed as arguments over a CAA (regardless it is a nested or a composite) from its enclosing context (e.g. O_1). Objects passed as arguments over a CAA are always considered as *external* for such a CAA. Hence, an *internal* object belonging to the caller CAA that is passed as an argument becomes *external* for the

CAA. This shows that the notions of *external* and *internal* objects are relative to the CAA where they are used. Of course, the type of each object being passed as an argument when calling a CAA must match the type of the parameter expected by the called CAA (the invariant *CheckTypesOnParams* checks this condition).

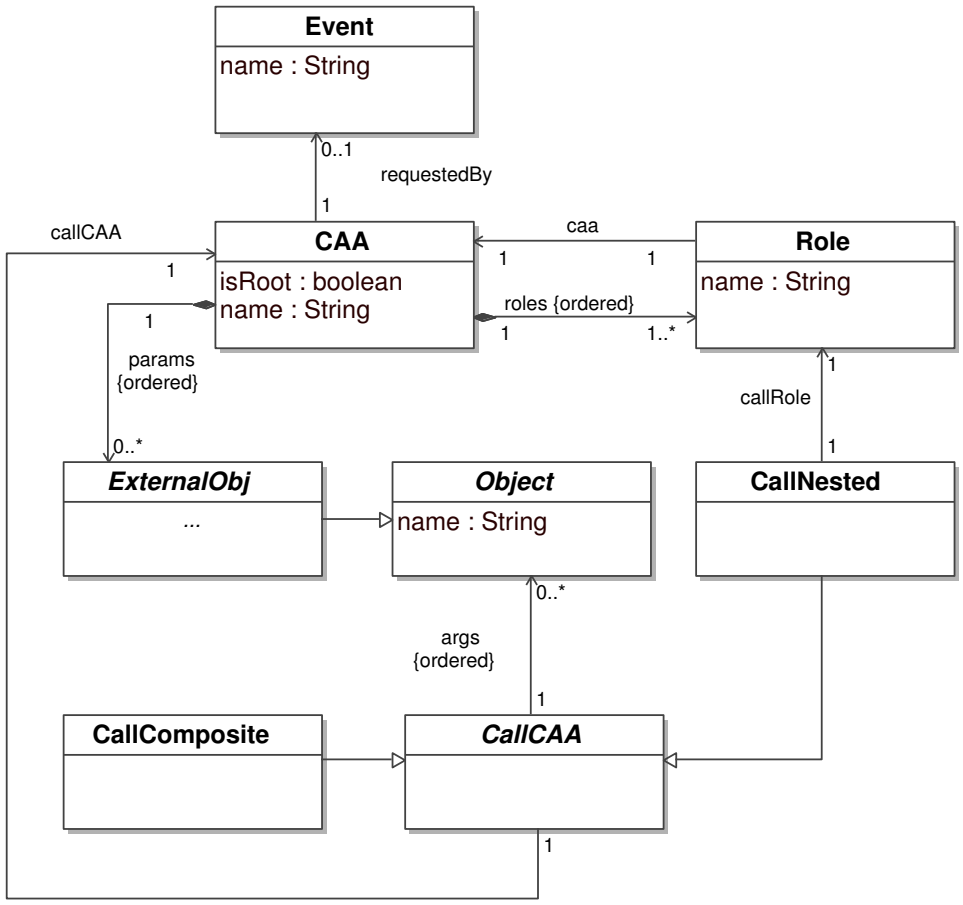

Fig. 4.22: Nested and Composite CAAs.

There are several points that apply both to nested and composite CAAs worth noting.

- In the case that an exception is raised within a CAA while an inner CAA is executing, then the inner CAA will receive a notification to abort its current execution such that those roles being involved in its execution can join with other peer roles of the enclosing CAA to deal with the raised exception. This means that the adopted model is the pre-emptive CAA scheme described in [RRS+97, XRR98].
- In the case that an inner CAA signals an exception, i.e. its outcome is either *Exceptional, Aborted* or *Failed*, every role involved in the calling of the inner CAA (it will be only one in the case of having called a composite CAA, and many in the case of a nested CAA) propagates the signaled exception to all its peer roles to start the recovery mechanism.
- Each participant playing a particular role within a CAA can only enter into one inner CAA at a time.
- A CAA may terminate only after all its inner CAAs have terminated.

```
context CAA inv Uniqueness:
        CAA.allInstances()->forAll(caa1,caa2|caa1<>caa2
                                implies caa1.name<>caa2.name)

context CAA inv eventOnlyForRootCAA:
        if (self.isRoot) then
                not(self.requestedBy.oclIsUndefined())
        else
                self.requestedBy.oclIsUndefined()
        endif


context Role inv nonSelfReference:
        Role.allInstances()->forAll(r:Role|
                (let
                        calls:Sequence(CallCAA)= r.instrs->
                          select(instr:Instruction|instr.oclIsTypeOf(CallCAA))->
                            collect(stmt| stmt.oclAsType(CallCAA))
                in
                        calls->forAll(c:CallCAA|c.callCAA <> r.caa)
                )
        )

context CallCAA inv CheckTypesOnParams:
        let
                args: Sequence(Object) = self.args->asSequence(),
                prms: Sequence(ExternalObj) = self.callCAA.params->asSequence()
        in
                args->collect(e| e.type) = prms->collect(e| e.type) and
                args->collect(e| e.type.name) = prms->collect(e| e.type.name)
```

Fig. 4.23: OCL invariants for ensuring structural consistency among CAAs.

4.2.2 Extensions

Since the CaaFWrk was conceived, several extensions were proposed to make it workable in different contexts, e.g. client/server paradigm [RE01, RPZ03], or simply deal with issues that were not initially considered. In the following, the extensions to the original CaaFWrk that are considered within the context of this thesis are presented. These extensions not only enhance the application of the conceptual framework when designing dependable software systems, but also make its use as a semantic domain in the definition of the DT4BP modelling language more suitable. It is worth mentioning that time-related extensions are addressed in a separate section (see Section 4.3) as time is one of the main dimensions of interest considered in this thesis.

4.2.2.1 Non-reversible external objects

Originally, the CaaFWrk assumes that every external object provides its own error recovery mechanism [XRR+95]. In this manner, when a CAA has to abort, i.e. to perform BER - see Section 4.2.1.4, it may rely on the support provided by each external to undo the effects of its execution. This recovery support given for each external is considered as part of the transactional support on which a CAA relies to achieve the ACID properties over the external objects it accesses.

However, sometimes the designer/programmer may want to or may have to use an external object that does not provide any recovery mechanism due to cost or physical constraints [AL81, pp. 146-149]. Designers or programmers then have to provide tailored hand-made recovery mechanisms for this kind of external object. Hence, depending on whether an external object

has or does not have its own recovery mechanism it can be categorised as *AutoRecoverable (AR)* if it does, or *ManuallyRecoverable (MR)* if it does not. Figure 4.24 shows the part of the meta-model that formalises these concepts. Both AR objects, captured by the class *Autorecoverable*, and MR objects, captured by the class *ManuallyRecoverable*, are kinds of external objects and thus a reason why these classes extend from class *ExternalObj*. This means that the parameters of a CAA may be both AR or MR objects.

The hand-made recovery designed for each MR object has to be placed within the BER of the CAA. The BER performance is divided into the activities designed by the programmer to undo the effects and those provided by each AR object. To allow the BER of a CAA to be split into two parts, i.e. one concerned with the recovery of MR objects, and other for the AR objects -this automatically supported, the notion of *Compensator* is introduced. A compensator is a kind of handler that contains the specific recovery actions needed for dealing with MR objects such that the CAA is considered as having been restored to its initial state in case of an abortion. These recovery activities are assumed to affect all the participants involved in the execution of the CAA, as the cooperation of other peer participants may be required to undo the effects of the operations having been executed over the different AR objects. Hence, a compensator, cpm_i, can be seen as a kind of handler that is made of several, i.e. one for each role r_j, $j = 1..n$ the CAA has) cooperative handlers cmp_{hnd_j}. Thus, a compensator is a kind of cooperative handler (thus of the class *Compensator* and extending from *CooperativeH*) that is meant only to recover MR objects, in the case when the CAA has to abort. This points out that every cmp_{hnd_j} (of type *Compensator* is only allowed to produce an *Aborted* outcome. The OCL invariant shown in Figure 4.25 checks this conditions.

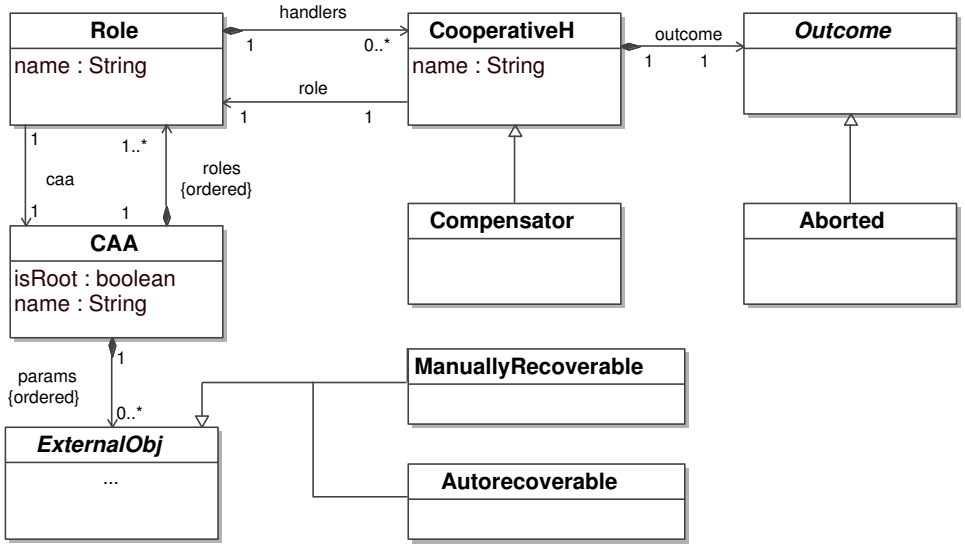

Fig. 4.24: AR and MR objects, and compensators for recovering MR objects.

```
context Compensator inv onlyAbortedOutcome:
        Compensator.allInstances()->forAll(cmp|cmp.outcome.oclIsTypeOf(Aborted))
```

Fig. 4.25: A compensator may only return an *Aborted* outcome.

It is worth mentioning that a compensator may be required in the case when a CAA (e.g. CAA_1

in Figure 4.22) encloses a composite CAA, e.g. CAA_3. This is due to the fact, that when the CAA, e.g. CAA_1, needs to abort, special activities that may be required to undo the effects of the composite CAA, e.g. CAA_3, once it has committed. These special activities enclosed within the compensator may include a second calling to the composite CAA, but this time for undoing the effects of its former execution.

4.2.2.2 Fixed and potential participants

In [RE01], Romanovsky and Ezhilchelvan introduced the new "conversation" [Ran75] scheme in order to bring flexibility to the way in which participants can join or enter into the same conversation. The authors come out with the notions of *fixed* and *potential* participants. A *fixed* participant, is defined as a participant that may *invite* other participants to join to a conversation that is currently underway. Those participants that may join a conversation upon request are defined as *potential* participants. Within this scheme, it is assumed that run-time support provides the operations that allow a participant p_i (1) to request another participant p_j to join the conversation, and (2) to register which conversations it is willing to enter upon invitation. This new scheme allows (potential) participants to join a conversation "on-the-fly", without the need of fixing which participants are required to join a particular conversation at modelling time.

Inspired by this new scheme for the conversation paradigm, the notion of *participant allocation policy* is introduced into the CaaFWrk as a flexible means for selecting the participants that may play a particular CAA. However, there are certain points that deserve more explanation before going further. First, it must be noted that any allocation policy applies only to the call of *composite* CAAs as it is (1) the only kind of CAA for which its request can be modelled within a CAA design (it is a duty of the environment to provide the participants in charge of playing the *root* CAA), and (2) there exists freedom in the participants that may play its roles (a *nested* CAAs is played by a subset of participants being engaged in the execution of the enclosing CAA). Second, the willingness of a participant in playing certain roles of different CAAs is modelled by the association named *play* (see Figure 4.26), which specifies what are the roles a particular participant may play at run-time. As already mentioned, at run-time a participant may only play a role a time. The notion of *Availability* is used to capture whether a participant is already engaged or not in the execution of a role.

In this manner, when calling a *composite* CAA, achieved by using the instruction *CallComposite* (see Figure 4.26), the modeller must specify what participants are required to play such a CAA. There exist four different kinds of *participant allocation policies*:

1. *Static*: used to select a particular participant among the pool of available *potential* participants,
2. *Dynamic*: used to select a subset of participants among all the pool of available *potential* participants. This subset is defined for all the participants that satisfy the first-order logic formula (written in OCL) specified by the allocation policy and captured by the attribute *predicate*,
3. *Reference:* used when the participant to be used must be the same used in the execution of a formerly called composite CAA. The reference to this already used participant is recorded by a variable of type *ParticipantVar* created when the former call was made. Tthis is what the composite relationship named *var* allows the modeller to capture when calling a composite CAA), and

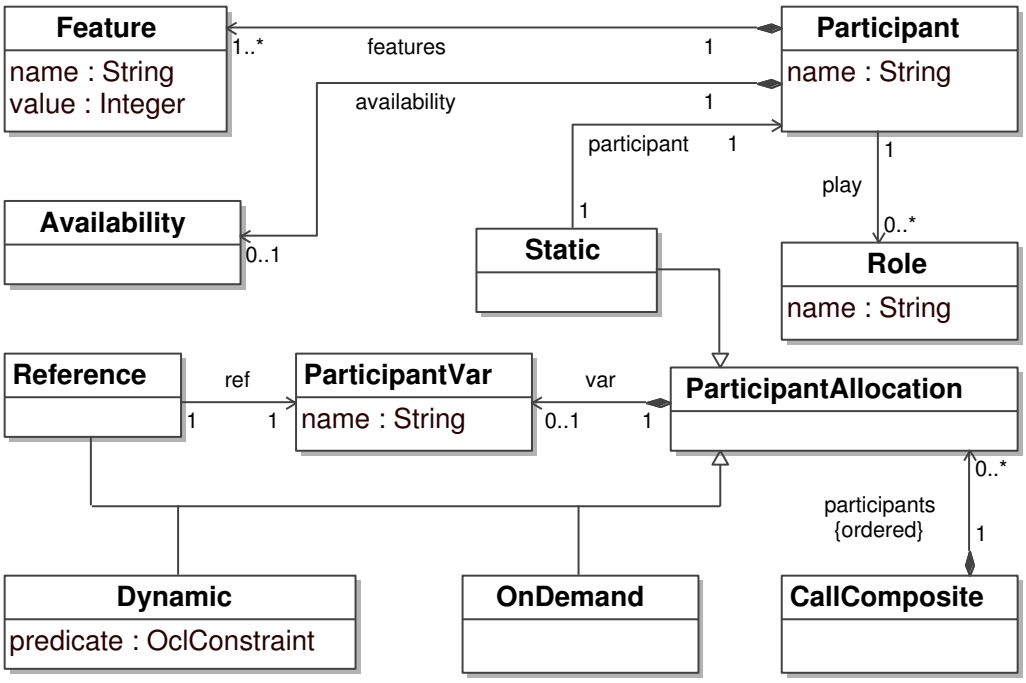

Fig. 4.26: Allocation of participants upon calling a composite CAA.

4. *Dynamic:* used to create a new participant "on-the-fly", which is different from any other participant defined at modelling time. This allocation policy allows the modeller to create participants at run-time for playing a particular CAA. It is assumed that once the CAA is completed, the participant is no longer available for re-use, unless a reference to it has been saved. In other words, there exists a variable of type *ParticipantVar* that saves a reference to the participant.

4.2.2.3 Split and Spawn instructions

Vachon in [Vac00] proposes to enlarge the set of instructions to a role by including the notion of *Spawn*. The aim of this instruction is to allow a role to open a new branch of execution, i.e. thread or process, such that the role may execute its internal instructions in a concurrent manner. It is assumed that a role ends its execution once all the spawned execution branches have also ended. Adhering to the idea of allowing a role perform its internal instructions in a concurrent manner, the notion of *Spawn* as well as the notion of *Split* is introduced in the allowed set of instructions to a role. While the instruction *Spawn* (concrete syntax **spawn{**$instr_1, ..., instr_n$**}**) is meant to create new branches, i.e. one for each $instr_i$, that will execute in a concurrent manner not only amongst themselves, but also with the current branch used to start them. Conversely, the instruction *Split* (concrete syntax **split{**$instr_1, ..., instr_n$**}**) allows the modeller to specify the creation of new branches, i.e. one for each $instr_i$, that will execute in a concurrent manner amongst themselves, but the branch used for their creation waits until all instructions have ended to continue its execution. The formalisation, in terms of metamodelling, of these two new control-kind of instructions is shown in Figure 4.27.

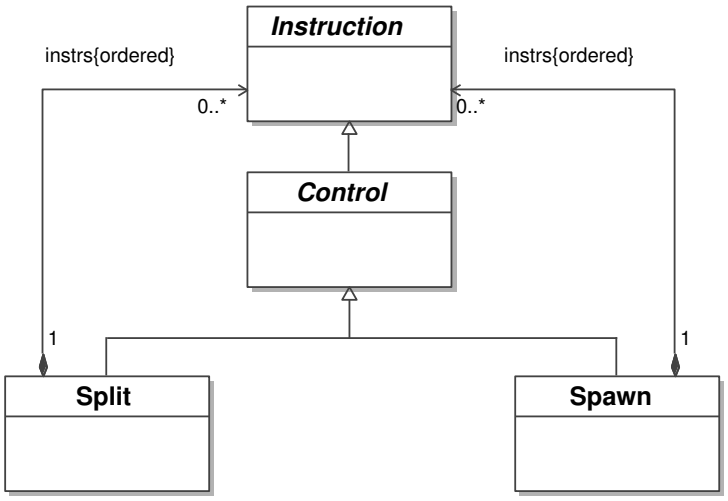

Fig. 4.27: Split and Spawn instructions.

4.2.2.4 Local handling

Collaboration is a central concept within the CaaFWrk. It is used not only during the normal execution of a CAA when roles collaborate between themselves to achieve the CAA's goal, but also during the recovery phase. As explained in Section 3, once (at a minimum) one exception is raised during the execution of the CAA roles, a recovery mechanism is started. This recovery mechanism involves all the current participants being engaged in the execution of the CAA. Hence, as soon as an exception is raised within a role, all the other peer roles are (1) notified about the exception and interrupt their current execution, and then (2) join the recovery phase.

It may be the case that a role has all the required information to deal with a raised exception during its execution. In that case, it is not only unnecessary to interrupt the other peer roles, but also a disadvantage with respect to the time it takes the CAA to complete its execution[8]. The proposal then is to allow roles to deal with an exception in a local manner, before notifying its peer roles of the exception. A *local handler* (formalised by the class *LocalH* in Figure 4.28) can be created to deal with one or more exceptions (the association *handles* captures these exceptions) that may be raised during the execution of certain instructions enclosed within a role (the composite relationship *intrs* of the class *Role*). The ordered sequence of instructions to be executed by a local handler are those captured by the composite relationship *instrs* owned by the abstract class *Handler* from which *LocalH* extends.

4.3 The Timed-CaaFWrk

The CaaFWrk has been introduced as a design method to structure the software system activities to meet the user requirements (specifications). Concurrent object-oriented software systems with high levels of dependability are mainly the kind of software system for which the CaaFWrk was conceived to be used as design method.

[8] Interrupting the execution of all the peer roles, and then restarting the execution of their respective cooperative handlers introduces an extra overhead in the runtime system that may not necessarily be negligible in contexts where every second counts, e.g. real-time systems.

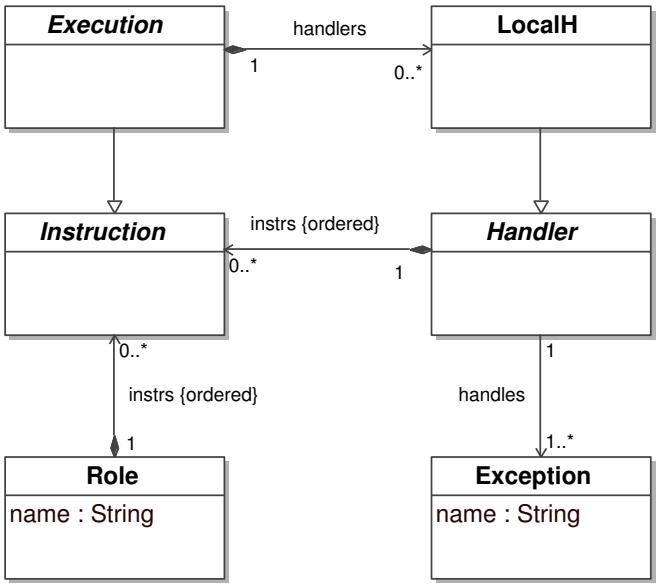

Fig. 4.28: Handling exceptions within a role.

Real-time software systems are either inherently or imposed concurrent and very often have dependability requirements ([BW01], pages 7-12). In the context of this thesis a real-time software system is considered as a software system that has within its requirements specification at least one time requirement. A time requirement is one that falls into one of the following categories [XR97]:

- *Event* timing constraints:
  - if event e_1 occurs, event e_2 must occur within time t
  - no two consecutive events must occur within time t
  - predicate C must remain true for at least a length of time t before process p can start
- *Sporadic* timing constraints: when event e occurs, if predicate C is true, trigger process p with deadline D
- *Periodic* timing constraints: while predicate C is true, execute process p with period T and deadline D
- *Data* timing constraints: each data has a valid time interval

Real-time software systems are an example of the type of systems to be designed using the CaaFWrk. However, the timing requirements imposed by most real-time software systems are not easily or impossible to be modelled by the original version of the CaaFWrk. Therefore, extensions to make it available for designing real-time software systems are required.

4.3.1 Fundamentals

Romanovsky et al. in [RXR99][9] were the first authors to propose time extensions for the CaaFWrk. These time-related extensions were meant to allow designers deal with time requirements belonging to the categories of *event, sporadic*, and *periodic* at the level of CAAs. For timing constraints belonging to the category of *data*, this new version of the CaaFWrk with time extensions (*Timed-CaaFWrk* from here on out) is assumed to rely on existing real-time object models like those described in [KK94] and [KS97]. This points out that the Timed-CaaFWrk was conceived to be combined, if necessary, with a real-time object model to cover all kinds of timing constraints.

4.3.1.1 Time constraints

As just discussed, time-related extensions are addressed at the level of the CAA abstraction. These time-related extensions[10] allow for the description of (1) the moment at which a CAA has to start, (2) the point in time at which the CAA is expected to be complete, (3) the minimum required and/or maximum allowed elapse time since the CAA started its execution, (4) the maximum time the CAA can spend executing, and (5) of the frequency with which the CAA must be executed. The informal concrete syntax shown in Figure 4.29 is aimed at giving an idea of how to express such time-related information over a particular CAA, *tCaa1*. Figure 4.30 shows how these timing constraints look in the classic graphical representation used to describe CAAs:

```
CAA tCaa1(formal parameters)
period(Te,Tu), start[t0,t1], finish[t2,t3] elapse [E1,E2]
   role R1{...} exec C1
   role R2{...} exec C2
   role R3{...} exec C3
```

Fig. 4.29: Informal concrete syntax to model time constraints over a CAA and its roles.

This indicates that the CAA named *tCaa1* has to: (1) start[11] between times t_0 and t_1, (2) finish between t_2 and t_3, (3) complete its activities in more than E_1, but in less than E_2 time units, and (4) be executed every T_e time units, its period, until the point in time T_u is reached. Since the CAA activities are performed by its roles, it makes sense to assign maximum execution times (represented as C) on each role. In this example, R_1,R_2 and R_3 have C_1, C_2 and C_3 time units as maximum to perform their activities, respectively.

Please note, t_0, t_1, t_2, t_3, T_e and T_u are relative to the point in time when *tCaa1* is requested to start its execution, i.e. request time t_{rq}), whereas E_1 and E_2 are relative to the point in time when *tCaa1* starts its actual execution, i.e. release time t_{rl}. In theory, when no delay of the CAA start is specified, i.e. t_0 is not specified, no difference between t_{rq} and t_{rl} should exist . In practice, however, the difference between t_{rq} and t_{rl} depends on the operations of the underlying layers, e.g. scheduler, middleware, O.S., needed to effectivly the execute the CAA once it is requested. Obviously, the main goal is to make the difference between t_{rq} and t_{rl} negligible.

[9] Similar and extended information can be found in [RXR98, BRR+98].

[10] Due to variations between the proposed extensions reported in [RXR99, RXR98] and [BRR+98], an unified and consolidated view of these extensions is considered within this thesis.

[11] A CAA is considered to have started its execution once all the participants have started playing their respective roles.

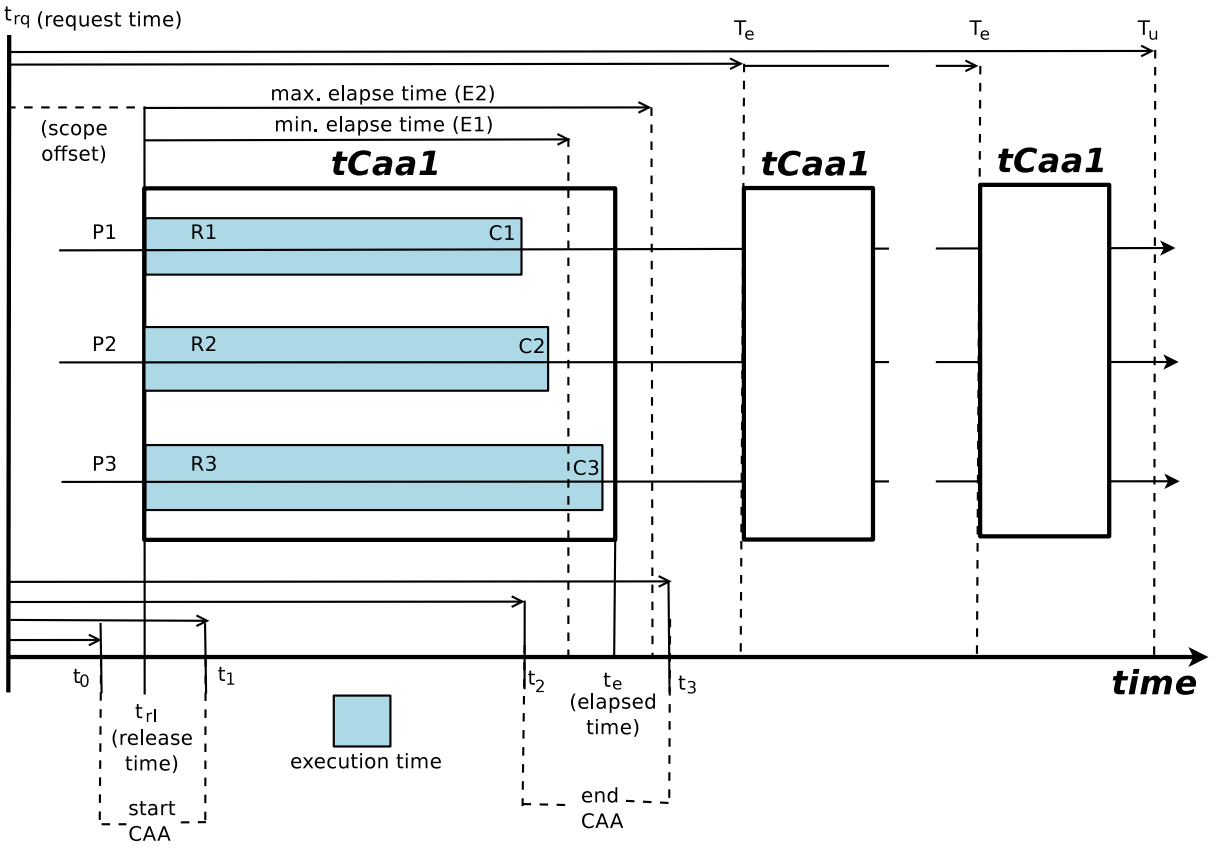

Fig. 4.30: Possible timing constraints over a CAA

As explained in Section 4.2.1.6, a CAA is started by either an event coming from the environment where it is enclosed or an explicit (nested or composite) call performed by an enclosing CAA. Hence, the point in time at which the CAA is requested to start, t_{rq}, is defined by the moment at which the event is raised, or the call is performed. Notice that the underlying software system, which would be used to manage the execution of both timed and non-timed constrained CAAs (aka the run-time support layer) is considered as part of the environment. Thus, the event that leads towards the execution of a CAA may also be generated for this run-time support layer. A real-time clock that generates an event due to having reached certain *wall clock time*[12] is an example of an event triggered by this run-time support layer. In this case, it is up to the run-time support layer to create the required participants to play the CAA's roles, as well as to remove/release them once the CAA has completed its execution. Obviously, this layer has to create and remove such participants in order to start and end the CAA execution in accordance with the specified timed constraints on the CAA.

The formalisation of the time-related extensions is shown in Figure *TimedCAA-MM-timedCAA*. This part of the meta-model shows that the composite relationships *delay*, *deadline*, and *elapse* are meant to allow the modeller to capture the time constraints related to the start (i.e. t_0, t_1), finish (i.e. t_2, t_3) and elapse (i.e. E_1, E_2) times of the CAA. The notions of *start, finish* and *elapse* are captured by the classess *StartT, FinishT* and *ElapseT*, respectively. These classes extend from the abstract class *TimeRange*. The attributes *min* and *max* of type *TimeExp* (i.e.

[12] Time as seen in the physical world

an abstract data type aimed at capturing valid time values) owned by the class *TimeRange* allow for modelling the minimum and maximum boundaries for each of the time constraints that can be held by a CAA.

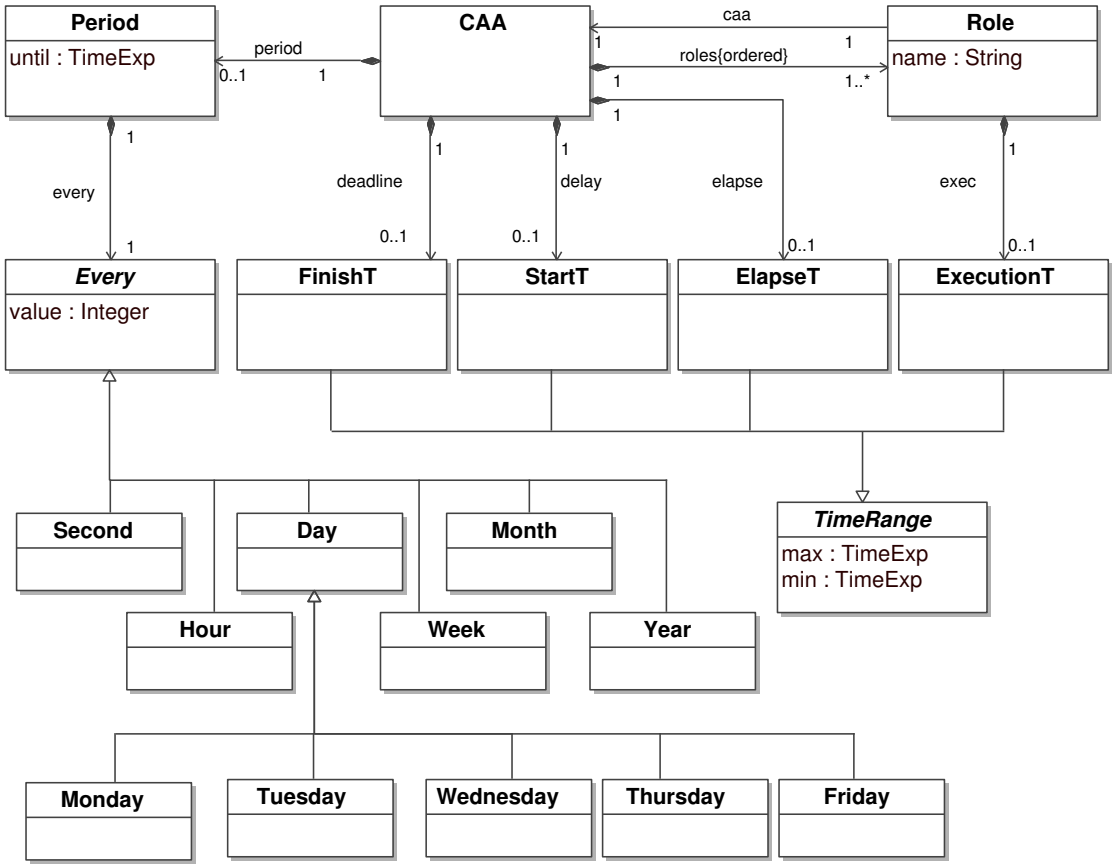

Fig. 4.31: Time constraints over a CAA and its roles.

Nothing forbids a CAA to have its *start, finish* and *elapse* times constrained, and be also *periodic*. As shown in Figure 4.29, the period of a CAA is defined by the two values T_e and T_u. The value T_e determines how often the CAA has to be executed. Whereas the value T_u defines the time elapsed since the last periodic call of the CAA. These concepts are captured in the meta-model of Figure 4.31 by the classes *Period* and *Every*, and the composite relationships of the same name. The composite relationship owned by the class *CAA* captures whether a CAA is periodic or not. Whereas the composite relationship *every* captures the information related to T_e. In fact, the attributes *until* in class *Period*, and *value* in class *Every* capture the values T_e and T_u, respectively. It must be noted that *until* is expected to own a time expression (explaining why it is of type *TimeExp*), whereas *value* is simply an integer that together with its type of class, i.e. *Second, Day, Month, etc.*, define how often the CAA must be executed.

The remaining time constraint to be fomalised is the one regarding the *execution time* of a role. This concept is captured by the class *ExecutionT*. In the original work, the authors only consider the maximum allowed execution time as a constraint to be set over a role. However, in order to be more complete and general, a lower bound regarding the role execution time could be set. Following this general and complete view, the meta-model allows the modeller to constrain both the minimum and maximum execution time of a role. That is the reason why the class

ExecutionT extends from *TimeRange*.

Obviously, the time constraints that may be set over a CAA must be consistent such that at least one instance of the modelled CAA may be created. Figures 4.32 and 4.33 show the OCL conditions that ensure the values held by the above-mentioned time constraints are such that at least one CAA instance that meets them. Such OCL conditions are specified over the assumption that all time-related values are in the same time unit[13].

```
context TimeRange inv ConsistentRange:
  (self.min > 0 or self.max > 0) and
  (if(self.min > 0 and self.max > 0) then
        self.min <= self.max
  else  true  endif)

context CAA inv ConsistentFinish:
  if(self.deadline.max > 0) then
        if(self.delay.min > 0) then
                if(self.elapse.min > 0) then
                        self.delay.min + self.elapse.min <=  self.deadline.max
                else
                        self.delay.min <=   self.deadline.max
                endif
        else
                if(self.elapse.min > 0) then
                        self.elapse.min <=  self.deadline.max
                else true endif
        endif
  else  true  endif

context CAA inv ConsistentPeriod:
  if(self.period.every.value > 0) then
        if(self.delay.min > 0) then
                if(self.elapse.min > 0) then
                        self.delay.min + self.elapse.min <=  self.period.every.value
                else
                        self.delay.min <=  self.period.every.value
                endif
        else
                if(self.elapse.min > 0) then
                        self.elapse.min <=  self.period.every.value
                else true endif
        endif
  else  true  endif

context CAA inv ConsistentFinishAndPeriod:
  if(self.deadline.max > 0 and self.period.every.value > 0) then
        self.deadline.max <= self.period.every.value
  else  true  endif
```

Fig. 4.32: Consistency between time constraints set over a CAA.

The invariant *ConsistentRange* is meant to ensure that time ranges are always valid. In other words, this invariant checks that a minimum value is always lower than or equal to its respective maximum. Otherwise, the time range is considered inconsistent. The other invariants shown in Figure 4.32 are meant to check the consistency between time constraints set at the level of a CAA. In this manner, the invariant *ConsistentFinish* checks that the minimum guaranteed time a CAA takes to execute does not go beyond its maximum deadline (i.e. t_3 or *Finish.max*). The invariant *ConsistentPeriod* performs the same check, but with respect to the period of the CAA, as it is assumed that a CAA completes its execution before starting again. Finally, the

[13] Technically, this assumption can be implemented by performing a pre-processing of the model.

invariant *ConsistentFinishAndPeriod* checks that a periodic CAA completes its period greater or equal than the time it takes to complete its execution (i.e. t_3 or max. deadline).

```
context Role inv ConsistentRoleExecAndMaxElapse:
        if(self.exec.min > 0 and self.caa.elapse.max > 0) then

                self.exec.min < self.caa.elapse.max
        else true endif

context Role inv ConsistentRoleExecAndMaxFinish:
        if(self.exec.min > 0 and self.caa.deadline.max > 0) then

                self.exec.min < self.caa.deadline.max
        else true endif

context Role inv ConsistentRoleExecAndPeriod:
        if(self.exec.min > 0 and self.caa.period.every.value > 0) then

                self.exec.min < self.caa.period.every.value
        else true endif
```

Fig. 4.33: Consistency between time constraints set over a CAA and its roles.

The invariants shown in Figure 4.33 check the consistency of the constraints regarding the execution time of a role with respect to those time constraints being set over the CAA enclosing the role. Hence, the invariant *ConsistentRoleExecAndMaxElapse* ensures that the minimum execution time of a role does not go beyond the maximum allowed elapse time of the CAA, otherwise it is always impossible to complete the CAA execution on time. Invariants *ConsistentRoleExecAndMaxFinish* and *ConsistentRoleExecAndPeriod* perform similar checks, but regarding the maximum deadline of the CAA, and its period, respectively.

4.3.1.2 Time-related exceptions

According to the previous information, a CAA with time constraints, a timed-CAA, is a simple way to describe the launching, ending and duration (both in units of time and processing) of a group of jointly concurrent activities. Any violation of these timing constraints is seen as a time-related exception. The time-related exceptions that may take place during the execution of a timed-CAA are the following:

- *Late starting of the CAA:* Occurs when at least one participant does not start its role before the maximum allowed delay (t_1 in Figure 4.31) allowed to start a CAA. The raised exception is $MaxCAAStartT$;
- *Early finish of the CAA:* Occurs when all the participants have ended their roles before the minimum deadline (t_2 in Figure 4.31). The raised exception is $MinCAAFinishT$;
- *Late ending of the CAA:* Occurs when at least one participant has not completed its role before either (1) the maximum deadline (t_3 in Figure 4.31), or in the event the CAA is periodic, (2) the period of the CAA (T_e in Figure 4.31). The raised exceptions are $MaxCAAFinishT$ and $CAAPeriodT$, respectively;
- *Short elapse time:* Occurs when all the participants have ended their roles before the minimum required elapse time (E_1 in Figure 4.31). The raised exception is $MinCAAElapseT$;

- *Long elapse time:* Occurs when at least one participant has not completed ts role before the maximum allowed elapse time (i.e. E_2 in Figure 4.31). The raised exception is *MaxCAAElapseT*;
- *Short role execution time:* Occurs when the participant completes the execution of its role using less than the minimum required execution time (i.e. C_{min})). The raised exception is *MinRoleExecT*;
- *Long role execution time:* Occurs when the participant exceeds the maximum allowed execution time (i.e. C_{max}) for executing its role. The raised exception is *MaxRoleExecT*;

It must be noticed that the time constraint related to the minimum required delay (i.e. t_0) is assumed as always being met. Hence, as this kind of time constraint cannot be missed, no exception is required to signal this type of time constraint violation.

In the case that a time-related exception is raised, and following the same principles of the original CaaFWrk, a recovery phase is started to handle the exceptional situation. The kind of recovery, i.e. FER or BER, to be performed depends on the time-related exception being raised. The time-related exception *MaxCAAStartT* signals the unsuccessful start of the CAA. In this case the CAA must return the *Aborted* outcome of its enclosing context since the effects are the same as having applied a BER. In other words, the fact that the CAA did not execute is equivalent to having executed and then applied BER. Notice that the *Aborted* outcome is propagated to all the participants that should have entered into the CAA.

The time-related exceptions *MaxCAAStartT* indicate that the CAA has run out of time to complete its execution. Since it is too late to perform any complex activity, the recovery phase is meant to abort the CAA execution by performing a BER. It is assumed that aborting a CAA is a much faster process that any FER.

The time-related exceptions *MinCAAFinishT* and *MinCAAElapseT* signify the early completion of the CAA. Since there is still time to perform certain activities, it is allowable to use an FER to deal with any of these time-related exceptions.

Despite the exception *MaxCAAElapseT* (denoting a late completion of the CAA) it is also allowable to use an FER to handle it. The hypothesis of using an FER to handle the *MaxCAAElapseT* exception is provided on the assumption that the maximum allowed elapse time is a means to detect a timing problem in the execution of the CAA earlier than the "hard" deadline t_3 that the CAA is not allowed to overpass. Therefore, the elapse time can be considered as an internal check for controlling the temporal evolution of the CAA, giving the modeller an opportunity to put into place specific recovery activities (FER) to fix the problem in time as defined by the time constraint t_3.

The time-related exceptions *MinRoleExecT* and *MaxRoleExecT* indicate problems regarding the execution time of a role. Relying on the notion of local handling introduced in Section 4.2.2 as part of the proposed extensions to the CaaFWrk, a recovery phase can be placed within the role to deal with any of these exceptions such that the other peer roles are unaware that an exception and its recovery have taken place. It is worth noting, that in the case that these exceptions are not handled within the context of role where they may be raised, their propagation to the other peer roles enclosed within the same CAA leads to the abortion of the CAA. Hence, the time-related exceptions *MinRoleExecT* and *MaxRoleExecT* are allowed to be handled by a local handler, only. The OCL invariant shown in Figure 4.35 ensures this condition. Notice that this OCL invariant relies on the part of the meta-model being shown in Figure 4.34, which formalises the time-related exceptions previously introduced. The time-related exceptions

MaxCAAStartT, *MinCAAFinishT* and *MinCAAElapseT* are not part of the formalisation since they are not allowed to be handled; their flagging leads the abortion of the CAA.

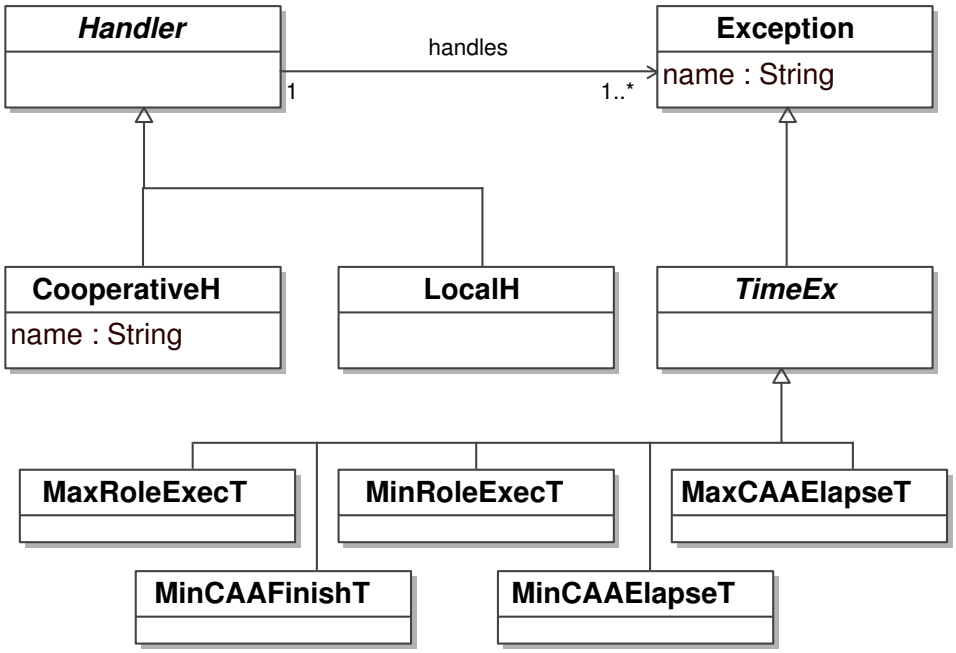

Fig. 4.34: Time-related exceptions for which may exist a handler.

```
context Handler inv localHandlerForMaxRoleExecT:
        Handler.allInstances()->
            collect(h|h.handles.oclIsTypeOf(MaxRoleExecT))->
                forAll(h|h.oclIsTypeOf(LocalH))

context Handler inv localHandlerForMinRoleExecT:
        Handler.allInstances()->
            collect(h|h.handles.oclIsTypeOf(MinRoleExecT))->
                forAll(h|h.oclIsTypeOf(LocalH))
```

Fig. 4.35: Consistency between time constraints set over a CAA and its roles.

4.3.1.3 Concurrent time-related and value-related exceptions

Some of the time-related exceptions, i.e. an exception that signals the absence of a time constraint, presented in the previous section may be raised concurrently with one or more value-related exception, i.e. an exception that notifies a logical problem. In the following discussion, the different cases that may lead to concurrent time and value-related exceptions are analysed. As a vehicle to support this analysis and ease its explanation, Figure 4.36 shows the exception graph used in Section 4.2.1.4 along with all possible time-related exceptions that may also occur during the execution of a timed-CAA. All time-related exceptions hang on the universal exception. This means that every time-related exception can be raised during the execution of a timed-CAA, but not in a concurrent manner with any other time-related exception.

The first time-related exception to be considered is *MaxCAAStartT*. Since this exception flags

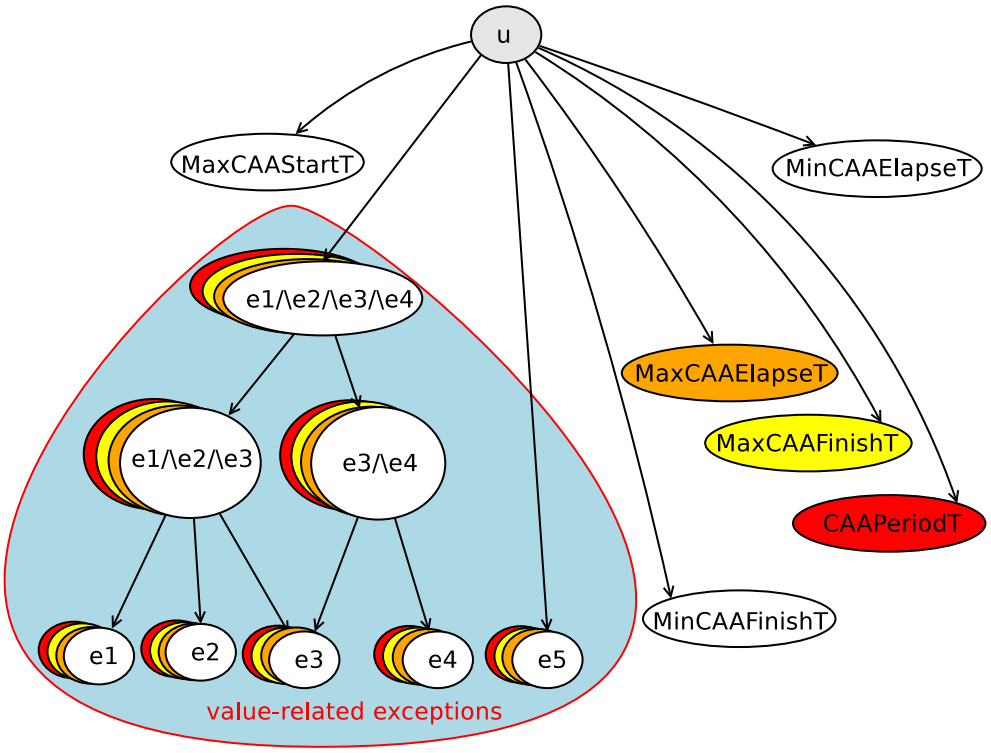

Fig. 4.36: Exception graph with all possible time-related exceptions.

the late start of the CAA, it may raised in an isolated manner as none of the CAA's roles have been started by its respective participant. Other time-related exceptions that may only occur in an isolated manner are $MinCAAElapseT$ and $MinCAAFinishT$ as all the participants have ended their roles completing the execution of the CAA, however earlier than expected. In the unusual, but still possible situation in which the time constraints t_2, the minimum required deadline and E_1, the minimum required elapse time specify the same values, $t_2 = E_1$, it is assumed that only the exception $MinCAAFinishT$ is raised as this exception is considered as having a higher precedence than $MinCAAElapseT$.

The time-related exceptions $MaxCAAElapseT$, $MaxCAAFinishT$, and $CAAPeriodT$ might be raised concurrently with one or more value-related exceptions. The overlapping colored nodes depicted in Figure 4.34 show the potential concurrent occurence of these time-related exceptions with any of the value-related exceptions $e_{1..5}$, $e_1 \wedge e_2 \wedge e_3$, $e_3 \wedge e_4$, and $e_1 \wedge e_2 \wedge e_3 \wedge e_4$. However, only one time-related exception can be raised concurrently with one or more value-related exceptions. In the unusual, but still conceivable case where the time constraints t_3, the maximum allowed deadline, E_2, the maximum allowed elapse time, and T_e, the period of the CAA, all specify the same values, i.e. $t_3 = E_3 = T_e$, only one exception will be raised according to the following (ascendant) order of precedence: $MaxCAAElapseT < MaxCAAFinishT < CAAPeriodT$. Thus, one or more value-related exceptions can be raised concurrently with either $MaxCAAElapseT$, $MaxCAAFinishT$, or $CAAPeriodT$.

The cases in which a value-related exceptions is raised concurrently with $MaxCAAFinishT$, or $CAAPeriodT$ deserves special attention. As was explained previously, both $MaxCAAFinishT$ and $CAAPeriodT$ lead to the BER, since the CAA has run over time completing its tasks and its termination is the last opportunity to leave the software system in a consistent state. Hence, once $MaxCAAFinishT$ or $CAAPeriodT$ are raised the CAA has to be aborted, regardless of whether

these time-related exceptions were raised concurrently or not with a value-related exception. In this manner, the only time-related exception (1) that may be raised concurrently with a value-related exception, and (2) that leads to a FER process is $MaxCAAElapseT$. However, Romanovsky et al. in [RXR99] argue that time-related exceptions should not be included in the directed graph since handlers for these time-related exceptions are not suitable for concurrent value and timing exceptions. Therefore, if a time-related exception is raised concurrently with one or more value-related exceptions, then the former is not considered during the exception resolution phase. Hence, the resolution graph of a timed-CAA implicitly contains those time-related exceptions that may be handled by means of FER. Figure 4.37 shows how what the resolution graph used during this analysis should look, where implicit exceptions are depicted as grey nodes.

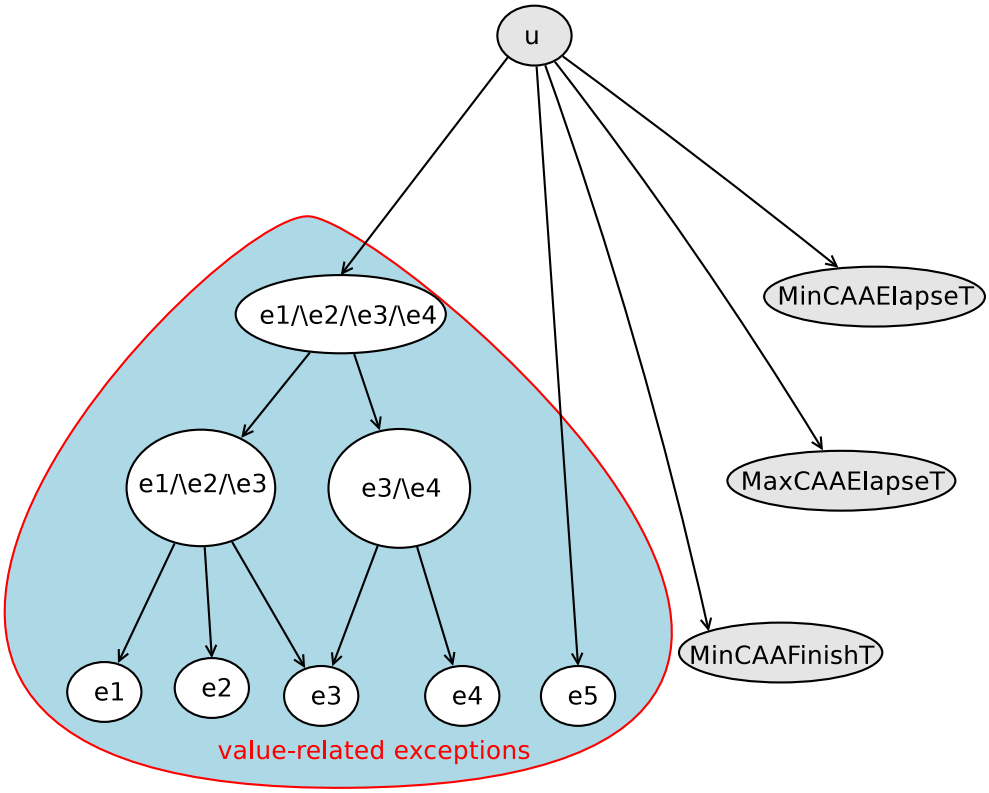

Fig. 4.37: Resolution graph of a timed-CAA.

The authors, in the work (i.e. [RXR99]) cusorily introduce two alternatives for dealing with concurrent time-related and value-related exceptions. The first alternative solution is that every node in the directed graph holds the worst case execution time for its associated handler. The exception to be handled, then, is such that its associated handler takes less or equal time than that allowed for the recovery phase[14]. The second alternative solution is to design two handlers for each value-related exception: one to be used when this value-related exception has been raised concurrently with one of the time-related exceptions, and the other to be used when no timing constraint has been violated.

[14] For example, the informal syntax "**within** T" can be used to fix the maximum duration allowed for the whole CAA recovery phase. Whereas "**exec** C" can be used to set the maximum execution time allowed by a handler.

4.3.2 Extensions

In the same manner as was done for the CaaFWrk, extensions to the Timed-CaaFWrk are proposed such that time constraints not considered thus far can be modelled. These new time constraints are not only meant to ease the capture of the user requirements. They are also to be used by the modeller as a means to have better control over the temporal evolution of CAA such that time-related errors can be detected earlier providing the opportunity to either spend more time in recovery, or a faster outcome to the CAA enclosing context.

4.3.2.1 Time constraints within a role

For a CAA with time constraints such as E_2 or t_3 it is important to recognise any problems as soon as they arise in order to start the recovery process with the maximum allowed time. Then the recovery does not miss any of these time constraints.

In the current version of the Timed-CaaFWrk, a CAA will start its recovery process before reaching the time constraints E_2 or t_3 when (1) a value-related exception has been raised, or (2) the maximum allowed execution time for any of the roles has been exceeded. Raising a value-related exception represents an incorrect functional progress of the role with respect to the expected CAA goal. Exceeding the maximum execution time allowed is a sign that the timing progress of the role was unexpected. While a value-related exception is evidence that the role was executing its instructions, the raising of the exception *MaxRoleExecT* may be due to the fact that (1) the execution time assigned to the role is very high for the instructions that it had to perform, or (2) the role gets blocked[15] either more often or longer than expected. For the first case, the only way to prevent this time-related exception from being raised is to increase the time constraint. For the second case, the role may be monitored such that when certain instructions are not executed after a certain period of time, then the role is not progressing with its execution as expected.

The proposal, then, is to let the designer include time constraints within the role. These time constraints are aimed at working as internal checkpoints that provide some idea about the progress of the tasks performed by a role. As soon as an internal time constraint is missed, the appropriate recovery phase may be called to handle the situation. This recovery phase may be either the usual cooperative handling, or it may take place within the role, without engaging the other peer roles enclosed within the same CAA. In any case, the goal is to recognise the slow progress of a role early enough so that appropriate actions can be taken that allow the issue to be resolved without violating E_2 or t_3.

Note that if a designer wants to cope with this situation using the current Timed-CaaFWrk, he has to add specific time-related instructions for measuring the progress of the role. In the case where that the role does not progress as expected, then the role has to be designed to raise an exception to allow the recovery to take place. Therefore, these time-related instructions would be in-line with the tasks to be performed by the role in the context of the CAA. However, these issues make the design much more difficult to read, interpret and maintain.

The formalisation of the proposed extension is shown in Figure 4.38. This part of the meta-model describes any executable instruction, i.e. any instruction that is sub-type of *Execution*, enclosed within a role that may have a *deadline*. This deadline represents the maximum allowed time an instruction may take to execute. Notice that for completeness, it is also allowable to constrain

[15] A role gets blocked when waiting for either (1) receiving a message, or (2) getting access to an external object.

the minimum required time of an instruction. Thus, the composite relationship *deadline* may have a time constraint (of type *ElapseT*), which captures the lower and/or upper bound of an instruction regarding its elapses time.

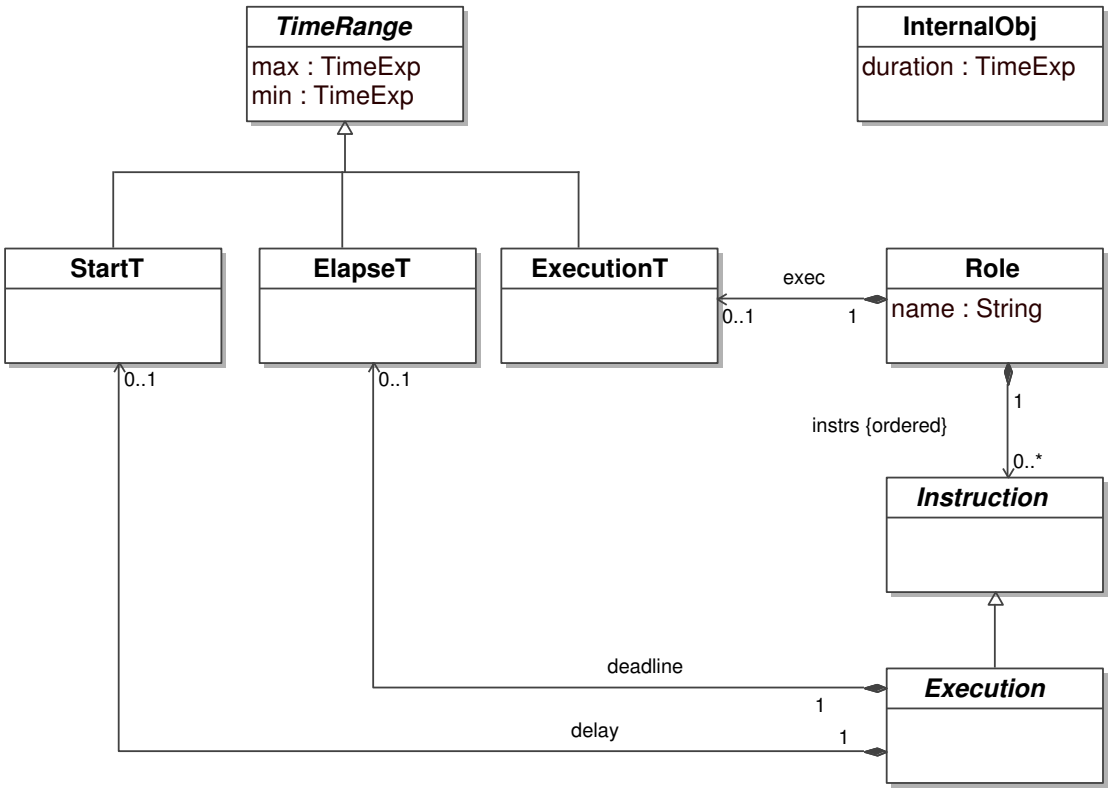

Fig. 4.38: Time constraints over instructions within a Role, and internal objects.

Other extensions are aimed at allowing the modeller to include delays within a role in the same manner as deadlines are included. Despite many programming languages with real-time support for the notion of delay in a native way, the idea is to include this time-related facility at the level of Timed-CaaFWrk to be consistent and complete with respect to the concepts being dealt with at the level of the CAA. In other words, since the notion of delay is considered at the level of the CAA, it makes sense to include this concept at the level of the role. The formalisation of the notion of *delay* at the level of a role is captured by the composite relationship of the same name. This relationship allows the modeller to specify how much time an instruction must be delayed before executing. Despite the fact that a delay is modelled as an instance of the class *StartT*, which allows for the specification of a minimum and a maximum value, it is assumed that both attributes hold the same value. The OCL shown in Figure 4.39 ensures this condition. It is worth noting that this delay is relative to either (1) the point in time at which the previous instruction completed its execution, or (2) the starting of the CAA in the case that the instruction is the first one in the ordered sequence of instructions, *instrs*, to be performed by the role.

The last proposed extension is aimed at allowing the modeller to impose a time constraint over an internal object[16]. This time constraint determines the time interval for which the internal objects's value is valid. Hence, the time constraint works as a kind of expiration timestamp for

[16] A time constraint that is closed to this approach is briefly mentioned in [BRR+98].

```
context Execution inv sameValuesForMinMaxAttr:
        Execution.allInstances()->forAll(e|e.delay->notEmpty()
              implies e.delay.min=e.delay.max)
```

Fig. 4.39: Attributes *min* and *max* have the same value when modelling an instruction delay.

the value held in the object. The attribute *duration* of the class *InternalObj* is the means used to capture this time constraint.

An instruction that misses its *deadline* (regardless of whether it is the lower or upper bound), or tries to access an internal object beyond its duration time leads to a time-related exception within the role. (The *delay* of an instruction is assumed to be always satisfied.) These time-related exceptions are (shown in Figure 4.40.):

- *MinInstrExecElapseT*: this exception is raised when an instruction completes its execution before the minimum deadline. This exception may be handled within the role where it takes place. Otherwise, its propagation leads to the CAA being aborted.
- *MaxInstrExecElapseT*: this exception is raised when an instruction does not complete its execution before the maximum deadline. This exception may be handled within the role where it occurs. Otherwise, its propagation leads to the CAA being aborted.
- *DataExpired*: this exception is raised when there is an attempt to read an internal object once its duration has expired. This exception may be handled within the role where it takes place. Otherwise, its propagation leads to the CAA being aborted.

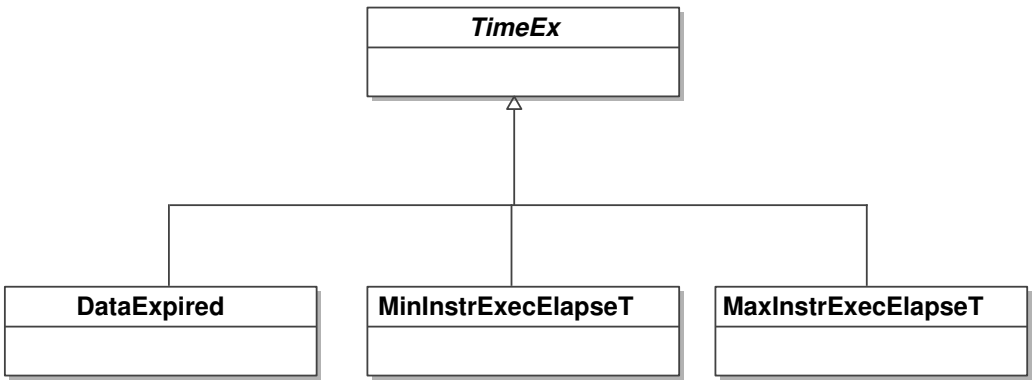

Fig. 4.40: Time-related exceptions that may be raised within a role.

4.4 The Timed-CAA-DRIP implementation framework

Within the context of this thesis, a *conceptual framework* is considered as a set of concepts with a particular semantics. The Timed-CaaFWrk is a conceptual framework aimed to support the design of dependable distributed and concurrent real-time object-oriented (OO) software systems. However, since this is simply a *conceptual* framework, the implementation of every

software system that is designed by means of the Timed-CaaFWrk has to be carried out without any support and relying on the knowledge and experience of the programmers involved in such process.

In this thesis, an *implementation framework* is any kind of support that eases the programmers implementation of a particular design. The Timed-CAA-DRIP implementation framework presented in this Section is aimed at supporting programmers in the implementation of software systems being designed with the Timed-CaaFWrk. This implementation framework is composed of a collection of Java [Jav] classes and interfaces customised by the programmer to fit the software system needs. The main reasons for having chosen Java to develop the implementation framework are (1) it is considered the de-facto programming language in the research field, and (2) it is a further development of the CAA-DRIP implementation framework presented in [CGP+06], also developed in Java. The latest version of Timed-CAA-DRIP, along with examples can be downloaded from [DT410].

4.4.1 Design overview

An implementation framework providing support for the CaaFWrk must allow participants (within Timed-CAA-DRIP they are implemented as threads) to enter synchronously, perform the collaborative work specified in the roles of the CAA aimed at providing certain service, and then finish their execution in a synchronised manner. Furthermore, this expected behaviour must be achieved even in the presence of failures, for which the implementation framework must also provide support. This support is achieved by putting alternative behaviours in place when faced with certain abnormal situations. Hence, the CAA's service is still reached either as initially planned, or in a degraded version (but still good enough to continue with the execution of the software system).

The implementation framework is divided into two parts. One part is concerned with the support provided at run-time for synchronising participants upon entry into the CAA, for letting the participants playing the CAA's roles, and synchronising them upon exit from the CAA. The run-time support also includes the synchronisation between the collaborative work performed by the participants once they are enclosed within the CAA, as well as the resolution and recovery mechanisms that occur when one (or more) exceptions occur during the execution of the CAA. This run-time support has been designed and implemented as a Java library that runs on top of the *Java Runtime Environment (JRE)*. Details regarding the role of this library during execution of a software system that implements a particular CAA design are given in Section 4.4.2.

The second part is concerned with the facilities given to the programmers to use the implementation framework. Interfaces, and ways of extending the implementation framework are the means programmers must use to achieve the Java code that implements the CAA Design. These interfaces and extensible classes were designed to let programmers deal with the same concepts that appear at the design level. Hence, there exist classes named *CAA, Role, Handler* and *Compensator* such that the way of using them is similar to the manner they are employed during the design phase. Thus, the gap for reaching the actual implementation for a given CAA design is shortened with respect to implementation being developed without any kind of support. The implementation framework, from the programmer's viewpoint, is structured into eight components: *caa, caa.time, exceptions, exceptions.time, allocator, instructions, datatypes*, and *util*. These components along with the classes and interfaces belonging to them are presented in Section 4.4.3.

Figure 4.41 gives a general overview of the Timed-CAA-DRIP framework. This figure shows

how a software system, implemented according to a certain CAA design, interfaces with either the JRE (when using native Java code) or the Timed-CAA-DRIP (when using CAA's concepts).

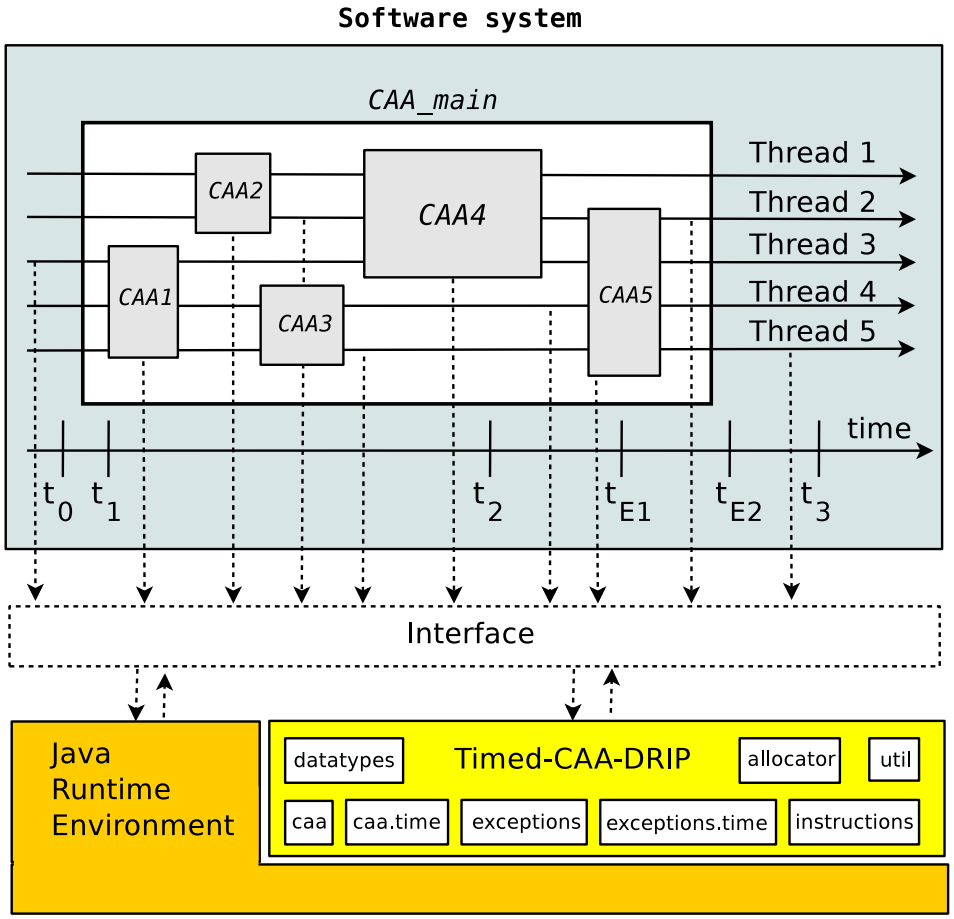

Fig. 4.41: Timed-CAA-DRIP implementation framework overview.

4.4.2 Run-time support

Once the software system has been deployed and started, the main objective of Timed-CAA-DRIP is to give run-time support to allow the CAAs composing the software system behave as expected. This section describes the internal executions performed by the implementation framework to achieve this expected behaviour. It is worth noting that the way the executions are performed depend not only on whether the CAA has time constraints or not, but also if an exception takes place in one of its roles during the execution. Hence, the description of the Timed-CAA-DRIP internal executions are divided among those required to execute a (normal or non-timed) CAA, a timed CAA and those that take place when dealing with an exception.

4.4.2.1 CAA execution

The activation of a CAA is considered as the process that begins when at least one participant enters into the CAA to play one of its roles. This process (i.e. the activation of the CAA) is

completed once all the required participants have entered into the CAA, and the latter is ready to play their respective roles.

There exist two ways of requesting the activation of a CAA: the *composite* and *nested* ways. When calling a CAA in a *composite* manner, a single participant is required to achieve both the activation and execution of this CAA, since the implementation framework will, according to the allocation policy specified, locate the necessary participants. This activation is achieved by calling the method *executeAll* of the particular CAA that wants to be executed.

Conversely, when calling a CAA in a *nested* manner, such an allocation does not take place, since the participant to be used is the same that requests the activation of the CAA. Notice that in the case of nesting, a participant requests the activation of the CAA that wants to be executed by calling the method *execute(r,caa)*, where r is the role to be played by the calling participant into the CAA of name *caa*. To succeed in the activation of this CAA, there must be as many participants requesting the activation of the CAA *caa* as roles it has.

Hereafter, the explanation is only focused on the activation and execution of a composite CAA. The Sequence Diagram shown in Figure 4.42 is used as a vehicle to support (and ease) the explanation of the internal operation performed by the implementation framework.

A participant, which is considered the client of the CAA being called, executes the method *executeAll* to request the activation of the CAA. Once the participant has executed this method, the control of the execution is passed to the Timed-CAA-DRIP framework (server side), which takes charge of the complete execution of the requested CAA. Notice that the CAA to be executed (according to the Sequence Diagram named *caa*) is assumed as properly defined by the client. Details about how to define a CAA are given in the following section.

Once the CAA has taken control of the execution it starts the activation process by executing the internal method *runCAA*. The first step within the activation process is to gather the needed participants to play each of its roles. The method *assign_participants* looks for the specified participants and then allocates them to each of the roles. Once the allocation has completed, a thread is created for each participant such that they can play their respective roles. This is achieved by the method *executeAll* of the class *Role*. This method, after having created the thread, passes control (by the method *controller*) to the *manager* of the role, which is the entity that leads its execution. A manager, then starts the execution of the role by calling the method *run*. Once this method has completed, the execution of the CAA life cycle starts. Notice that there exists a manager (named mgr_i in the Sequence Diagram) for each role (named $role_i$) to be played.

The CAA life cycle is coded in the *Manager* class as a sequence of operations that it executes after its activation. The first activity a manager executes is to synchronise itself with all other managers that are taking place within the CAA. This is done by calling the *syncBegin* method. This method gets blocked until all the managers have synchronised. Once the *syncBegin* method returns, the manager checks if the pre-condition of the role is valid by calling the method *preCondition*. If the pre-condition of the role is not satisfied, then an internal exception called *PreConditionException* is raised, leading the CAA to its recovery phase, which is described at the end of this section.

Once all the managers have checked that the roles' pre-conditions are met, the activation process of the CAA is considered as complete. Its status is changed to "start" by calling the method of the same name. Starting the CAA leads each manager to execute the role that is under its control by calling the *bodyExecute* method of the *Role* object. This method calls the method *body*, which has been overridden by the programmer with the instructions the role is expected

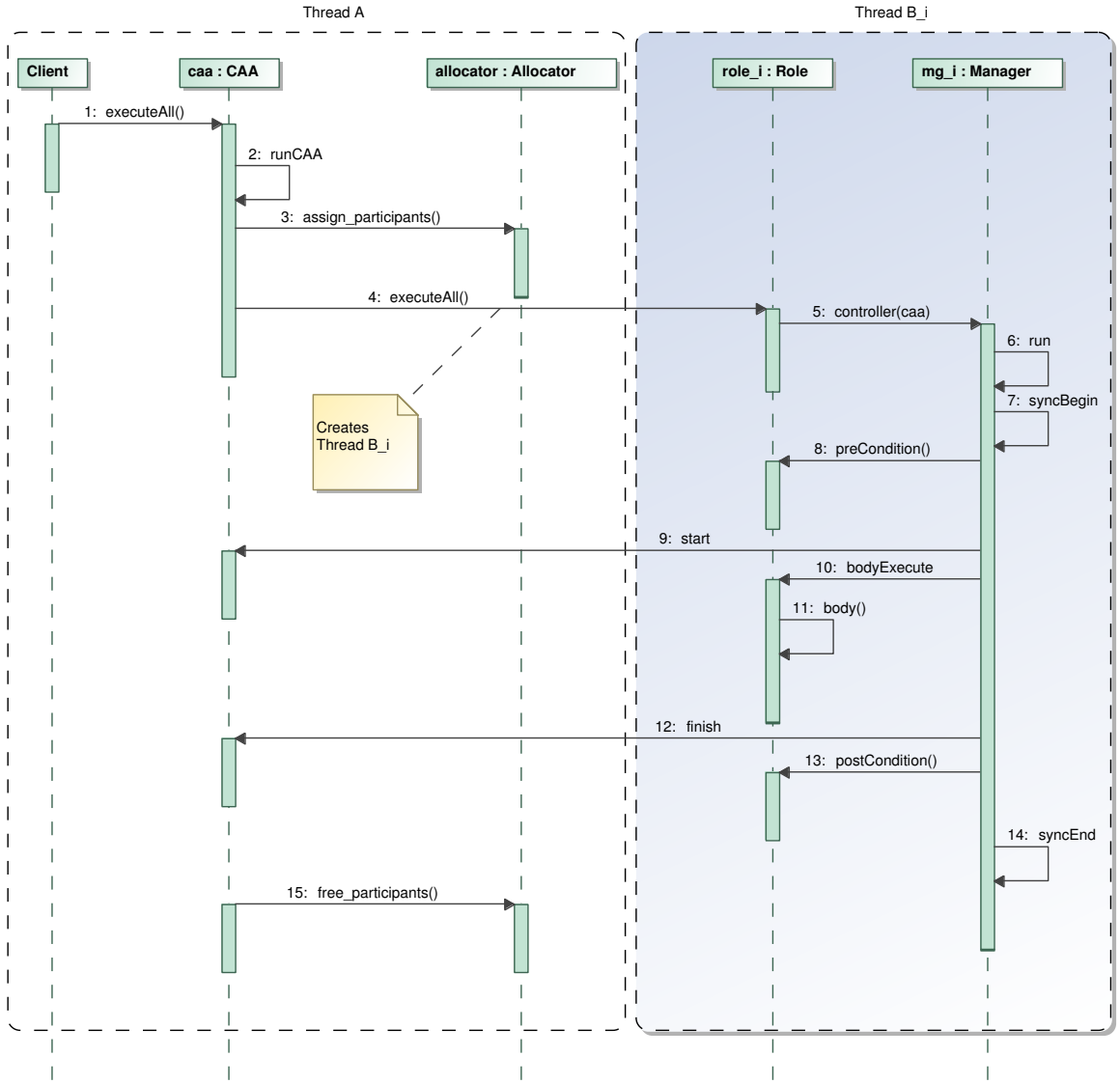

Fig. 4.42: Internal executions for starting a (non-timed) CAA.

to perform.

After all the roles have finished executing, the CAA status is changed to "finish". It only remains to check whether each role passes its post-condition or acceptance test. If the post-conditions are satisfied, then the manager, once again, synchronises with all the other managers to finish the CAA execution synchronously. Otherwise the internal exception *PostConditionException* is raised, making the CAA recovery phase start.

Managers, after having synchronised upon exit, return control to the CAA (*caa* in the Sequence Diagram). Notice that every time a manager returns control to the CAA, the thread that is used to carry out the role execution is stopped[17]. The last step is concerned with the release of the allocated participants. This is achieved by calling the method *fre_participants*.

[17] A stopped thread is eventually removed by the JRE.

4.4.2.2 Timed-CAA execution

The run-time support provided by the implementation framework for launching a timed-CAA differs not only (1) in the sequence of method calls taking place, but also (2) in the overhead introduced due to the monitoring of the time constraints that can be set over the CAA. Details about these differences are given in the following order: first, the sequence of method calls required to make a timed-CAA runnable are detailed, and then information regarding the overhead introduced by the implementation framework is given.

The Sequence Diagram shown in Figure 4.43 describes the complete sequence of method calls taking place during the execution of a Timed-CAA. This Sequence Diagram shows that the mayor difference between launching a (non-timed) CAA and a timed-CAA is in the use of the Java native interface *ScheduledExecutorService*, which is aimed at scheduling (Java) instructions not only to run after a given delay, or to execute periodically, but also to check whether their execution is completed within the expected deadlines.

After executing the method *schedule*, the sequence of method calls remains the same as for a non-timed CAA. However, the behaviour between a non-timed and a timed-CAA is differentiated by the number of threads involved in their execution. The method *schedule* is in charge of creating new threads both to activate the requested timed-CAA, and to monitor the timing constraints this CAA has to meet. Beside the thread used to activate and then execute the CAA, there may exist threads to monitor (1) the maximum allowed delay in effectively starting of the CAA (i.e. t_1 in the Figure **??** shown in Section 4.3.1.1), (2) the minimum and maximum finish time or deadlines (i.e. t_2 and t_3), (3) the minimum and maximum elapse time (i.e. E_1 and E_2), and (4) the deadline imposed by its periodicity (i.e. T_e), in case the requested timed-CAA is periodic. For each kind of timing constraint, a new thread is created. This represents an extra overhead, in terms of memory and processing power, with respect to a non-timed CAA. A performance analysis similar to the one provided in [CGP+09] is planned to be carried out to quantitatively assess the overhead introduced by monitoring the time-constraints. However, it is worth noting that such an overhead is limited, since in the worst case, there will exist four extra threads running in parallel with those threads required to execute each of the CAA's roles.

Each of these threads is aimed at monitoring a particular time constraint. Each thread will raise an internal time-related exception as soon as the watched time constraint is missed. The raising of this time-related exception leads the timed-CAA to its recovery phase. Notice that the recovery phase of a CAA is the same regardless of whether it is timed or not.

4.4.2.3 Recovery phase

The recovery phase of a CAA starts when an exception is raised during the execution of (at least) one of its roles. This points out that the CAA has already started, since a role is able to execute once all of the CAA's roles have met their respective pre-condition. It is worth recalling that an exception may be raised either by the role itself, or any of the underlying layers it relies on to perform its execution. The Sequence Diagram shown in Figure 4.44 depicts the raising of exception *ex* within the role $role_i$ as a self-message taking place in such a role.

As was explained in Section 4.2.2, a role may include a local handler for dealing with some exceptions. Hence, depending on whether there exists a local handler (first alternative), the raised exception can be handled by the role itself in a single manner. Otherwise, the exception must be handled in a cooperative manner by all the participants engaged in the execution of the CAA (second alternative). In this situation (i.e. second alternative), the role where the

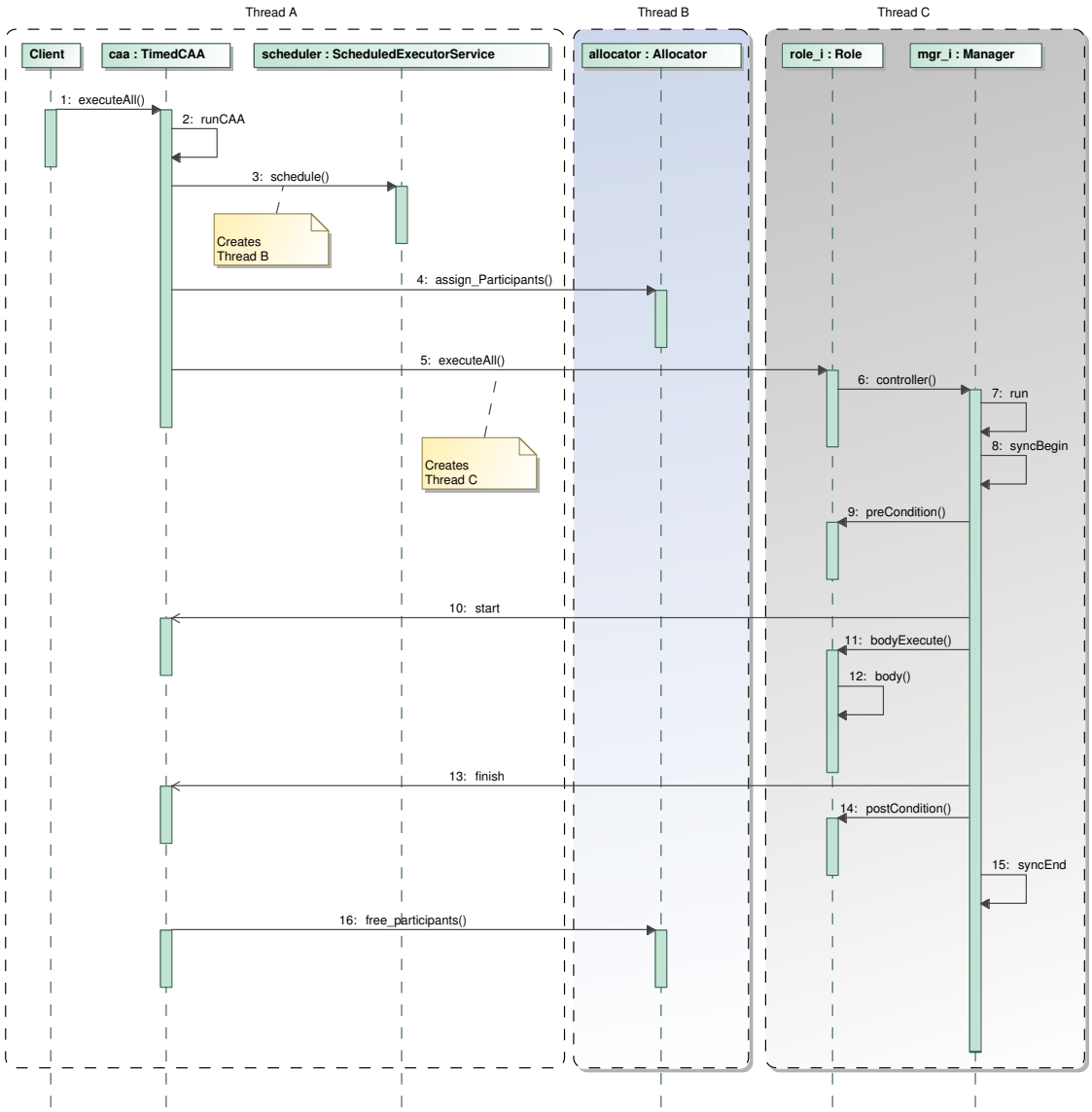

Fig. 4.43: Internal executions for starting a timed-CAA.

exception was raised, notifies its manager (named mgr_i in the Sequence Diagram). This is achieved by propagating the exception using the Java native instruction *throw*.

Once the manager is notified about an exception, it starts the resolution mechanism by executing the method *exceptionResolution*. This method includes a call to the method *notifyException* over the *leader*[18] manager. The aim of the *notifyException* method is: (1) to interrupt the execution of all the other roles enclosed within the same CAA, and (2) to determine the resolving exception to be handled. The second goal is achieved by calling the method *getResultException*. Once all the managers have been informed as to which exception is to be handled, the recovery mechanism is initiated.

[18] Among all the existing managers used to lead the execution of each the CAA's role, the first one created is designated as the leader. The "leader" manager is the centralised point for performing commons tasks within the CAA, such as interrupting the roles, or determining the common exception to be handled in the case where concurrent exceptions are raised

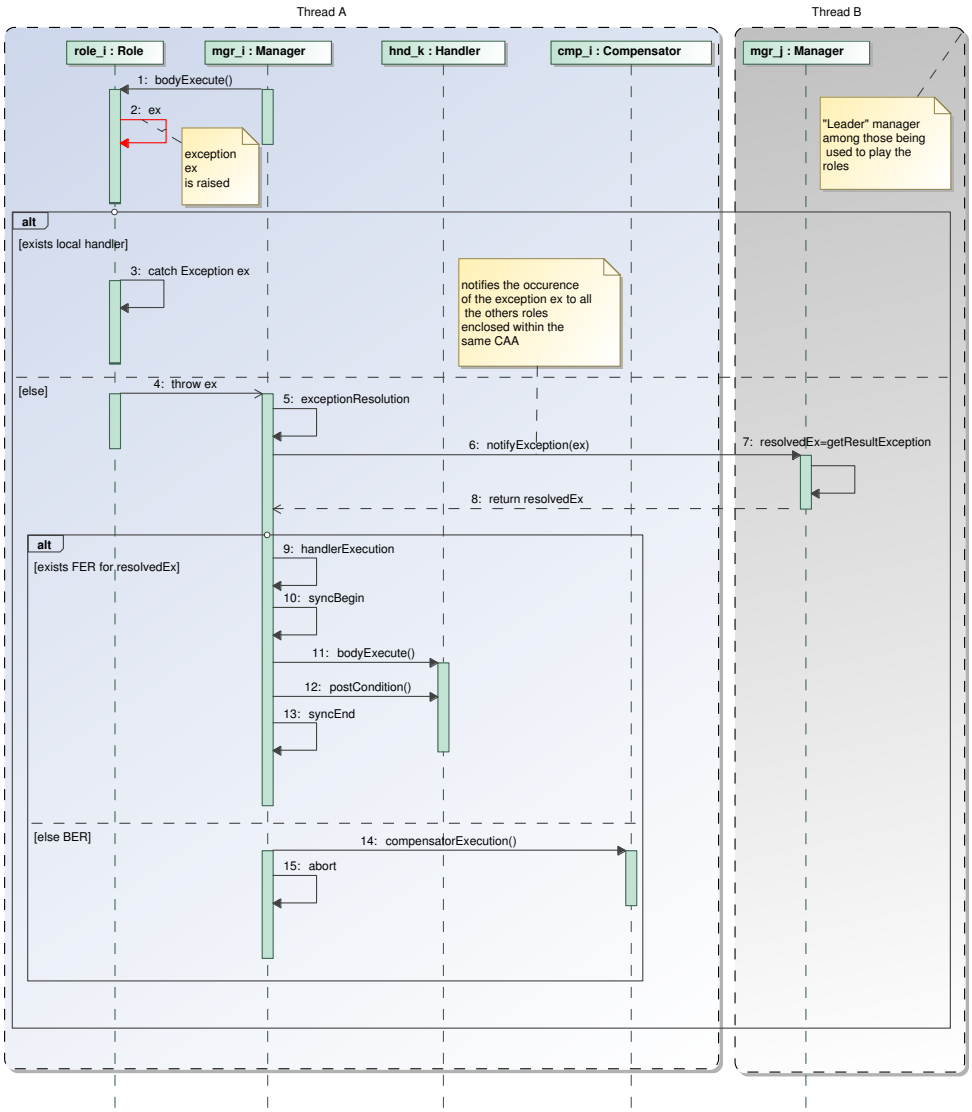

Fig. 4.44: Internal execution for recovering from an exception.

The recovery mechanism executed by Timed-CAA-DRIP is either FER or BER, depending on whether there exists a cooperative handler for the exception. It is worth recalling that a cooperative handler is defined as a set of handlers. Within Timed-CAA-DRIP, handlers used to define a cooperative handler are instances of the class *Handler*.

In the case that a cooperative handler does exist (and then the CAA has as many instances of class *Handler* as roles) for dealing with the exception *ex*, the *handlerExecution* method is called to act over each handler instance. For the sake of simplicity, only one handler (named hnd_k) is shown in the Sequence Diagram. The aim of the *handlerExecution* is to collaboratively execute with the other handlers, those instructions that allow the CAA to recover from the exception. This is achieved by, first synchronising the *handlerExecution*'s initiation with the other peer handlers (by calling the method *syncBegin*), and then calling the method *bodyExecute*. This method is overridden by the programmer to include Java native instructions that allow certain instructions to be handled.

It must be noted, that the same manager used to lead the execution of the role, is also used to lead the execution of the handler. Hence, once the handler completes the execution of *bodyExecute*, the manager calls the post-condition method of the same handler to check whether it passes or fails its acceptance test. In case all the handlers have passed their respective tests, the execution follows the same sequence of method calls as in the case when no exceptions have been raised. Otherwise, the BER is started in order to abort the CAA.

The BER occurs when a cooperative handler for the raised exception does not exist or when an exception takes place during the FER. The BER performs the necessary steps to undo all the effects of having executed the CAA thus far. This is achieved by (1) compensating the execution of each role, and (2) calling the process *abort*. The compensation of the tasks done by role $role_i$ are performed by calling the method *compensatorExecution*. This method encodes the instructions such that each *manually-recoverable* external object can be restored to a consistent state, presumably a state similar to that that object had before initiating the CAA.

On the other hand, the method *abort* (belonging to the manager) interfaces with each *auto-recoverable* external object by calling its implementation of the method *abort*. The *abort* method implemented by each external object is part of the *Transaction* interface to be implemented. (Details of this interface and its aim are given in the next section).

4.4.3 Classes and interfaces for Java

This section presents the interface for the programmer implementing a given CAA design. This interface has been engineered to force programmers to adhere to the Timed-CaaFWrk concepts. This fact indicates that the interface drives the work of the programmer during the implementation phase such that the final implementation is consistent with respect to the given design. Consistency , in this case, is determined by the manner in which each CAA of the design is implemented, i.e. as roles, handlers, compensators, external and internal objects, etc.. It must be noted, however, that the interface is a means to ease the implementation phase. Further, its use does not guarantee a correct implementation.

This section also includes information regarding the expected method for employing the different elements of the interface. This section, then, can be considered as the programmer's reference manual.

The Timed-CAA-DRIP implementation framework is composed of eight components (or Java packages). The Class Diagram that depicts these packages along with their contents is shown in Figure 4.45. In the following, each of these packages is explained. The aim is to present the interfaces and classes that must be implemented and extended by the programmers to achieve the desired implementation.

4.4.3.1 Package caa

Package caa is the core package of the implementation framework. It provides implementation support for the notions of *caa, role, handler* and *compensator*. The class *CAA* is the "facade" (as described in [GHJV95]) that the programmer uses to combine all the parts of a CAA (i.e. roles, handlers and compensators) to achieve its implementation. The class *CAA* (see Listing 4.1) has the methods that allow programmers to set up a CAA, perform its launching, define internal objects within a role of the CAA, exchange messages between these roles to share internal objects, and gather objects that were declared within the context of a same participant,

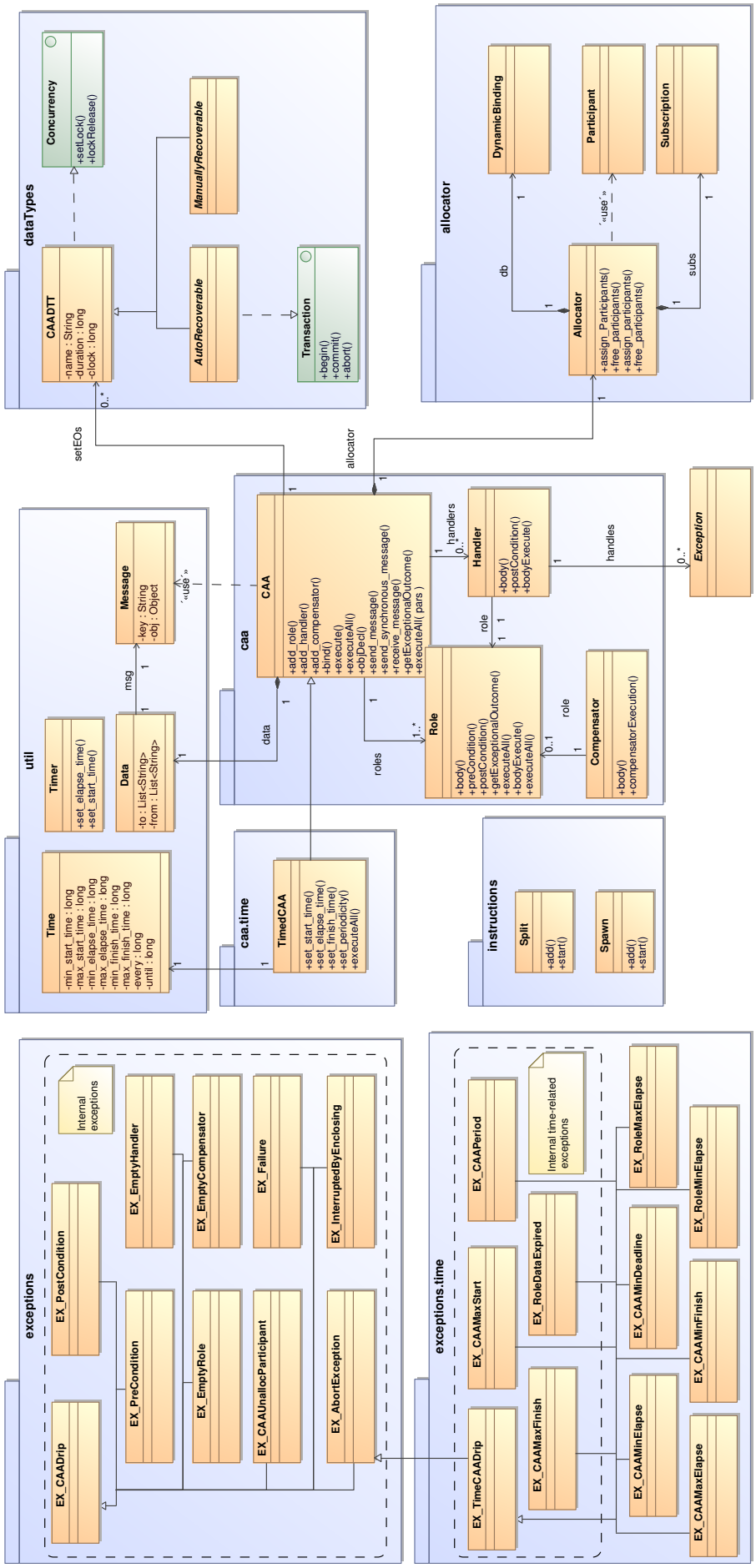

Fig. 4.45: Timed-CAA-DRIP class diagram.

but in the different phases it may go through (i.e. normal phase, implemented using the class *Role*; and abnormal phase, implemented using the classes *Handler* and *Compensator*).

Listing 4.1: Methods of class CAA to be used by the programmer.

```
public class CAA {

  // Constructor
  public CAA(String name);

  // CAA set up
  public void add_role(Role role);
  public void add_handler(Handler handler, Role role);
  public void add_compensator(Compensator comp, Role role);
  public void bind(Exception ex, Handler handler);

  // CAA launching
  public void execute(Role role);
  public void executeAll(Participant [][] pars);

  // Internal object declaration
  public Object objDecl(Role role, String objName, Object obj);
  public Object objDecl(Handler handler, String objName, Object obj);

  // Gathering an internal object
  public Object getObj(String objName);

  // Gathering of the CAA outcome
  public Exception getExceptionalOutcome();

  // Message exchange
  public void send_message(Message msg, String fromRoleName, String toRoleNAme);
  public void send_synchronous_message(Message msg, String fromRoleName,
                                       String toRoleName);
  public Message receive_message(String ID, String fromRoleName,
                                 String receiverRoleName);
}
```

The setting or construction of a CAA is defined by putting together not only its roles, but also the handlers and compensators used during the recovery phase when exceptions are raised. Hence, once a new instance of the class CAA has been created, it is used to add 1) roles (by the method *add_Role*), 2) handlers (by the method *add_Handler*), and 3) compensators (by the method *add_Compensator*) that compose the CAA. When adding a handler and/or a compensator to a CAA, the programer must indicate which role(s) they are related to. An example demonstrating how these methods are employed is given in the Listing 4.6. In this example, the handlers hnd_1 and hnd_2 are associated with the roles $role_1$ and $role_2$, respectively. A similar association between compensators and roles is shown.

It is worth pointing out, that while the existence of a CAA is determined by the roles that compose it (i.e. the method *add_role* must be always used when setting a CAA), its handlers or compensators depend on the exceptions the CAA must deal with. In this manner, it is not mandatory to use the methods *add_handler*, and *add_compensator* when setting a CAA.

The roles, handlers and compensators to be used when setting a CAA must be instances of the classes created by the programmer. This is achieved by extending the implementation framework classes *Role, Handler* and *Compensator*. For each of these classes, the programmer has to override certain methods. The Listing 4.2 shows which methods are to be overridden by the programmer when extending from the class *Role*, whereas the Listings 4.3 shows a (partial) example of a role class (named *Role1*) as the programmer is expected to provide. In this example, the methods *preCondition* and *postCondition* are overridden in a manner such that the value they return depends on the value held in the boolean variables *pre* and *post*.

Listing 4.2: Methods of class Role to be overridden by the programmer.

```
public class Role {

  public void body();
  public boolean preCondition();
  public boolean postCondition();
}
```

Listing 4.3: Example that shows how the class *Role* can be extended.

```
1  package test.caa.myCAA;
2
3  public class Role1 extends Role {
4
5    boolean pre, post;
6
7    // Constructor
8    public Role1(String n, CAA caa) {
9            super(n);
10           setCAA(caa);
11   }
12
13   public boolean preCondition(){
14           if(pre)
15               return true;
16           else
17               return false;
18   }
19
20   public void body() {
21     try{
22         System.out.println("Role 1 executing its body");
23     }catch (Exception e) {
24         throw e;
25     }
26   }
27
28   public boolean postCondition(){
29           if(post)
30               return true;
31           else
32               return false;
33   }
34 }
```

The methods of the classes extending from *Handler* and *Compensator* to be overridden by the programmer are shown in the Listings 4.4 and 4.5, respectively. It must be noted, that in all these classes, the method *body* has to be overridden with instructions that specify the behaviour of the role, handler and compensator. However, only a role is expected to have an associated pre-condition, since for both handlers and compensators, their pre-condition is always assumed as trivial (i.e. true). In addition, a handler may have a post-condition which describes the acceptance test to be passed by the participant when executing the handler. Whether the condition is met or not will determine the kind of outcome to be returned by the overall CAA. On the other hand, a compensator does not have a post-condition as its aim is to restore the manually recoverable objects such that the CAA can be considered as aborted.

Listing 4.4: Methods of class Handler to be overridden by the programmer.

```
public class Handler {

  public void body();
  public boolean postCondition();
}
```

Listing 4.5: Method of class Compensator to be overridden by the programmer.

```
public class Compensator {

  public void body();
}
```

A CAA knows which exceptions it is capable of dealing with once an exception is bound to a particular handler. The method *bind* is the means the programmer uses a link a particular exception to the handlers that replace the respective roles when such an exception occurs. In the example of Listing 4.6, the handlers hnd_1 and hnd_2 are bound to the exception *myException*. Hence, whenever this exception occurs during the normal execution of the CAA named *MyCAA*, these handlers will takeover the roles $role_1$ and $role_2$, respectively, to execute the instructions coded in their body methods.

Listing 4.6: Example that shows how to set up a CAA.

```
1  public static void main(String[] args) {
2
3    try {
4      CAA caa = new CAA("MyCAA");
5
6      // Creating roles
7      Role role_1 = new test.caa.myCAA.Role1("Role1",caa);
8      Role role_2 = new test.caa.myCAA.Role2("Role2",caa);
9      // Adding roles to the CAA
10     caa.add_role(role_1);
11     caa.add_role(role_2);
12
13     // Creating handlers
14     Handler hnd_1 = new test.caa.myCAA.handlers.Handler1("Handler1",caa);
15     Handler hnd_2 = new test.caa.myCAA.handlers.Handler2("Handler2",caa);
16     // Binding each Handler with its respective Role
17     caa.add_handler(hnd_1, role_1);
18     caa.add_handler(hnd_2, role_2);
19
20     // Binding Handlers with the exceptions to be handled
21     EX_MyException myException = new EX_MyException();
22     caa.bind(myException, hnd_1);
23     caa.bind(myException, hnd_2);
24
25     // Creating compensators
26     Compensator cmp_1 = new test.caa.myCAA.compensators.Compensator1("Cmp1",caa);
27     Compensator cmp_2 = new test.caa.myCAA.compensators.Compensator2("Cmp2",caa);
28     // Binding each Compensator with its respective Role
29     caa.add_compensator(cmp_1, role_1);
30     caa.add_compensator(cmp_2, role_2);
31
32     ...
33   } catch (Exception ex) {
34         ...
35   }
36 }
```

The methods concerned with the setting and gathering of internal objects, as well as with the way the outcome of the executed CAA is known are detailed in the following subsections. Their explanation requires the use of other interfaces which have not been introduced, yet.

4.4.3.2 Package caa.time

Package caa.time includes those classes related to the management of timed CAAs. A timed-CAA has all the features of a CAA, plus the means to set any of the allowed time constraints a CAA may own. Hence, the class TimedCAA (shown in the Listing 4.7) extends from the class

CAA in order to capture the same methods that allow the programmer to interface with the other components of the CAA. In addition the class TimedCAA also includes the methods for setting the constraints regarding its starting time (by the method *set_start_time*), elapse time (by the method *set_elapse_time*), finish time (by the method *set_finish_time*), and periodicity (by the method *set_periodicity*). The values (type *long*) to be passed to these methods are expected to be in the time unit *seconds*. This is the time unit used by default by the implementation framework.

Listing 4.7: Methods of class *TimedCAA* to be used by the programmer.

```
public class TimedCAA extends CAA {

  public TimedCAA(String name); //Constructor

  public void set_start_time(long t0_time, long t1_time);
  public void set_elapse_time(long t_emin, long t_emax);
  public void set_finish_time(long t_fmin, long t_fmax);
  public void set_periodicity(long t_every, long t_until);
}
```

4.4.3.3 Package instructions

The programmer, when overriding the method *body* in the classes that extend from either *Role, Handler* or *Compensator* uses Java native code to implement the behaviour described in the design. At the design level, this behaviour is described by the instructions supported by the conceptual framework CaaFWrk. These include: *While, If, Repeat, Split, Spawn* and others (see Sections 4.2 and 4.2.2 for full details about the instructions supported by the conceptual framework). For instructions such as *Repeat, While, If, Raise* and *Signal*, there exists direct support in native Java code (keywords with the same semantics exist in the Java Language). However, for the instructions *Split* and *Spawn* that are part of the CaaFWrk, there does not exist Java Language equivalent that allows the programmer to implement them in a straightforward manner.

The Timed-CAA-DRIP solves this problem by providing the programmers the package *instructions*, which contains the classes *Split* and *Spawn*. These classes are aimed at easing the implementation of the conceptual-level instructions of the same name. The Listings 4.8 and 4.9 shows the methods being provided to the programmer to achieve the *Split* and *Spawn* behaviour, respectively. Since the means to interface with both classes are the same, the explanations are given only for the *Split* class. It is worth noting that despite the fact that both classes export the same methods, their behaviour at run-time (supported by Timed-CAA-DRIP) is different since a *Split* instruction is aimed at creating new execution processes (in this case implemented as threads) that have to be joined at the end of their execution to allow the original execution process that created them to continue (i.e. it gets blocked upon calling *Split*). Conversely, *Spawn* creates new execution processes that (1) do not need to joined upon completion, and (2) their creation does not block the execution of the original execution process (see Section 4.2.2 for details about their behaviour).

The class *Split* gives the programmer two methods: *add* and *start*. The first method is used to define instructions, whereas the second one to create new execution branches. Those instructions defined using the method *add* are executed in one of the new branches created upon calling the method *start*. An example demonstrating the use of these methods, is given in the Listing 4.10. In this example, the class *Split* is used to create two new execution processes or branches, as is commented in the sampled code. The method *add* must receive an object of type *Callable*. An

object callable is one that implements the interface *Callable*. Implementing this interface means the method *call*, which must return a result, is overridden. (In this example, the result returned by the method *call* is an *Integer*. The instructions to be executed by each of these branches are those coded within the method *call* (lines 8-11 for the first branch, and lines 17-20 for the second branch).

Listing 4.8: Methods of class *Split* to be used by the programmer.

```
public class Split {

  public void add(Callable obj);
  public void start();
}
```

Listing 4.9: Methods of class *Spawn* to be used by the programmer.

```
public class Spawn {

  public void add(Callable obj);
  public void start();
}
```

Listing 4.10: Example that shows how to use the class *Split*.

```
1  public void body() {
2    try{
3         ...
4         Split s = new Split();
5
6         // Branch 1
7         s.add(new Callable() {
8                 public Integer call() throws Exception {
9                         System.out.println("Branch 1 - Instruction 1");
10                        System.out.println("Branch 1 - Instruction 2");
11                        return 0;
12                }
13        });
14
15        //Branch 2
16        s.add(new Callable() {
17                public Integer call() throws Exception {
18                        System.out.println("Branch 2 - Instruction 1");
19                        System.out.println("Branch 2 - Instruction 2");
20                        return 0;
21                }
22        });
23
24        // Launching the branches
25        s.start();
26
27        ...
28   }catch (Exception e) {
29      ...
30   }
31 }
```

Once the branches have been created and added (in the above example the creation of the branches is done at the same time they are added), all that remains is to call the method *start* to make the creation of the new execution processes effective. The Timed-CAA-DRIP implementation framework, when executing this code at run-time, will create a new thread for each branch, blocking the thread from where the call has been made, which waits for the completion of the newly created threads.

4.4.3.4 Package allocator

The package *allocator* includes those classes that are concerned with the creation and allocation of the participants modelled at design-time. The class *Participant* (see Listing 4.11) by its constructor allows the programmer to create the participants. Each participant has a name that is used as its identifier. The methods *get_feature* and *add_feature* provide the means to set and retrieve the features that characterise a particular participant. As previously explained, these features can be used to decide which participant fits the needs of the CAA to be executed better.

Listing 4.11: Method of class *Participant* to be used by the programmer.

```
public class Participant {

  public Participant(String name); //Constructor
  public Integer get_feature(String key);
  public void add_feature(String key, Integer value);
}
```

The creation of the participants must be followed by their subscription to the roles executed within a CAA. The class *Subscription* (see Listing 4.12) allows the programmer to link participants with roles of different CAAs. The method *add_participant* (as shown in its interface) is used to register a participant p, to play the role of name *roleName* in the CAA of name *caaName*. Retrieving participants from the subscription table is achieved by the methods *get_participants* and *get_participant*. The former returns an array of participants such that each element in the array is a participant that has been subscribed for playing the role that was passed as a parameter along with the CAA it belongs to. On the other hand, *get_participant* returns a participant by indicating its name along with the role and the CAA is has been subscribed to.

Listing 4.12: Methods of class *Subscription* to be used by the programmer.

```
public class Subscription {

  public boolean add_participant(String caaName, String roleName, Participant p);
  public Participant [] get_participants(String caaName, String roleName);
  public Participant get_participant(String caaName,String roleName,String partName);
}
```

The example shown in Listing 4.13 gives an idea of how these methods are to be used by the programmer after having set a CAA and before its launching. In this example, a private method named *setParticipants* implements the subscription of the participants to be used for executing the CAA named *MyCAA*. These participants (named $Participant_{1,2,3}$) are subscribed such that the role $Role_1$ may be played by the participant $Participant_{1,3}$, whereas the role $Role_2$ may be played by $Participant_{2,3}$.

Listing 4.13: Example that shows how to use the classes *Participant* and *Subscription*.

```
1  public class Main {
2
3    static Subscription subscription = Subscription.Instance();
4
5    private static void setParticipants() {
6      Participant p1 = new Participant("Participant1");
7      Participant p2 = new Participant("Participant2");
8      Participant p3 = new Participant("Participant3");
9
10     subscription.add_participant("MyCAA", "Role1", p1);
11     subscription.add_participant("MyCAA", "Role2", p2);
12     subscription.add_participant("MyCAA", "Role1", p3);
13     subscription.add_participant("MyCAA", "Role2", p3);
14   }
```

```
15
16
17  public static void main(String[] args) {
18
19    try {
20      ...
21      // Creating and subscribing participants
22      setParticipants();
23      ...
24      // Launching the CAA
25      caa.executeAll(new Participant[][]
26        {null, {subscription.get_participant("MyCAA","Role2","p2")}});
27
28      // Gathering CAA outcome
29      Exception result = caa.getExceptionalOutcome();
30
31      // Checking what kind of outcome it has returned
32      if(result!=null){
33          System.out.println("Not normal outcome!");
34
35          if(result.getClass().isAssignableFrom(EX_MyException.class))
36             System.out.println("The exceptional outcome of the CAA is EX_MyException");
37          if(result.getClass().isAssignableFrom(exceptions.EX_AbortException.class))
38             System.out.println("The CAA has aborted");
39          if(result.getClass().isAssignableFrom(exceptions.EX_Failure.class))
40             System.out.println("The CAA has failed");
41
42      }else
43          System.out.println("Normal outcome!");
44      ...
45    } catch (Exception ex) {
46            ...
47    }
48  }
```

This example is also useful for demonstrating how the methods *executeAll* and *getExceptionalOutcome*, which are part of the class *CAA* (see Listing 4.1). The method *executeAll* is used by the programmer to start *composite* CAAs, only[19]. Lines 25-26 in the example show how the method is used. Notice that this method must receive as an input parameter a multidimensional array. The $i-th$ element of the array is also an array, which contains those participants willing to play the $i-th$ role of the CAA. At run-time, the Timed-CAA-DRIP implementation framework will select, for each role, one (and only one) participant among those passed as parameters willing to play it. In the example, it is indicated that $Role_2$ has to be played by $Participant_2$, whereas for $Role_1$ the implementation framework will choose one from those that have been subscribed to play the role $Role_1$.

Once the CAA is launched by means of the method *executeAll*, the caller may want to check the result of its execution. The class *CAA* gives the programmer the method *getExceptionalOutcome* (see line 29 in the Listing 4.13) to check whether execution ended normally. i.e. its outcome is normal, or in a degraded manner. An exceptional outcome that returns a *null* value presents the normal execution of the CAA. Conversely, when the value is not *null*, the CAA has either completed its execution in a degraded manner (lines 35-36), aborted (lines 37-38), or failed (lines 39-40). Notice that the exception returned by the method is used as a means to identify not only whether the CAA has completed normally or not, but also to determine the nature of degraded result.

The Sequence Diagram depicted in Figure 4.46 merges the examples presented in the Listing 4.6 and 4.13 to give the reader a complete view of the sequence of calls that are required to set up and launch a CAA when using Timed-CAA-DRIP.

[19] For starting *nested* CAAs, the programmer has to used the method *execute*, which receives as its input parameter the role to be started by the caller upon execution of the method.

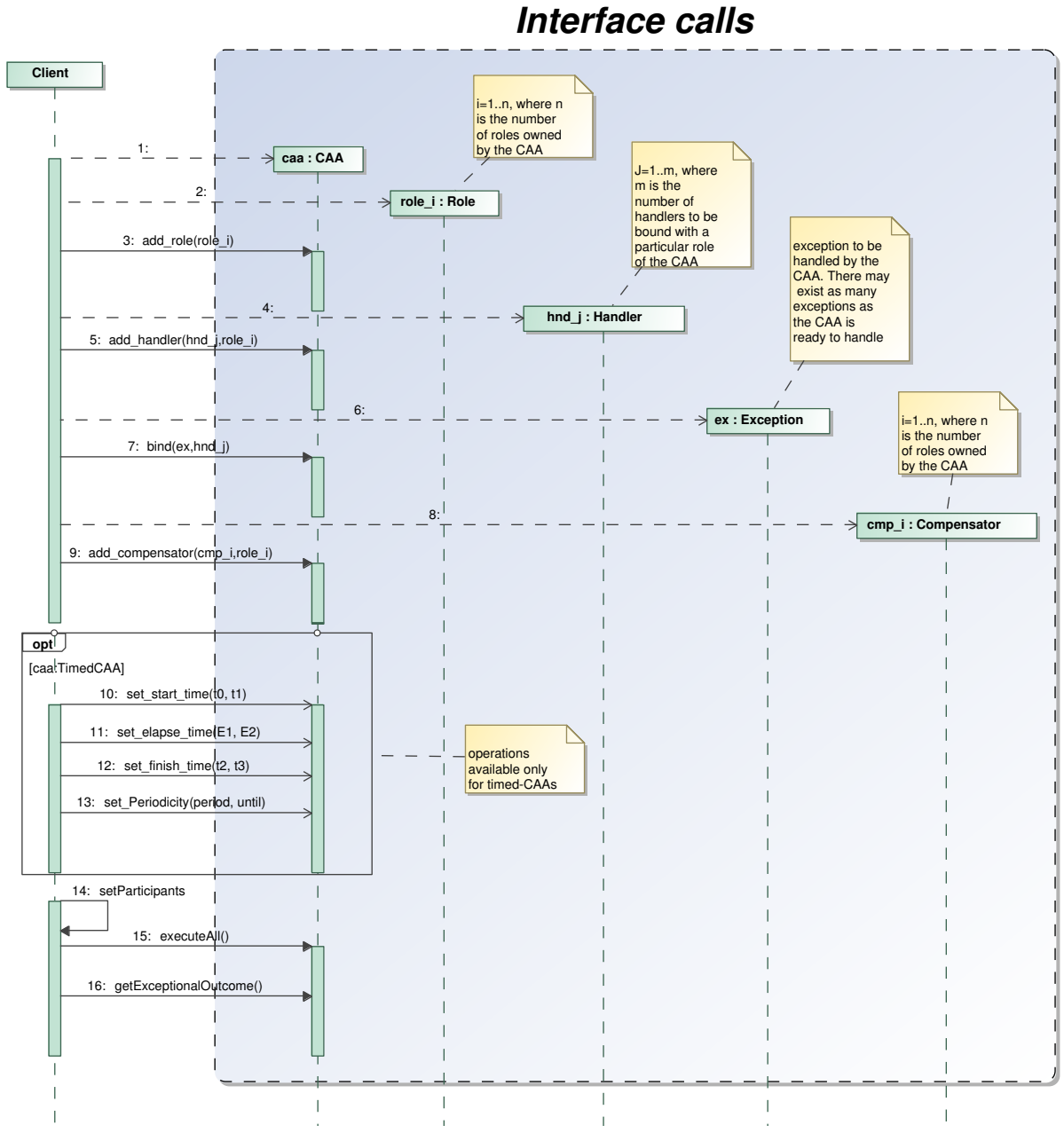

Fig. 4.46: Setting up and launching of a CAA.

4.4.3.5 Package util

The package *util* includes classes aimed at supporting the implementation of time-constraints within roles and the definition of messages to be used when exchanging information between participants during the normal and abnormal phases.

According to the proposed time-related extensions to the Timed-CaaFWrk, time constraints may be set within a role (see Section 4.3.2 for details about this extension). Implementing this time constraint in a role imposes in-line instructions within the *body* method, as this is the method that implements the role behaviour as defined at the design phase. These in-lines operations embedded within a role are aimed at either delaying the execution of a set of instructions or monitoring whether some executed instructions actually execute within certain time frame.

To release the programmer from the burden of engineering and implementing the code that fufils these needs, the Timed-CAA-DRIP has been enhanced with features to make the implementation of time constraints within a role much easier. The class *Timer* is the means provided by the implementation framework to let the programmer set the delay and (minimum and maximum) deadlines over some coded instructions within a role. The Timed-CAA-DRIP will ensure that at run-time the delay is respected. Further, it will monitor whether the instructions are executed within the time frame defined by the deadlines.

Listing 4.14: Methods of class *Timer* to be used by the programmer.

```
public class Timer {

  public void set_start_time(long t0_time);
  public void set_elapsed_time(long t_emin, long t_emax);
  public void start(Callable obj);
}
```

The setting of the delay and deadline is achieved by using the methods *set_start_time* and *set_elapse_time* (see Listing 4.14). Both methods expect time value (of type *long*) represented in *seconds*. Once the time constraints have been set, the remaining tasks for the programmer is to start the execution of the instructions concerned with these constraints. This is achieved by the method *start*, which receives as its input a *Callable* object. This *Callable* object encodes the instructions to be executed. The example shown in the Listing 4.15 is aimed at detailing how these methods are to be used by the programmer. In this example, a timer is created to set a delay of 10 seconds for the instruction that print out to the console, which is expected to be completed in less than 30 seconds.

Listing 4.15: Example that shows how to use the classes *Timer* within the implementation of a role.

```
1  public void body() {
2    try{
3        ...
4        Timer timer = new Timer();
5        timer.set_start_time(5);
6        timer.set_elapsed_time(-1, 30);
7
8        timer.start(new Callable() {
9            public Integer call() throws Exception {
10             System.out.println("Executing this statement")
11             return 0;
12           }
13       });
14       ...
15   }catch (Exception e) {
16     ...
```

```
17    }
18  }
```

Roles within the same CAA are expected to collaborate. This collaboration is achieved by message exchange. A message may carry *data* as part of the information being conveyed. The class *Message* is the means programmers use to deal with the notion of message as considered at the design phase. Depending on whether a message carries data or not, an *object* must be passed as a parameter as well upon constructor invocation. The two different constructors to be used by the programmer when creating a message are shown in the Listing 4.16. Please note, that the objects to be conveyed with a message are expected to be *internal objects*, since these objects are local to the role where they are declared.

Listing 4.16: Methods of class *Message* to be used by the programmer.

```
public class Message {

        public Message(String key);
        public Message(String key, Object obj);
}
```

An example that shows how messages are declared using these constructors and how messages are exchanged using the methods *send_message* and *receive_message*, which are part of the class *CAA*, is given in the Listing 4.17. In the example it is shown how a role named *currentRoleName*, first sends three different messages ($SendMsg_{1,2,3}$) to a peer role named *receiverRoleName*, and second receives two messages ($ReceiveMsg_{1,2}$) from the peer roles $senderRole_{1,2}$, respectively.

Listing 4.17: Example that shows how to exchange messages between roles.

```
1  public void body() {
2    try{
3        ...
4        /*
5         * Sending messages SendMsg1, SendMsg2 and SendMsg3.
6         * Notice that Msg1 goes along with the internal object called "obj",
7         * whereas SendMsg3 is sent synchronously.
8         */
9        caa.send_message(new Message("SendMsg1"),
10                        "currentRoleName", "receiverRoleName");
11
12       MyDataType obj = (MyDataType)caa.objDecl(this, "obj", new MyDataType());
13       caa.send_message(new Message("SendMsg2",obj),
14                        "currentRoleName", "receiverRoleName");
15
16       caa.send_synchronous_message(new Message("SendMsg3"),
17                        "currentRoleName", "receiverRoleName");
18       ...
19       /* Receiving message ReceiveMsg1 and  ReceiveMsg2 from roles
20        * senderRole1 and senderRole2, respectively.
21        * Notice that upon receiving message ReceiveMsg2 the internal
22        * object "obj2" is created
23        */
24       caa.receive_message("ReceiveMsg1","senderRole1","currentRoleName");
25
26       MyDataType obj2 = (MyDataType) caa.objDecl(this, "obj2",
27         caa.receive_message("ReceiveMsg2","senderRole2","currentRoleName").get_data());
28       ...
29   }catch (Exception e) {
30     ...
31   }
32 }
```

The role *currentRoleName* asynchronously sends the message $SendMsg_1$ (lines 9-10). Notice that that this message does not convey an object. The second message (i.e. $SendMsg_1$) is also

sent asynchronously (lines 13-14), but it conveys the object *obj*, which is declared using the method *objDecl* belonging to the class *CAA*. This method is aimed at giving the programmer explicit support for declaring internal objects within a role. Finally, the third message is sent synchronously (lines 16-17), meaning the sender gets blocked until the recipient acknowledges receipt of the message.

Regarding the reception of messages, it is always made in a synchronous manner. Hence, the received gets blocked till the message is obtained, means that another peer role has sent it. The role named *currentRoleName* receives the first message *ReceiveMsg*1 from the role named $senderRole_1$. This message does not convey any object, as does the second message received, i.e. $ReceiveMsg_2$. The role, upon receiving the second message, creates a new internal object, obj_2, with the data conveyed by the message.

4.4.3.6 Package datatypes

The *Package datatypes* package includes those classes that are related to the management of the called *external* and *internal* objects that the implemented CAAs must deal with. As explained in Section 4.2.1, the existence of a transactional system is assumed to ensure the *consistency, isolation,* and *durability*, i.e. the CID properties, of the external objects that a particular CAA accesses during its execution. The *atomicity*, i.e. the A in the ACID acronym, of the external objects is directly supported by the framework as it has its own semantics for achieving this property. Since the conceptual framework does not set any constraint over the transactional support to be used to achieve the CID properties, the Timed-CAA-DRIP implementation framework cannot assume any technical characteristic of the transactional mechanism employed by the programmer, except that it exists. Hence, the implementation framework provides the interfaces *Concurrency* (see Listing 4.18) and *Transactional* (see Listing 4.19), which must be implemented by the programmer to allow Timed-CAA-DRIP to interface with the chosen locking technique for ensuring *isolation* and transactional support for ensuring *consistency* and *durability.*

Listing 4.18: Interface *Concurrency* to be implemented by the programmer.

```
public interface Concurrency {

        public void setLock();
        public void lockRelease();
}
```

Listing 4.19: Interface *Transaction* to be implemented by the programmer.

```
public interface Transaction {

        public void begin();
        public void commit();
        public void abort();
}
```

Notice, that while all the objects regardless of whether they are *internal* or *external* may be concurrently accessed. But external objects are required to be persistent. Thus, objects that are used as external in the CAA must be of a datatype that implements both interfaces. Conversely, internal objects are datatypes that implement the *Concurrency* interface only. That is the reason why there exists one interface for dealing with concurrency and another interface for handling the transactional support.

4.4.3.7 Packages exceptions and exceptions.time

The packages *exceptions* and *exceptions.time* contain those pre-defined exceptions (non-timed and timed-related, respectively) that may take place at run-time during the execution of the implementation framework. Pre-defined exceptions are divided between those that can be handled by the programmer, i.e. there may exist a handler associated to deal with it, and those that are only handled within the framework and are not visible to the programmer. The latter exceptions are referred to as *internal exceptions*. The Class Diagram shown in Figure 4.45 clearly depicts what the internal exceptions in both packages are. Hence, those that are not internal can be associated using the method *bind* of the class *CAA* with the handlers in charge of recovering the CAA when they occur.

4.5 Automatic code generation

Timed-CAA-DRIP represents a clear advantage for programmers implementing software systems designed according to the *Timed-CaaFWrk* paradigm. The advantage to programmers using *Timed-CAA-DRIP* is the high level of abstraction between the terminology used in the design description and the interface elements of the implementation framework.

Despite the improved facility of implementing design descriptions using *Timed-CAA-DRIP*, which is very similar to the high-level design-description/implementation framework correspondence in *Timed-CaaFWrk*, programmers still have to invest time to learn it. This means that programmers need to go through a training process that implies learning its interface and trying it out several times untll enough experience and confidence is attained. In order to make the implementation phase easier and faster, the goal would be to increase the correspondence between the design description and the implementation framework even further such that the implementation phase is as automated as possible. The formalisation of the *Timed-CaaFWrk* according to the metamodelling principles allows programmers to exploit the MDE approach; in particular, model transformation techniques and tools meant for code generation.

A model transformation is defined as a set of *rules*. These rules determine how a target model can be obtained from a source model. Despite the fact that rules are used to automatically generate the target model from a given source model, their definition is given in terms of the constructs used to define both the source and target models. Hence, each rule describes how one or more *constructs* in the source language can be transformed into one or more constructs in the target language.

Considering the *left hand side (LHS)* of a rule as the part of the rule description that contains information related to the source model, and the *right hand side (RHS)* as the part that owns information related to the target model, the description of a rule can be seen as a function[20] $R : LHS \rightarrow RHS$. Depending of the complexity and structure of the rule's LHS, it would be desirable to decompose its definition into sub-rules to make it tractable. It is said that a rule R has a *compositional* definition when it is defined in terms of sub-rules $R_{sub_1}, ..., R_{sub_n}$.

Model transformations aimed at generating textual artifacts from a given model are known as *Model-to-Text* (M2T) transformations. Those aimed at generating a model are known as *Model-to-Model* (M2M) transformations[Ecl10].

The aim of the required model transformation is to generate *Timed-CAA-DRIP*-compliant (Java) source code from a given *Timed-CaaFWrk*-compliant model. This Java source code is

[20] It is assumed that rules are not ambiguous.

considered as a textual artifact. Hence, the model transformation technique required to achieve the Java source code from a given *Timed-CaaFWrk*-compliant model generator has to support the notion of M2T transformation.

4.5.1 The Kermeta metamodelling language

The implementation of the transformation that allows the programmer to generate Java code from a given *Timed-CaaFWrk*-compliant model is given in *Kermeta* [Tri10a]. *Kermeta* is an open source object-oriented metamodelling language that was initially created to describe operations at the level of the meta-model. Hence, the specification of a meta-model comprises both a structural component defined by the structural relationships between the concepts and a behavioural component defined by the operations enclosed by each concept. The compliance of a model is determined by its adherence to the structural relationships specified by the meta-model, whereas its semantics is determined by the result that is obtained by executing the meta-model operations. Therefore, *Kermeta* can be seen as a language for specifying the behaviour of a domain-specific language (DSL), which is defined in a metamodelling fashion.

The definition of *Kermeta* is given in terms of the metamodelling principles. This means that there exists a meta-model that defines its (abstract) syntax. This meta-model is an extended version of the EMOF[21] meta-model with concepts that introduce the concepts of control structures, inheritance, operation overriding, as well as convenient constructions derived from the Object Constraint Language (OCL), such as closures, e.g. each, collect, select. A complete list of the extensions can be found in [MFJ05]. It is worth mentioning that the action language defined by the extensions introduced to the EMOF's is imperative and object-oriented.

The Listing 4.20 shows an example of an operation described in *Kermeta*. This operation, *step*, is aimed at checking whether there exists at most one element of type *Transition* that has as a name the value (named *c* and of type *String* -line 1) passed as input parameter. In the case that there exists more than one element with the value passed as the input parameter (line 7), then an exception of type *nonDeterminism* is raised (line 8). Otherwise, the name of the fired transition (line 11) is returned.

Listing 4.20: Step operation in Kermeta (Taken from [BNT07]).

```
1  operation step(c: String):String is do
2
3    var validTransitions: Collection<Transition>
4
5    validTransitions := outgoingTransition.select{t | t.input.equals(c) }
6
7    if validTransitions.size > 1 then
8      raise NonDeterminism.new
9    end
10
11   result := validTransitions.one.fire
12
13 end
```

According to the expected shape a rule in a model transformation should have, and in spite of considering an operation as a rule, it is very difficult to identify the LHS and RHS of a rule, when it is specified in *Kermeta*. This is because *Kermeta* was not originally thought to be a transformation language. However, due to the way it is defined and due to its expressiveness

[21] EMOF stands for *"Essential MOF [Obj06]"*.

and features it can be used to such a purpose. However, the main reasons a programmer to chooses *Kermeta* as the tool to implement the M2T transformation are:

- Its compatibility with the Eclipse Modelling Framework (EMF) [Ecl10, BBM03], which allows the Eclipse Development Environment to be used as a workbench to edit, store, and visualise models; and
- above all, the existence of the Kermeta Emitter Template (KET) [Tri10b] facility, which is specifically meant to ease the development process of an M2T transformation.

This KET facility, which is part of the *Kermeta* tool set, allows the programmer to specify within a template an overall sketch of the source code to be generated. This template, which is known as a *KEt* template, is made up of the text that should be written in the output, and tags that are interpreted to generate string values from some computation. There exist three kinds of tags:

- "< % − −" and "− − % >": to define comments within the template,
- "< % =" and "% >": to embed *Kermeta* expressions within the template. The result of the expression is placed within the resulting generated text, and
- "< %" and "% >": to embed *Kermeta* statements or blocks within the template. Each block or set of *Kermeta* statements is known as a *scriptlet*. There is not any limitation on the number of *scriptlets* a template may contain. Moreover, a *scriplets* may reference elements that have been defined in other *scriptlets*.

The interpretation of the tags take place during the compilation of the template. The compilation of a *KET* template, performed by the template engine, produces a *Kermeta* file. It must be noted that the *KET* template may also be embedded in-line *Kermeta* statements, which remain once the template is compiled. So far, the compilation of a *KET* template must be manually performed before launching the M2T transformation. This M2T transformation is made of *Kermeta* files, only.

The Listing 4.21 shows some parts of the *KET* template used to generate the Java source code corresponding to a *Role* element in a *Timed-CAA-DRIP*-compliant implementation.

Listing 4.21: *KET* template for a *Timed-CAA-DRIP Role* class implementation.

```
1  <%@ket
2  package="TimedCAAFWrk"
3  require="kermeta http://timedCAAFWrk/1.0 ../utils/Helper.kmt InternalFunGenerator.kmt"
4  using="TimedCAAFWrk kermeta::standard kermeta::utils"
5  isAspectClass="true"
6  class="Role"
7  ismethod="false"
8  operation="generate"
9  parameters=""
10 %>
11
12 <%var instr: Set<Instruction> init self.instructions(self.instrs)%>
13 <%instr.select{i| i.isKindOf(Execute)}.each { i | %>
14 <%=i.asType(Execute).~operation%>
15 <%}%>
16
17 package lu.uni.lassy.<%=self.container.container.asType(CAADesign).name%>
```

```
18                          .caa.<%=self.caa.name%>;
19
20  import java.rmi.RemoteException;
21  ...
22
23  public class <%=self.name%> extends RoleImpl {
24    ...
25
26    public <%=self.name%>(String n, CAA caa) throws RemoteException {
27        super(n);
28        setCAA(caa);
29    }
30
31    public void body() throws Exception {
32        ...
33    }
34  }
```

The first part of the template (lines 1-9) corresponds to the mandatory *header* that every *KET* template must have. This *header* contains information regarding the generation of the *Kermeta* file. This information is provided by eight directives [Cyr10]:

- *package:* specifies the root package of the *Kermeta* file,
- *require:* set of requires that the Kermeta file should contain,
- *using:* declares the external *Kermeta* files that need to be imported,
- *class:* main class of the Kermeta file,
- *isAspectClass:* true if the main class is an aspect,
- *operation:* name of the main operation,
- *isMethod:* true if the main operation is a redefinition of an existing operation, and
- *parameters:* parameters of the generate(...) method. These parameters can be used in *KET* tags to gather data from outside the generator.

Directives like *package, require, using, class,* and *parameters* are mandatory, whereas *isAspectClass, operation* and *isMethod* are optional.

The remaining part of the example shows how *kermeta* expressions (lines 14, 17, 18, 23 and 26) and *scriptlets* (lines 12-13 and 15) are embedded within the template.

4.5.2 Transforming Timed-CaaFWrk models into Java source code

The layout of the M2T transformation that allows a programmer to generate Java source code compliant with the *Timed-CAA-DRIP* implementation framework from a given *Timed-CaaFWrk*-compliant model is shown in Figure 4.47. This figure shows both the native *Kermeta* files and those automatically obtained from the compilation of a *KET* template. The binding between a *KET* template (double-border rectangle) and the *Kermeta* file, the single-border rectangle, that is obtained from its compilation is shown in the figure by a curly arrow. The straight arrow is used to denote the "use" (or inclusion) of a *Kermeta* file by another one. The *Kermeta* file that does not hold any incoming straight arrow represents the entry point or the

main file of the transformation. This file is *M2T.kmt*, and it is called by the user to obtain the generated code. This file receives as input a model that is expected to be compliant with the *Timed-CaaFWrk* meta-model, and generates a set of Java files into the a directory called *output*. There exist other two *Kermeta* files that form part of the model transformation: *Helper.kmt* and *Persistance.kmt* (rounded-corner rectangle). The *Helper.kmt* contains general purpose functions that are required from different files, whereas *Persistence.kmt* centralises the operations that deal with the load and store the model and Java source files, respectively. The full set of files that compose this M2T transformation can be downloaded from [DT410].

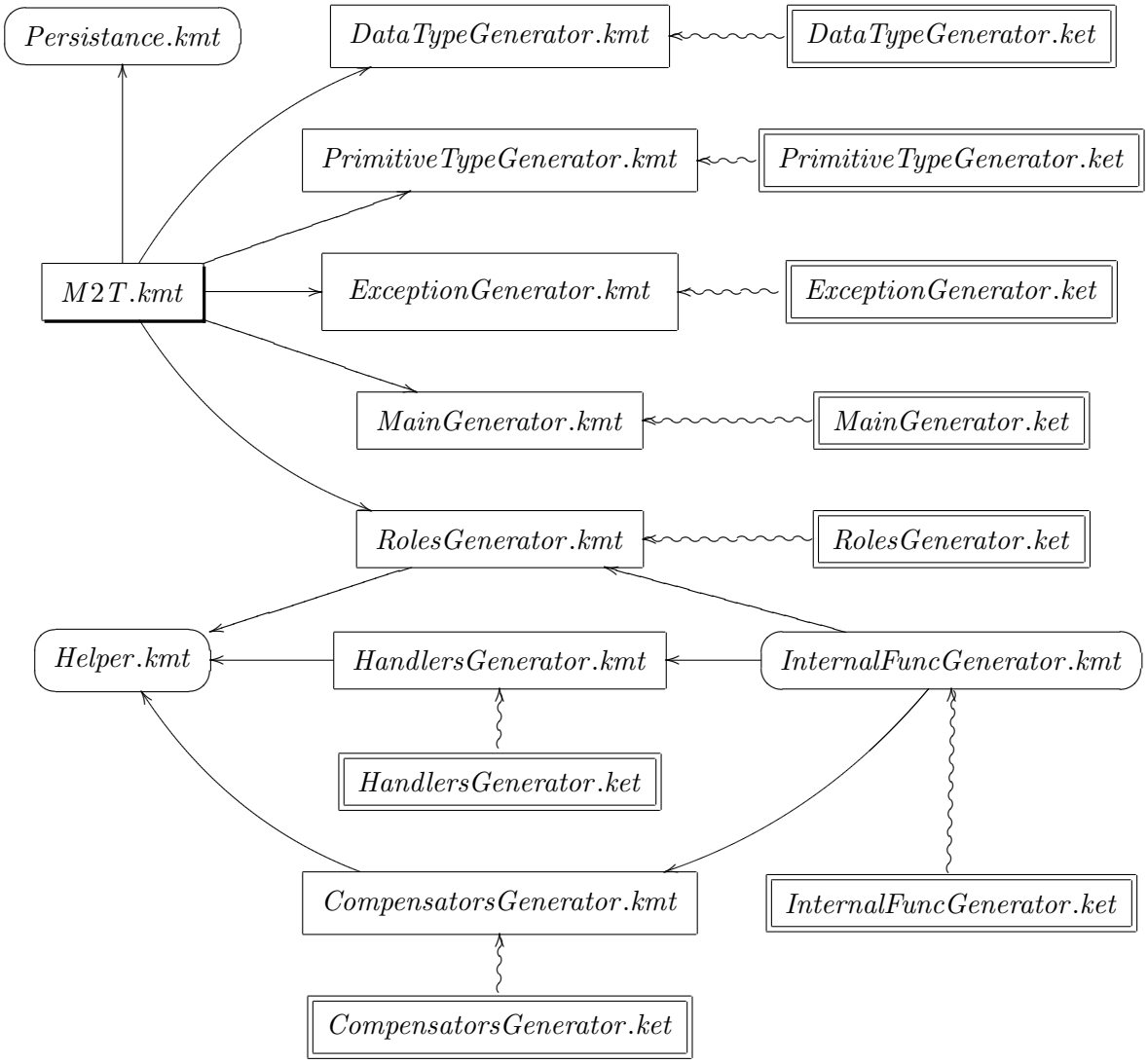

Fig. 4.47: Layout of the M2T transformation.

It is worth emphasising that this M2T transformation is general purpose. This means that the generated source code may be used as a prototype to evaluate whether the modeller has captured the requirements properly or to be deployed and released as the software to be used "in production". Notice that in this case the modeller has described the requirements according to the *Timed-CaaFWrk* paradigm. Thus, the specification of the requirements, which are given in

CAA terms, will usually describe the *design* of the software system to be developed. However, nothing forbids describing the user's requirements directly in terms of CAAs.

5. DT4BP SEMANTICS

Abstract

This Chapter describes the semantic mapping that relates each DT4BP *concepts with a concept in the* Timed-CaaFWrk *semantic domain. The chapter opens by providing an overview of the expected information to be provided by a semantic mapping, and the particular notation being used to describe the mapping. This is followed by a description of the semantic mapping. The chapter closes describing how this semantic mapping is combined with other model-driven artifacts to achieve the execution of* DT4BP *models, and how to use this approach for validation purposes.*

5.1 Overview

A language L consists of a *concrete syntax* (or syntactic notation, which defines the legal elements of L), an *abstract syntax* (or abstract version of L suitable for its manipulation by computer tools), a *semantic domain* (used to provide meaning to the legal elements of L), and a *semantic mapping* (used to bind each legal element with its respective meaning) [HR04]. Hence, when defining a language, these four elements, i.e. concrete and abstract syntax, semantic domain, and semantic mapping, must be explicitly defined.

Regarding the definition of the *DT4BP* modelling language, its *abstract* syntax is described in Chapter 3 using the metamodelling principles, whereas the complete list of BNF[1] rules that define the *concrete syntax* is given in the Appendix A. The Timed-CaaFWrk conceptual framework presented in Chapter 4 is the semantic domain used to assign meaning to *DT4BP* syntactic expressions. What remains to be provided is the semantic mapping that makes explicit the association of each *DT4BP* syntactic element with its meaning as an element over the semantic domain.

Making explicit the definition of the semantic mapping implies relying on a notation that is able to manipulate both the language under definition (aka *source* language) and the semantic domain (aka *target* language). As explained in Chapter 2 Section 2.3.2 and Chapter 4 Section 4.5, within the context of MDE and more precisely within the area of metamodelling, the technique on which this thesis relies on when formalising concepts, exists the notion of *model transformation*, whose aim is to provide support in specifying the way in which a "target" model may be obtained from a "source" model. Hence, considering the *abstract syntax* of the language under definition as the source model, and the *core concepts* of the semantic domain used to give meaning to the elements of the source language as the target model, the notion of model transformation is a suitable means for specifying the semantic mapping.

[1] Backus-Naur Form [AU77].

5.1.1 The ATLAS model transformation language

The semantic mapping element that corresponds to the definition of the *DT4BP* modelling language is described by means of the ATLAS model transformation language (ATL) [ATL10a, JK06, JABK08]. ATL is not a general purpose metamodelling language as *Kermeta*, since it has been engineered to deal with model transformations, and in particular with Model-to-Model (M2M) transformations.

In ATL, the notion of *module* is used to define a model transformation. A module contains a *header* and a set of *rules* that determine the transformation. The *header* defines the name of the transformation and the variables used to refer to the source and target models. The variables used to refer to the source and target models are typed by the meta-models to which these models must conform. The variable that refers to the target model is defined using the keyword *crate*, whereas the keyword *from* is used to define the variable that refers to the source model. The Listing 5.1 shows the header of the M2M transformation that defines how *DT4BP*-compliant models are transformed into *Timed-CaaFWrk*-compliant models. This transformation is named *DT4BP_to_TimedCaaFWrk* (line 1), and the variables used to refer to the source and target models are named *IN* and *OUT*, respectively (line 3)

Listing 5.1: Header of the *DT4BP* to *Timed-CaaFWrk* transformation.

```
1 module DT4BP_to_TimedCaaFWrk;
2
3 create OUT : TimedCAAFWrk from IN : DT4BP;
```

It is worth mentioning that during the execution of an ATL transformation, the source model may be navigated, but it cannot be changed (i.e. it is loaded as read-only). Conversely, the target model is available as write-only, but is cannot be navigated. ATL requires that both models and meta-models are expressed in the XMI OMG serialisation format. Notice that regarding meta-models, in the context of this thesis both the *DT4BP* and *Timed-CaaFWrk* meta-models have been implemented as Ecore meta-models [BBM03].

The way in which the target model is created from the source model is specified by the set of rules enclosed within the module. A rule, defined by the keyword of the same name, is composed of two mandatory parts referred to as the *source pattern* and *target pattern*. The *source pattern* is defined using the keyword *from*. The *target pattern* is defined by the keyword *to*. Both keywords allow the programmer to specify a typed variable. For the source pattern, the variable type must correspond to a concept of the source meta-model. For the target pattern, the variable type must correspond to a concept of the target meta-model.

The aim of source pattern is to specify the matching type an element of the source model has to conform to, thus allowing the rule be triggered. Every time the rule is triggered, a new element of the target model is created according to the specification given by the target pattern. The Listing 5.2 shows the definition of the rule named *Exception*. This rule specifies that for each instance of type *Exception* in the *DT4BP*-compliant source model (line 3), an instance of type *Exception* in the *Timed-CaaFWrk*-compliant target model is to be generated (line 5). Moreover, the rule also states that the attribute *name* of the instance (of type *Exception*) to be created in the *Timed-CaaFWrk*-compliant target model is initialised with the value of the attribute *name* owned by the instance in the *DT4BP*-compliant source model that triggers the execution of the rule. This is defined in the *binding* part of the target pattern (line 6).

Listing 5.2: Transformation rule for the *Exception* concept.

```
1 rule Exception {
2         from
3                 s : DT4BP!Exception
4         to
5                 t : TimedCAAFWrk!Exception(
6                         name <- s.name
7                 )
8 }
```

The binding part of a target pattern is aimed at specifying how the features (i.e. attributes and association) of the target type have to be initialised. The symbol '$<-$' is used to assign the initialisation expression to the target type feature. Notice that it is usually the case that the initialisation expressions are defined in terms of the source type feature. In that case, the variable defined in the source pattern are the means to access the value of the element belonging to the source model that caused the triggering of the rule.

It is worth noting that the evaluation of an expression may (or may not) lead to the triggering of another rule before assigning the evaluated value to the target type feature. Whether a rule is triggered or not depends on the type of the resulting value that must be assigned to the target type value. If the type is either a primitive type (i.e. *Boolean, Integer, Real* or *String*) or a type defined in the target meta-model, then the value is simply assigned to the corresponding target type feature. Otherwise its type is either a source meta-model type or a Collection type (i.e. *Set, OrderedSet, Bag* or *Sequence*) in which case the evaluation has to go on until either a primitive or target meta-model type is found. This resolution algorithm then not only allows the elements of the target model to be associated with each other simply navigating the source model, but also eases the definition of compositional rules due to its automatic rule chaining.

A module may also contain a *helper* section, which is used to define "functions" that can be called from different rules. These functions may or may not be included within the same module where they are used. In the case that they are placed in a different file from that where they are called, the caller must include the file. The inclusion of one or more files is achieved by making use of the *import* section, which enables to declare the ATL libraries that must be imported.

Since this section was aimed at providing a short introduction of ATL so that the reader is able to understand the *DT4BP* to *Timed-CaaFWrk* M2M transformation that is presented in the next Section, more details about the *helper* and *import* sections, as well as the transformation language in general can be found in the ATL User Guide [ATL10b].

5.2 Translating DT4BP models in Timed-CaaFWrk models

This section describes how when given a determined *DT4BP*-compliant source model, the major rules that allow the programmer to automatically obtain a *Timed-CaaFWrk*-compliant target model. The complete set of rules is given in the Appendix E. It is worth recalling that rules are described in ATL, and the source and target meta-models, which they rely on to accomplish their execution, are extended versions of the *DT4BP* and *Timed-CaaFWrk* meta-models shown in Appendices B and D, respectively. The extensions introduced in the original meta-models are those required to make them implementable as Ecore models. Hence, new concepts were introduced such that both meta-models adhere to a tree structure when considering the *composite* relationship as a means to bind two different concepts. The extended Ecore version of these meta-models are also shown in the Appendix E, beside the description of the M2M transformation rules.

The rules are presented in the order in which they are "expected" [2] to be triggered. This order is determined by the tree-structure of the source meta-model to which the source model adheres due to its compliance with the respective meta-model. Hence, the first rule to be described is the one whose source pattern is related to the root concept of the source meta-model. The definition of this rule, then, can be considered as the "main" rule of the transformation, since its triggering leads to the subsequent triggering of the other rules that compose the transformation. Thus, in the following, this main rule and some that are subsequently called are described.

5.2.1 Main

Let m_{DT4BP} be a model that is compliant with the *DT4BP* source meta-model. Due to this compliance, the model m_{DT4BP} contains one instance of type *Enterprise*. This instance will lead to the triggering of the rule named *Enterprise*, whose description is shown in the Listing 5.3. The triggering of this rule is the first step in the generation of a model $m_{Timed-CaaFWrk}$ compliant with the target meta-model *Timed-CaaFWrk*.

The target pattern of the rule *Enterprise* specifies that a target element of type *CAADesign*[3] is created every time the rule is triggered (line 6). Hence, for every element of type *Enterprise* in the source element, an element of type *CAADesign* is generated in the target model. The binding part of the rule specifies that the *name* of the target element is initialised with the attribute name value of the source element (line 7). A similar binding between the source and target elements is performed for the *events, dttps* and *basicTypes* attributes (line 8, 23, and 24), respectively. However, it must be noted that the *resources* and *business processes (bps)* of the source element are used to initialise the *participants* and *caas* of the target element (line 22 and 25), respectively.

Therefore, a *business process* is semantically defined as a *CAA*, whereas a *resource* as a *participant*. The fact that a business process is considered as a CAA is due to their similar structural properties. i.e. one or more enclosed parties each with its own set of activities that are expected to be executed. In addition, the properties they do have, i.e. synchronisation upon entry/exit, message exchange between the enclosed parties, and concurrent exception resolution and handling, also support the concept of a business process as a CAA. On the other hand, semantically considering a *resource* as a *participant* is due to the the role they play to execute the business process or CAA, respectively. The actual execution of a business process is performed by the assigned resources. Similarly, the actual execution of a CAA is performed by the *participants* that are assigned to play the roles that make up the CAA.

The way the attribute *exceptions* is initialised deservers further explanation (line 9). This attribute is initialised by the set that results of joining 1) the exceptions held by the source element with 2) the exceptions *maxIntrsExecElapseT, minInstrExecElapseT, maxRoleExecT, minRoleExecT*, and 3) the exceptions appearing in the resolution part of each *business process* owned by the source element (lines 15-21). It should be noted that the exceptions *maxIntrsExecElapseT, minInstrExecElapseT, maxRoleExecT* and *minRoleExecT* are generated by calling the rules *tMaxIntrsExecElapseT, tMinInstrExecElapseT, tMaxRoleExecT* and *tMinRoleExecT* (lines 27-37). This semantic mapping explicitly defines that the set of exceptions a particular *CAA* design is going to handle is determined by the combination of exceptions handled by each business process and the pre-defined exceptions of type *Abort, DataExpired, TimeoutMinBPLast* and *TimeoutMaxBPLast*.

[2] Rules are defined without any assumption about the order in which they are triggered, as ATL is a declarative language.

[3] The type *CAADesign* is the root concept in the target *Timed-CaaFWrk* meta-model.

Listing 5.3: Rule Enterprise.

```
1  rule Enterprise {
2  from
3          s : DT4BP!Enterprise
4
5  to
6          t : TimedCAAFWrk!CAADesign(
7                  name <- s.name,
8                  events <- s.events,
9                  exceptions <- OrderedSet {
10                                 s.exceptions,
11                                 tMaxInstrExecElapseT,
12                                 tMinInstrExecElapseT,
13                                 tMaxRoleExeT,
14                                 tMinRoleExeT
15                                }->union(
16                                   let res: Set(DT4BP!Resolution) =
17                                     s.bps->collect(e|e.resolution)->flatten()
18                                   in
19                                     res->iterate(e ;
20                                           excs: OrderedSet(TimedCAAFWrk!Exception) =
21                                                 OrderedSet{} | excs.append(e))),
22                 participants <- s.resources,
23                 predefinedTypes <- s.basicTypes,
24                 dttps <- s.dttps,
25                 caas <- s.bps
26         ),
27         tMaxInstrExecElapseT : TimedCAAFWrk!MaxInstrExecElapseT(
28                 name <- 'maxInstrExecElapseT'
29         ),
30         tMinInstrExecElapseT : TimedCAAFWrk!MinInstrExecElapseT(
31                 name <- 'minInstrExecElapseT'
32         ),
33         tMaxRoleExeT : TimedCAAFWrk!MaxRoleExecT(
34                 name <- 'maxRoleExecT'
35         ),
36         tMinRoleExeT : TimedCAAFWrk!MinRoleExecT(
37                 name <- 'minRoleExecT'
38         )
39 }
```

Excluding the evaluation of the expression required to initialise the target attribute *name*, the remaining evaluations lead to the triggering of different rules. The next rule to be presented here is related to the transformation of *business processes* into *caas*, which is triggered due to evaluation of the source attribute *bps* in order to initialise the target attribute *caas*. The transformations defined by the rules triggered when evaluating the source attributes *events, basicTypes* and *dttps* are straightforward, and, thus, are not considered further in this section.

5.2.2 Business Process

As previously mentioned, a business process is semantically defined as a CAA. The rule named *BusinessProcess*, which is shown in the Listing 5.4 defines the semantic mapping between these concepts. The target pattern of this rule specifies that a target element of type *CAA* is created (line 5) every time a source element of type *BusinessProcess* is found in the source model (line 3).

The binding part of the rule specifies that 1) the *name* (line 6), 2) the event that requests its execution (line 7), 3) the executing *period* (line 10), 4) the parameters (line 11) and 5) the data types (line 13) of the target element are initialised with the values held by the same attributes of the source element. Moreover, the target element will have the same position in the hierarchical

CAA structure as the source element in the business process structure. Hence, a *root* CAA is generated only by a *root* business process (line 14).

Listing 5.4: Rule BusinessProcess.

```
1  rule BusinessProcess {
2          from
3                  s : DT4BP!BusinessProcess
4          to
5                  t : TimedCAAFWrk!CAA(
6                          name <- s.name,
7                          requestedBy <- s.requestedBy,
8                          delay <- s.start,
9                          elapse <- s.last,
10                         period <- s.period,
11                         params <- s.params,
12                         roles <- s.participants,
13                         dttps <- s.dttps,
14                         isRoot <- s.isRoot
15                 )
16 }
```

Regarding the time-related information, the binding specifies that the *delay* and *elapse* attributes of the target element are initialised according to the *start* and *last* attributes of the source element (lines 8 and 9), respectively. Hence, the time boundaries related with the start and duration of a business process CAA are semantically defined as the CAA time boundaries that constrain its start and duration.

The binding part also specifies that *participants* enclosed within a business processes are mapped to *roles* of the CAA. This semantic mapping between a business process participant and a CAA role is due to their having similar structural properties. Both define a grouping are for a set of tasks that are required to be executed by a same entity. In the *DT4BP* domain, this entity is the *resource* assigned to the business process participant, whereas in the semantic domain (i.e. TimedCaaFWrk) the entity is the *participant* in charge of playing the role. The following rule specifies this semantic mapping in detail.

5.2.3 Participant

First, recall that participants that compose a business process are assumed to be ordered. This is formally specified in the *DT4BP* meta-model (see Appendix B).

The specification of the semantic mapping between a business process *participant* and a CAA *role* comprises two different rules: 1) *ParticipantFirst*, that corresponds to the first participant enclosed within the business process; 2) *ParticipantNonFirst*) for the other participants enclosed within the same business process. This section focuses on the rule *ParticipantFirst* since it is an extended version of the other rule.

The target pattern of the rule *ParticipantFirst* (shown in the Listing 5.6) specifies that a target element of type *Role* is created (line 5) every time a source element of type *Participant* is found in the source model and this source element is the first in the ordered composite relationship *participants* (line 3). While the type matching is specified in the source pattern, the ATL helper named *isFirstParticipant* (shown in the Listing 5.5) defines a predicate that checks whether the current source element of type *Participant* is the first or not. Therefore, the triggering of the rule requires that the source element satisfiies both the type matching and predicate defined by the helper.

Listing 5.5: Helper for *Participant*.

```
1 helper context DT4BP!Participant def: isFirstParticipant(): Boolean =
2         let
3                 first: DT4BP!Participant = self.bp.participants->first()
4         in
5                 self = first
6         ;
```

The binding part of the rule specifies that the name of the target element is initialised with the same name as the source element (line 6). The enclosing CAA of the role is initialised with the enclosing business process of the participant that led to the rule to being triggered (line 7). The pre-and post-condition of the business process that encloses the source element are used to initialise the pre- and post-condition of the target element (line 8 and 11), respectively.

The way in which the pre- and post conditions of a business process are semantically mapped over the pre- and post-conditions of a CAA is the key component that led to splitting the mapping into two mutually exclusive rules. Thus, this mapping deserves further explanation. As was discussed in Chapter 4 Section 4.2.1, the pre(post)-condition of a CAA is defined as the conjunction of the pre(post)-conditions owned by each enclosed role. Since the pre(post)-condition of the business process is semantically defined as the pre(post)-condition of a CAA, the mapping between the former over the latter implies selecting one single role where the pre(post)-conditions of the business process are set. In this manner, the pre(post)-condition of a business process *bp* is semantically mapped as the pre(post)-condition of a role r_i enclosed within a CAA *caa* such that *caa* is the CAA to which *bp* has been semantically mapped. Any role r_i enclosed within *caa* can be select to fullfil this requirement. The semantic mapping defines the first role created during the model transformation as the role where the pre(post)-condition of the business process is placed. Since a role is created in the target model every time a participant element is found in the source element, it is important to discriminate whether the participant is the first or not. In the case that it is the first, the both the pre- and the post-condition have to be mapped as this is specified in the binding part of the rule. Otherwise, the pre- and post-condition of the business process do not have to be considered in the semantic mapping between a participant and a role. This situation is specified by the rule *ParticipantNonFirst*.

The remaining part of the binding specifies that the statements (line 9) enclosed within the source element (of type *Participant*) are used to initialise the instructions of the target element (of type *Role*). In addition, the minimum and maximum allowed working times (line 10) of the source element are used to initialise the maximum and minimum allowed execution time of the target element. The *bp-handlers*[4] of a particular business process *bp* are semantically defined as the collection of *cooperative handlers* enclosed within the CAA *caa* that semantically define *bp*. Hence, the attribute *handlers* of a role target element is initialised by the collection of cooperative handlers associated with the participant source element (lines 13-15).

Listing 5.6: Rule ParticipantFirst.

```
1 rule ParticipantFirst {
2   from
3       s : DT4BP!Participant(s.isFirstParticipant())
4   to
5       t : TimedCAAFWrk!Role(
6           name <- s.name,
7           caa <- s.bp,
8           pre <- s.bp.pre,
```

[4] See Chapter 3, Section 3.4.4.3 for details about *bp-handlers* and *p-handlers*.

```
9           instrs <- s.stmts,
10          exec <- s.workFor,
11          post <- s.bp.post,
12
13          handlers <- s.bp.cooperativeHandlers->
14                      collect(e|e.hndParticipants)->
15                        flatten()-> select(e|e.participant=s),
16
17          handlersExecT <- if (not s.workFor.oclIsUndefined()) then
18                             OrderedSet {s.workFor.handlers}
19                           else
20                             OrderedSet {}
21                           endif
22      )
23
24  }
```

The time constraints concerning the minimum and maximum allowed working time of a participant may have up to two associated *p-handlers* aimed at dealing with the absence of any of these time constraints. This kind of handler is semantically defined as a *role local handler* within the semantic domain. In the case that there exists at least a *p-handler* associated with the *workFor* time constraint (lines 17-21), then it is used to initialise the *role local handler* that will occur at run-time when any of the role execution time constraints is absent.

The next sections provide details about the semantic mapping of *DT4BP* statements, deviations and exception handling both at the level of the business process, as well as at the level of the participant.

5.2.4 Statements

The *statements* enclosed within a business process are semantically defined as the *instructions* of a role. The set of statements a participant may contain are classified as *control, execution* and *declaration*. Statements such as *If, Repeat* and *While* belong to the first type of statement, whereas *Receive, Send* and *Activity* belong to the second. The only statement that forms the third kind of statement is *ObjDecl*. For each statement, regardless of its classification, a transformation rule is provided to specify its semantic definition in terms of instructions belonging to the *Timed-CaaFWrk* semantic domain. Among all these transformation rules, the one that provides the semantic definition for the *Atomic* statement is shown since it is probably the most complex among all statement-related rules. Furthermore, the description of this rule helps in understanding other rules like those that provide the semantic definition for *Composite* and *Nested* statements.

5.2.4.1 Atomic

The semantic definition of the statement *Atomic* comprises two rules. One rule that applies over *atomic* activities that hold a post-condition, and other for those activities that do not. Depending on whether an *atomic* activity holds a post-condition or not, a different semantic definition is achieved in terms of instructions belonging to the *Timed-CaaFWrk* semantic domain.

The rule named *AtomicWithPost*, which is shown in the Listing 5.7, specifies the semantic definition for *atomic* activities that hold a post-condition. The target pattern of the rule specifies that a target element of type *Block* (line 5) is created every time a source element of type *Atomic* with a post-condition is found in the source model (line 3). The target element being created is

composed of an ordered set of two instructions (line 6) such that the first one is a target element of type *Execute*, and the second of type *If* (lines 9 and 26, respectively).

Listing 5.7: Rule AtomicWithPost.

```
1  rule AtomicWithPost {
2    from
3            s : DT4BP!Atomic(not s.post.oclIsUndefined())
4    to
5            t : TimedCAAFWrk!Block(
6                    instrs <- OrderedSet {t2,t3},
7                    isTry <- false
8            ),
9            t2 : TimedCAAFWrk!Execute(
10                   operation <- s.name,
11                   args <- s.args-> collect (e | e.obj),
12                   delay <- s.in,
13                   deadline <- s.within,
14                   pre <- if(not s.pre.oclIsUndefined()) then
15                              s.pre.predicate
16                           else
17                              'true'
18                           endif,
19                   post <- s.post.predicate,
20                   handlers <- if (not s.within.oclIsUndefined()) then
21                                 OrderedSet {s.localHandlers, s.within.handlers}
22                               else
23                                 OrderedSet {s.localHandlers}
24                               endif
25           ),
26           t3 : TimedCAAFWrk!If(
27                   cond <- 'not('.concat(s.post.predicate.toString().concat(')')),
28                   then <- OrderedSet {s.deviations,t4}
29           ),
30           t4 : TimedCAAFWrk!Execute(
31                   operation <- 'System.exit()',
32                   pre <- 'true',
33                   post <- 'true'
34           )
35   }
```

The binding part of the rule *t2* specifies how the *execute* target element attributes are initialised (lines 10-25). This binding specifies that the *operation* attribute is initialised with the name of the source element (line 10), whereas the *arguments* (line 11), *pre-condition* (lines 14-18), and *post-condition* (line 19) attributes of the source element are used to initialise the attributes of the same name in the target element. The time-related attributes *in* and *within*, which correspond to the delay and deadline boundaries of the *atomic* activity are used to initialise the *delay* and *deadline* attributes of the target element (lines 12 and 13, respectively). An *atomic* activity may have one or more *p-handlers* associated with it, which are meant to deal with certain deviations that may occur during its execution. It must be noted that among these potential deviations, those that may arise due to a missing time constraint set over the activity are also included. These *p-handlers* are used to initialise the *role local handlers* of the execute target element (lines 20-24).

The rule *t3* upon calling, creates a target element of type *If*, which is aimed at defining the semantic of a deviation that occurs due to the violation of the post-condition. The reason why it is important to know whether the *atomic* activity holds a post-condition not, because those that do not are assumed to always execute without deviating from their expected behaviour. The binding part of this rule specifies that the *condition* attribute of the *if* target element is initialised as the negation of the post-condition of the source element (line 27), and the *then*

attribute is initialised as an ordered set of instructions. The last instruction in this ordered set is generated by calling the rule *t4*. This role generates an *execute* instruction in the target aimed at interrupting the execution of the role. This instruction is executed only when an exception has been raised for which there do not exist any handler. This situation, at the *DT4BP* domain represents the occurrence of an unexpected event that leads towards the failure of the business process, and results in its abrupt interruption.

An *atomic* activity that does not hold a post-condition is one that is assumed to always execute without deviating from its expected behaviour. The semantic definition of this kind of *atomic* activity is given by the rule *AtomicWithoutPost*, which is shown in the Listing 5.8.

Listing 5.8: Rule AtomicWithoutPost.

```
1  rule AtomicWithoutPost {
2    from
3         s : DT4BP!Atomic(s.post.oclIsUndefined())
4    to
5         t : TimedCAAFWrk!Execute(
6              operation <- s.name,
7              args <- s.args-> collect (e | e.obj),
8              delay <- s.in,
9              deadline <- s.within,
10             pre <- if(not s.pre.oclIsUndefined()) then
11                        s.pre.predicate
12                     else
13                        'true'
14                     endif,
15             post <- 'true',
16             handlers <- if (not s.within.oclIsUndefined()) then
17                             OrderedSet {s.localHandlers, s.within.handlers}
18                         else
19                             OrderedSet {s.localHandlers}
20                         endif
21        )
22 }
```

The target pattern of the rule specifies that an element of type *Execute* is created in the target model (line 5) every time an element of type *Atomic* that does not hold a post-condition is found in the source model (line 3). The binding part of this rule specifies the same semantic mapping as the rule *t2* shown in the Listing 5.7. Since this semantic mapping was already explained when describing the rule *AtomicWithPos*, no further description of the rule *AtomicWithoutPos* is required.

5.2.5 Deviations

A *deviation*, according to the *DT4BP* meta-model, is defined as either an *ActivityDeviation* or a *TimeDeviation*. An *ActivityDeviation* is used to specify the potential deviation of an *Activity* with respect to its expected behaviour. This kind of deviation is specified by giving the condition that determines whether the *Activity* to which it is associated has deviated or not from its expected behaviour, and (2) the exception to be raised in case the deviation takes place.

On the other hand, a *TimeDeviation* is used to specify the potential absence of a timing constraint. This kind of deviation is specified by signaling that the time bound may be only missing, since the exception to be raised flag its occurrence is always the same: *timeout*. Hence, as for a *TimedDeviation* none condition or exception form part of its description. Its semantic definition is given in terms of a handler (placed either at the level of the business process or at the level

of the participant) meant to handle such an exception. As a result of this fact, this section only shows details about the semantic definition of an *ActivityDeviation*.

The semantic definition of an *ActivityDeviation* is given by the rule of the same name, which is shown in the Listing 5.9. The target pattern of the rule specifies that a target element of type *If* (line 5) is created every time a source element of type *ActivityDeviation* is found in the source model (line 3). The initialisation of the *condition* attribute of the target element depends on the type of condition held by the source element. As specified in the *DT4BP* meta-model, the condition of an *ActivityDeviation* may be of type *Predicate, AbortDeviation, FailureDeviation* or *DataDurationDeviation*.

Listing 5.9: Rule ActivityDeviation.

```
1  rule ActivityDeviation {
2    from
3          s : DT4BP!ActivityDeviation
4    to
5          t : TimedCAAFWrk!If(
6                cond <- if s.condition.oclIsTypeOf(DT4BP!Predicate) then
7                              s.condition.predicate.toString()
8                        else
9                              if s.condition.oclIsTypeOf(DT4BP!AbortDeviation) then
10                                   'outcome=aborted'
11                             else
12                                   if s.condition.oclIsTypeOf(DT4BP!FailureDeviation) then
13                                         'outcome=failed'
14                                   else
15                                         'dataExpired'
16                                   endif
17                             endif
18                       endif,
19               then <- t2
20         ),
21         t2 : TimedCAAFWrk!Raise(
22               exception <- s.raise
23         )
24 }
```

The binding part of the rule specifies that in the case when the *condition* attribute of the source element is of type *Predicate* its value is used to initialise the *condition* attribute of the *if* target element (lines 6-7). Otherwise, it is initialised either with the value *outcome=aborted* (lines 9-10), *outcome=failed* (lines 12-13), or *dataExpired* (line 15) in the cases where the type of the *condition* attribute is either *AbortDeviation, FailureDeviation* or *DataDurationDeviation*, respectively.

Initialising the *condition* attribute of the target element with the value *outcome=aborted* or *outcome=failed* may only be possible if the source element is an *Activity* of type *Composite* or *Nested* as they are the only types of activities that may have deviations with these kinds of conditions. Conversely, to initialise the *condition* attribute of the target element with the value *dataExpired* it is possible when the source element is of type *Atomic* only as it is the unique place from where a data element may be accessed (regardless it holds a temporal constraint or not). Conversely, initialising the *condition* attribute of the target element with the value *dataExpired* is possible when the source element is of type *Atomic* only as this is the unique place from which a data element may be accessed (regardless of whether it holds a temporal constraint or not).

The initialisation of the *then* attribute of the target element (of type If), includes a called rule named *t2* (line 21). This rule specifies that a target element of type *Raise* is created in the target model every time it is called (line 21). The binding part of rule *t2* specifies that the attribute

exception of this target element (of type *Raise*) is initialised with the value of the attribute *raise* owned by the source element (line 22)

In summary, a (*DT4BP*) *deviation* is semantically defined as an (*Timed-CaaFWrk*) *If-then* statement such that the *then* part makes use of the *Raise* instruction as a means to signal the deviation of the business process during its execution. The remaining part of this section describes the rules that semantically define the *DT4BP* handlers meant to deal with every potential deviation included within the business process definition.

5.2.6 Exception Handling

The occurrence of a deviation during the execution of a business process is signaled by raising an exception. For each potential deviation being considered in the business process definition, a *handler* aimed at dealing with the exception must be included within the business process definition. As previously mentioned, there exist two kinds of handlers for dealing with exceptions: *business process handlers*, which are referred to as *bp-handlers*; and *participant handlers* referred to as *p-handlers*. The former involves every single participant enclosed within the business process in the handling process, whereas the latter is defined within the context of a particular participant involving one or more of its statements. More detail regarding these kinds of handlers can be found in Chapter 3, Section 3.4.4.3. The rules explained in this section are those that address the semantic definition of bp-handlers and *p-handlers*.

5.2.6.1 Business Process Exception Handling

The semantic definition of a *bp-handler* is divided into two mutually exclusive rules depending on whether the exception the handler is aimed at dealing with is of type *Abort* or not. The rule that provides the semantic definition for the for the *bp-handlers* that deal with *non-Abort* exceptions is named *HandlerParticipant*. This rules is shown in the Listing 5.10.

The target pattern of the rule specifies that a target element of type *CooperativeH* (line 5) is created every time a source element of type *HandlerParticipant* that does not deal with an *Abort* exception is found in the source model (line 3). The ATL helper named *dealsWithAbort* (shown in the Listing 5.11) defines the predicate that checks whether the current source element deals with an exception of type *Abort* or not (line 3). The way in which this helper determines whether a *HandlerParticipant* deals or not with an *Abort* exception deserves further explanation.

According to the *DT4BP* meta-model, a *BPHandler* is defined as a collection of one or more *handler participants* (i.e. attribute *hndParticipants* in the meta-model) and one single *outcome*. Every *BPHandler* that belongs to a certain business process *bp*, is expected to be placed at the right part of the *resolution* element of the business process *bp*. The case in which a business process may potentially be faced with an *Abort* implies the existence of a *BPHandler* to deal with such an exception. Thus, the business process definition must hold a *Resolution* that binds an *Abort* exception to the *BPHandler* element intended for dealing with such an exception.

In this manner, to know whether a particular *HandlerParticipant* is part of a *BPHandler* aimed at dealing with an *Abort* exception requires inspection of the left side part of the resolution element to which the *BPHandler* is associated. In the helper definition, the *res* element defined in the *let* part (line 3, in Listing 5.11) is defined to gather all the *resolution* elements of the *BPHandler* that enclose the current *HandlerParticipant* source element. The *in* part of the

helper (line 6) checks whether there exist at least one *resolution* element in the *res* set, whose type is *Abort*[5]

Listing 5.10: Rule HandlerParticipant.

```
1  rule HandlerParticipant {
2   from
3    s : DT4BP!HandlerParticipant(not s.dealsWithAbort())
4   to
5    t : TimedCAAFWrk!CooperativeH(
6        name <- s.actsIn.name,
7        role <- s.participant,
8        outcome <- s.actsIn.outcome,
9        instrs <- s.stmts,
10       handles <- let res: Set(DT4BP!Resolution) =
11                    s.actsIn.refImmediateComposite().resolution->
12                                          select(r|r.right = s.actsIn)
13                  in
14                  res->iterate(e; excs:OrderedSet(TimedCAAFWrk!Exception) = OrderedSet{}|
15                            if(res->exists(ex | ex.left.size() > 1)) then
16                                    excs.append(thisModule.resolveTemp(e, 't'))
17                            else
18                                    excs.append(res->collect(ex2| ex2.left))
19                            endif),
20       params <- s.participant.bp.params->
21                      union(s.participant.stmts->
22                          select(e|e.oclIsTypeOf(DT4BP!ObjDecl))->collect(e|e.var))
23    )
24 }
```

Listing 5.11: Helper for *HandlerParticipant*.

```
1  helper context DT4BP!HandlerParticipant def: dealsWithAbort(): Boolean =
2    let
3        res: Set(DT4BP!Resolution)=self.actsIn.refImmediateComposite().resolution->
4                        select(r|r.right = self.actsIn)
5    in
6        res-> exists(r | r.left-> exists(ex|ex.oclIsTypeOf(DT4BP!Abort)));
```

The binding part of the rule *HandlerParticipant* specifies that the attribute *name* of the target element is initialised with the *name* of the *BPHandler* referred to by the source element (line 6). This *BPHandler* element is used also to initialise the *outcome* attribute (line 8). The source element refers to a *participant* element, which is used to initialise the *role* attribute of the target element (line 7). This *participant* element is also used to reach the *parameters* of the business process that enclose the source element. These parameters along with the values gathered from inspecting the variable *var* held by each instruction *ObjDecl* and contained by the source element (lines 21-22) are used to initialise the attribute *params* of the target element (line 20). The *statements* contained by the source element are used to initialised the attribute *instrs* of the target element.

The initialization of the attribute *handles* of the target element deserves a detailed explanation (lines 10-19). This attribute is expected to hold the exceptions the target element (of type *CooperativeH*) has to deal with. Since a *HandlerParticipant* is semantically defined as a *CooperativeH*, then exceptions being handled by the former must be mapped to the exceptions the latest handles. The exceptions handled by a *HandlerParticipant hndPrt* are those placed in the

[5] In theory, for all the *resolution* elements contained in the set *res*, the type of the element placed on its left side part must be the same, since all these *resolution* elements refer to the same *BPHandler*.

left hand side of every *resolution* element, whose right hand side refers to the *BPHandler* that encloses *hndPrt*.

Finding these exceptions is achieved by first gathering all the *resolution* elements of the *BPHandler* that encloses the current *HandlerParticipant* source element. Notice that this results in a set of *resolution* elements named *res*, in the same manner as the helper *dealsWithAbort* previously explained does (lines 11-12). Once these *resolution* elements are obtained, the next step consists of extracting the exception that each of them holds in its left hand side. There may exist more than once exception on the left hand side of a *resolution* element. In case the number of exceptions a *resolution* element holds on its left hand side part is not higher than one (line 15), this exception is added to the ordered set of exceptions named *excs* (line 18). This set is used to initialised the *handles* attribute of the target element. Otherwise, the exception to be added to the set *excs* is the one that semantically corresponds to the concurrent flagging of the exceptions placed at the left hand side of *resolution* element that has to be created (line 16). Concurrent exceptions are generated by the rule *ConcurrentExceptions*, which is shown in the Listing 5.12.

Listing 5.12: Rule ConcurrentExceptions.

```
1  rule ConcurrentExceptions {
2    from
3        s: DT4BP!Resolution (s.left->size() > 1)
4    to
5        t: TimedCAAFWrk!Exception (
6                name <- s.left->
7                        iterate(e; res : String = 'CC_EX' |
8                          res.concat('_'.concat(
9                                      e.name.toString().regexReplaceAll('EX_|CC_EX_','')
10                                   )
11                               )
12                           ),
13               madeOf <- s.left
14       )
15 }
```

Notice that the rule *ConcurrentExceptions* is triggered every time a *resolution* source element that holds more than one exception on its left hand side is found (line 3). Its execution generates an element of type *Exception* in the target model (line 5). The binding part of this rule specifies that the *name* attribute of the target model is initialised with the value that results from concatenating the string *"CC_EX"* with the names of all exceptions placed on the left hand side of the *resolution* source element (lines 7-12), whereas the *madeOf* attribute is initialised with the exact value held by the left hand side of the *resolution* source element (line 13).

5.2.6.2 Participant Exception Handling

A *p-handler* is aimed at dealing with a deviation that may arise during the execution of a statement enclosed within certain participant. Whether the participant may complete its activities as originally planned depends on the success of the *p-handler* in dealing with the deviation. Not all statements enclosed within a participant may be confronted with a deviation during its execution. According to the *DT4BP* meta-model, only statements that extend from the type *Execution* will be faced with a deviation. Moreover, since a *Deviation* is refined between *TimeDeviation* and *ActivityDeviation*, the statement *Within* may only confront the former, whereas statements extending from *Activity* may only confront the latter.

The semantic definition of a *p-handler* is divided into two mutually exclusive rules depending on whether they are associated with a statement *Activity* or *Within*. The rule that semantically defines a *p-handler* associated with an *Activity* statement is detailed first. The rule related with *Within* statements is given subsequently.

For a *p-handler* associated with a statement *Activity*, its semantic definition is given by the rule *PHandlerForActivity*. This rule is shown in the Listing 5.13. The target pattern of the rule specifies that a target element of type *LocalH* is created every time a source element of type *PHandler* enclosed in an *Activity* statement (line 5) is found in the source model. The binding part of the rule specifies that the attribute *instrs* of the target element is initialised with the statements owned by the source element (line 6), whereas the attribute *handles* is initialised with the exceptions raised by the deviation it is associated with (line 7). It must be noted that for an *Activity* statement, the deviation is of type *ActivityDeviation*.

Listing 5.13: Rule PHandlerForActivity.

```
1 rule PHandlerForActivity {
2   from
3         s : DT4BP!PHandler(s.refImmediateComposite().oclIsKindOf(DT4BP!Activity))
4   to
5         t : TimedCAAFWrk!LocalH(
6               instrs <- s.stmts,
7               handles <- s.handles->collect(dev|dev.raise)
8         )
9 }
```

On the other hand, for a *p-handler* associated with a statement *Within*, its semantic definition is given by the rule *PHandlerForWithin*. This rule is shown in the Listing 5.14. The target pattern of the rule specifies that a target element of type *LocalH* is created every time a source element of type *PHandler* enclosed in an *Within* statement (line 5) is found in the source model. The binding part of the rule specifies that the attribute *instrs* of the target element is initialised with the statement owned by the source element in its attribute *stmts*, whereas the initialisation of the attribute *handles* depends on the kind of time-related deviation the source element is designed to handle.

Hence, for a source element that is aimed at handling a time-related deviation that is raised when a time bound is missed (regardless of whether it is the lower or upper time bound[6]), the attribute *handles* is initialised with an ordered set (line 9) that contains an instance of type *MinInstrExecFinishT* (lines 9-12) and an instance of type *MaxInstrExecFinishT* (lines 13-16). Otherwise, the source element is aimed at handling a time-related deviation that is raised when either the lower time bound is missed or the upper time bound is missed. In the former case, the attribute *handles* is initialised with an instance of type *MinInstrExecFinishT* (lines 20-23), whereas in the latter case with an instance of type *MaxInstrExecFinishT* (lines 24-27).

Listing 5.14: Rule PHandlerForWithin.

```
1 rule PHandlerForWithin {
2   from
3         s : DT4BP!PHandler(s.refImmediateComposite().oclIsTypeOf(DT4BP!Within))
4   to
5         t : TimedCAAFWrk!LocalH(
6               instrs <- s.stmts,
```

[6] The attribute *kind* of the deviation element is *"min_max"* (line 8).

```
7               handles <- if(s.refImmediateComposite().timeout
8                                 -> exists(timeDev | timeDev.kind.toString() = 'min_max'))
9                           then OrderedSet {TimedCAAFWrk!MinInstrExecFinishT.
10                                           allInstances()
11                                           -> asSequence()
12                                           -> first(),
13                                          TimedCAAFWrk!MaxInstrExecFinishT.
14                                           allInstances()
15                                           -> asSequence()
16                                           -> first()}
17                          else
18                                if(s.refImmediateComposite().timeout
19                                    -> exists(timeDev | timeDev.kind.toString() = 'min'))
20                                then TimedCAAFWrk!MinInstrExecFinishT.
21                                           allInstances()
22                                           -> asSequence()
23                                           -> first()
24                                else TimedCAAFWrk!MaxInstrExecFinishT.
25                                           allInstances()
26                                           -> asSequence()
27                                           -> first()
28                                endif
29                           endif
30          )
31  }
```

5.3 Validation of DT4BP models: putting it all together

In the introductory chapter (see Chapter 1, Section 1.1) it was stated that one of the aims of this thesis was to provide, in addition to a new modelling language oriented toward DCTC business process, a means to allow modellers check the correctness of the business model with respect to stakeholder's expectations. Thus, the goal is to develop the *validation process* that allows the programmer to determine whether a particular business process model (or process definition) *is the right model* from the stakeholder's viewpoint. One way to achieve this objective is by observing the dynamic behaviour of a particular business process model. This approach is known as *validation by simulation*.

The behaviour of a particular *DT4BP* model is achieved by exploiting the executable semantics owned by *Timed-CaaFWrk* (the semantic domain of *DT4BP* modelling language). As explained in Chapter 4, the execution of a *DT4BP* model M_{DT4BP} involves the M2M transformation (used to specify the semantic mapping between *DT4BP* and *Timed-CaaFWrk*), and the M2T transformation (used to generate Java source code from a given *Timed-CaaFWrk*-compliant model). These two transformations are chained (see Figure 5.1) such that, given a particular M_{DT4BP} model, it is possible to automatically obtain its representation in terms of a set of Java source code files.

Relying on an existing Java compiler, each Java source file belonging to this set can be transformed into an executable Java file, i.e. *.class* file. From the MDE's viewpoint, a compiler is considered as a text-to-text (T2T) transformation. In this manner, and in line with the MDE perspective, both the source model M_{Java}, the Java source file, and the target model $M_{JavaBytecode}$, the Java binary file, are considered as compliant models with respect to the Java programming language meta-model and the Java Virtual Machine meta-model, respectively.

The execution of the Java binary files results in a set of different traces, which can be considered as the different behaviours allowed by the model. Each trace represents a different *valid* execution specified by the M_{DT4BP} model. Hence, the existence of a single trace showing an undesired

behaviour, e.g. the wrong execution order of two activities, or non-execution of an activity, is sufficient to conclude the model has defects. Thus, validation by simulation allows the modeller (with the stakeholder) to reveal defects in the model in the early stages of the development of the software system to support the business process. However, this validation process does not allow the modeller to absolutely establish that the model is free of defects.

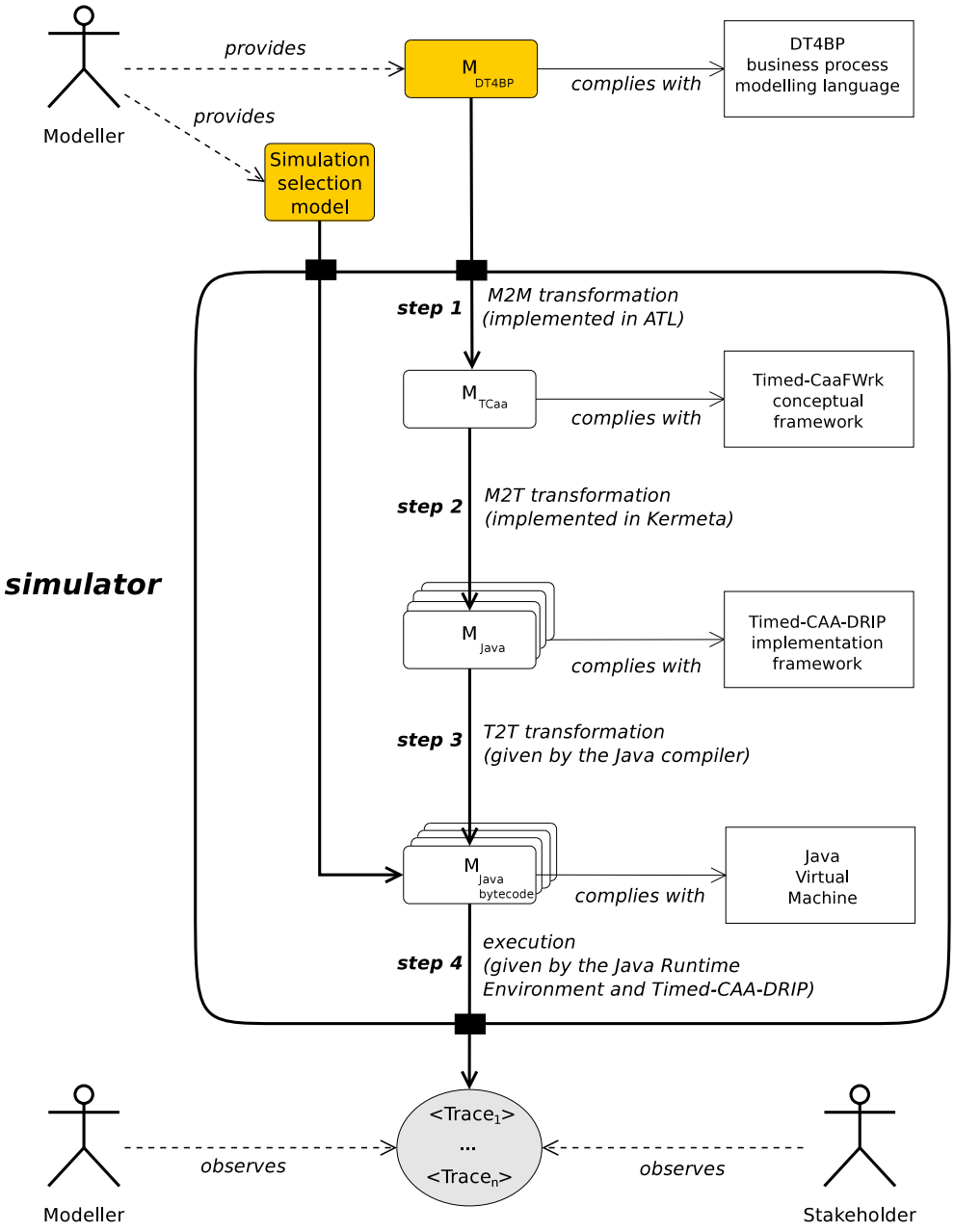

Fig. 5.1: Validation process overview.

This validation approach requires as input the particular *DT4BP* model to be simulated, but also some additional information that bounds the domain of interest of the model. This information is specified in the so-called *Simulation Selection* model. Both input models, denoted as orange squares on Figure 5.1, are expected to be provided by the modeller.

Every *Simulation Selection* model is created to validate the M_{DT4BP} model with respect to a certain viewpoint. In this manner, the overall validation of a model involves many different *Simulation Selection* models. However, every time the M_{DT4BP} model is altered, either because a mistake has been detected or the stakeholder has requested a new requirement, the validation process must be restarted from the very beginning. In other words, every single Simulation Selection model has to be (re-)inspected to confirm that it is still consistent with respect to the new *DT4BP* model being validated. Then this new *DT4BP* model must be simulated using each of these *Simulation Selection* models.

The different components that have been developed to build a tool (referred to as *simulator*) for validating *DT4BP* models include:

- to provide the elements that comprise the definition of the *DT4BP* modelling language such as the M2M transformation (explained in the Section 5.2 of this Chapter), which specifies the *semantic mapping* of the language;
- to extend and enhance the *Timed-CaaFWrk* conceptual framework, chosen *semantic domain* of the *DT4BP* modelling language, like the M2T transformation and *Timed-CAA-DRIP* implementation framework (explained in Chapter 3, Sections 4.5 and 4.4, respectively)

This simulator, then, allows modellers to be abstracted from the internal processes that generate the traces from a given *DT4BP* and a series of *Simulation Selection* models.

5.3.1 The validation process in practice

The *patient diagnosis* business process, the running example presented in Chapter 3, Section 3.3, is used here to demonstrate the validation process.

As shown in Figure 5.1, the simulator uses the *DT4BP* model of the *patient diagnosis* business process to perform its first internal step. This consists of transforming the input model into a *Timed-CaaFWrk*-compliant model. This resulting model then is used as input to the second internal step that leads to the generation of the Java source files. As described in the figure, these Java source files adhere to the *Timed-CAA-DRIP* implementation framework.

The complete list of Java source files[7] generated for the *patient diagnosis* model are shown in Table 5.3.1. These Java source files are grouped into different packages according to the concern they address. Moreover, Java source files implementing a CAA are subdivided into different packages according to the kind of behaviour, i.e. *normal, forward error recovery*, and *backward error recovery*, each file addresses. Hence, Java source files implementing the *roles*, i.e. *normal* behaviour, are grouped into one package. The *handlers*, i.e. *forward error recovery*, and *compensators*, i.e. *backward error recovery*, are grouped into separate packages.

The compilation and execution of these Java source files (steps 3 and 4 of the internal process, respectively) produce a set of traces. Each trace, as previously discussed, represents a valid behaviour that is allowed by the *patient diagnosis* model in accordance with the constraints specified in the *Simulation Selection* model. The Listing 5.15 shows four of the traces generated by the simulator for the *patient diagnosis* business process using the *Simulation Selection* model (partially) described in Listing 5.16.

[7] Files related to the simulation of the *patient diagnosis* business process are available for downloading at http://wiki.lassy.uni.lu/Projects/DT4BP.

(a) Files corresponding to the CAAs.

	Role	Handler	Compensator
package	*caa.diagnosis*	*caa.diagnosis.handlers*	
	DiagnosisUnit	EvacuationDiagnosisUnit	
package	*caa.registration*	*caa.registration.handlers*	*caa.registration.compensators*
	Patient	FillSpecialFormPatient	AdminOfficePatient
	Secretary	FillSpecialFormSecretary	AdminOfficeSecretary
package	*caa.makeDocument*		
	Nurse		
package	*caa.examination*	*caa.examination.handlers*	
	Assistant	CardiacUnitAssistant	
	Nurse	CardiacUnitNurse	
	Patient	CardiacUnitPatient	
		UndressAssistant	
		UndressNurse	
		UndressPatient	
		UndressANDCardiacUnitAssistant	
		UndressANDCardiacUnitNurse	
		UndressANDCardiacUnitPatient	
package	*caa.consultation*	*caa.consultation.handlers*	*caa.consultation.compensators*
	Doctor	HospitalAdmissionDoctort	
	Patient	HospitalAdmissionPatient	
package	*caa.giveInformation*		
	Nurse		
	Patient		

(b) Abstract Data Types and Exceptions used by the CAAs.

	Abstract Data Type	Exception
package	*datatypes*	*exceptions*
	BloodPressure	CC_EX_HighBP_HighTemp
	Calendar	EX_Fire
	DBoolean	EX_HighBP
	DFloat	EX_MalfunctionBPMonitor
	Diagnosis	EX_NotSSCard
	DInteger	EX_Admission
	DString	EX_Foreigner
	FireAlarm	EX_HighTemp
	ForeignerForm	EX_MalfunctionThermometer
	HealthStatus	EX_UndefHx
	Hospital	
	MedicalHistory	
	Medicine	
	PatientSheet	
	Person	
	Prescription	
	SocialSecurityCard	
	Temperature	
	Treatment	

Tab. 5.3.1: Generated Java source files.

The first trace corresponds to the information that appears in Listing part with white background. The second trace is described by adding to the first trace the information that appears in the Listing part with grey background. The third trace is described by adding the information shown in the Listing part with yellow background. The fourth trace is the defined by the sequential composition of the different Listings. It is worth noting that the first trace describes a behaviour where the model is not confronted with a deviation. Conversely, the second and third traces describe the simulation of the *patient diagnosis* business process when it is confronted with the deviations arrising from the exceptions *EX_Foreigner* and *EX_HighBP*, respectively. The fourth trace describes the simulation in which the business process is confronted with both deviations during its execution.

Listing 5.15: *Diagnosis business process* trace.

```
------ DIAGNOSIS ------
ALLOCATION >> Resource 'DU1' has been assigned to execute the participant 'DiagnosisUnit'
in BP 'Diagnosis'

Diagnosis.DiagnosisUnit[DU1]>> COMPOSITE CALL BP 'Registration'

------ REGISTRATION ------

ALLOCATION >> Resource 'Ann' has been assigned to execute the participant 'Secretary'
              in BP 'Registration'
ALLOCATION >> Resource 'Alfredo' has been assigned to execute the participant 'Patient'
              in BP 'Registration'
      Registration.Patient[Alfredo]>> Message 'reqSSCard' received from 'Secretary'
      Registration.Secretary[Ann]>> Message 'reqSSCard' sent to 'Patient'
      Registration.Patient[Alfredo]>> Executing 'searchSScard'
      Registration.Patient[Alfredo]>> Message 'ssCard' sent to 'Secretary'
      Registration.Secretary[Ann]>> Message 'ssCard' received from 'Patient'
      Registration.Secretary[Ann]>> Executing 'checkSSCard'
```

Trace 2

```
Deviation detected in 'checkSSCard', raising exception 'EX_Foreigner'
      Registration.Patient[Alfredo]>> Message 'ssCard' sent to 'Secretary'
      Participant 'Patient' was interrrupted!
      Registration.fillSpecialFormSecretary[Ann]>> Executing 'createForeignerForm'
      Registration.fillSpecialFormSecretary[Ann]>> Message 'askFillForm' sent to 'Patient'
      Registration.fillSpecialFormPatient[Alfredo]>> Message 'askFillForm' received from
                                                     'Secretary'
      Registration.fillSpecialFormPatient[Alfredo]>> Executing 'fillOutForm'
      Registration.fillSpecialFormSecretary[Ann]>> Message 'filledForm' received from
                                                   'Patient'
      Registration.fillSpecialFormSecretary[Ann]>> Executing 'registerPerson'
      Registration.fillSpecialFormPatient[Alfredo]>> Message 'filledForm' sent to
                                                     'Secretary'
```

Trace 1 - cnt'd

```
      Registration.Secretary[Ann]>> Executing 'getPersonAddress'
      Registration.Patient[Alfredo]>> Message 'reqAddress_and_City' received from
      'Secretary'
      Registration.Patient[Alfredo]>> Message 'p_address' sent to 'Secretary'
      Registration.Patient[Alfredo]>> Message 'p_city' sent to 'Secretary'
      Registration.Secretary[Ann]>> Message 'reqAddress_and_City' sent to 'Patient'
      Registration.Secretary[Ann]>> Message 'p_address' received from 'Patient'
      Registration.Secretary[Ann]>> Message 'p_city' received from 'Patient'
      Registration.Secretary[Ann]>> Executing 'validatePerson'
      Registration.Secretary[Ann]>> Executing 'registerPerson'
ALLOCATION >> The resource 'Ann' is free now
ALLOCATION >> The resource 'Alfredo' is free now

The BP 'Registration' has finished normally
```

```
———————— Diagnosis.DiagnosisUnit[DU1]>> STARTING SPLIT MODE ————————

Diagnosis.DiagnosisUnit[DU1]>> COMPOSITE CALL BP 'Examination'

Diagnosis.DiagnosisUnit[DU1]>> COMPOSITE CALL BP 'MakeDocument'

——— EXAMINATION ———
ALLOCATION >> Resource 'Alfredo' has been assigned to execute the participant 'Patient'
              in BP 'Examination'
ALLOCATION >> Resource 'Sue' has been assigned to execute the participant 'Nurse'
              in BP 'Examination'
ALLOCATION >> Resource 'Jane' has been assigned to execute the participant 'Assistant'
              in BP 'Examination'

——— MAKEDOCUMENT ———
ALLOCATION >> Resource 'Rose' has been assigned to execute the participant 'Nurse'
              in BP 'MakeDocument'
     MakeDocument.Nurse[Rose]>> Executing 'getHxPatient'
     MakeDocument.Nurse[Rose]>> Executing 'makePatientSheet'
     Examination.Nurse[Sue]>> Message 'reqProblemExplanation' sent to 'Patient'
     Examination.Patient[Alfredo]>> Message 'reqProblemExplanation' received from 'Nurse'
     Examination.Patient[Alfredo]>> Executing 'explainWhatIsWrong'

———————— Examination.Patient[Alfredo]>> STARTING SPAWN MODE ————————
     Examination.Patient[Alfredo]>> Message 'getTemp' received from 'Nurse'
     Examination.Nurse[Sue]>> Message 'getTemp' sent to 'Patient'
     Examination.Nurse[Sue]>> Executing 'checkTemp'
     Examination.Patient[Alfredo]>> Message 'getBP' received from 'Assistant'
     Examination.Assistant[Jane]>> Message 'getBP' sent to 'Patient'
     Examination.Assistant[Jane]>> Executing 'checkBP'
```

Trace 3

```
Deviation detected in 'checkBP', raising exception 'EX_HighBP'
     Examination.cardiacUnitAssistant[Jane]>> Message 'reqChangeRoom' sent to 'Patient'
     Examination.cardiacUnitAssistant[Jane]>> Message 'reqChangeRoom' sent to 'Nurse'
     Examination.cardiacUnitAssistant[Jane]>> Executing 'notifyNewRoomPatient'
     Examination.cardiacUnitPatient[Alfredo]>> Executing 'goToSpecialUnit'
     Examination.cardiacUnitNurse[Sue]>> Executing 'takePatientToSpecialUnit'
```

Trace 1 - cnt'd

```
ALLOCATION >> The resource 'Rose' is free now
ALLOCATION >> The resource 'Alfredo' is free now
ALLOCATION >> The resource 'Sue' is free now
ALLOCATION >> The resource 'Jane' is free now

The BP 'Examination' has finished normally
——— EXAMINATION ———

The BP 'MakeDocument' has finished normally
——— MAKEDOCUMENT ———

———————— Diagnosis.DiagnosisUnit[DU1]>> LEAVING SPLIT MODE ————————

Diagnosis.DiagnosisUnit[DU1]>> COMPOSITE CALL BP 'Consultation'

——— CONSULTATION ———
ALLOCATION >> Resource 'Alfredo' has been assigned to execute the participant 'Patient'
              in BP 'Consultation'
ALLOCATION >> Resource 'Marc' has been assigned to execute the participant 'Doctor'
              in BP 'Consultation'
     Consultation.Doctor[Marc]>> Executing 'evaluateExaminationResults'
     Consultation.Patient[Alfredo]>> Message 'check' received from 'Doctor'
     Consultation.Doctor[Marc]>> Message 'ckeck' sent to 'Patient'
     Consultation.Doctor[Marc]>> Executing 'checkPatient'
     Consultation.Doctor[Marc]>> Executing 'diagnosePatient'
     Consultation.Doctor[Marc]>> Executing 'prescribeTreatment'
```

```
      Consultation.Patient[Alfredo]>> Message 'd' received from 'Doctor'
      Consultation.Doctor[Marc]>> Message 'd' sent to 'Patient'
      Consultation.Patient[Alfredo]>> Message 'p' received from 'Doctor'
      Consultation.Doctor[Marc]>> Message 'p' sent to 'Patient'
      Consultation.Doctor[Marc]>> Executing 'fillPatientSheet'
ALLOCATION >> The resource 'Alfredo' is free now
ALLOCATION >> The resource 'Marc' is free now

The BP 'Consultation' has finished normally
------ CONSULTATION ------

Diagnosis.DiagnosisUnit[DU1]>> COMPOSITE CALL BP 'GiveInformation'

------ GIVEINFORMATION ------
ALLOCATION >> Resource 'Alfredo' has been assigned to execute the participant 'Patient'
              in BP 'GiveInformation'
ALLOCATION >> Resource 'Sue' has been assigned to execute the participant 'Nurse'
              in BP 'GiveInformation'
      GiveInformation.Nurse[Sue]>> Executing 'checkTreatment'
      GiveInformation.Patient[Alfredo]>> Message 'treatmentDetails' received from 'Nurse'
      GiveInformation.Nurse[Sue]>> Message 'treatmentDetails' sent to 'Patient'
      GiveInformation.Nurse[Sue]>> Executing 'checkMedicine'
      GiveInformation.Patient[Alfredo]>> Message 'medicineDetails' received from 'Nurse'
      GiveInformation.Nurse[Sue]>> Message 'medicineDetails' sent to 'Patient'
ALLOCATION >> The resource 'Alfredo' is free now
ALLOCATION >> The resource 'Sue' is free now

The BP 'GiveInformation' has finished normally
------ GIVEINFORMATION ------

ALLOCATION >> The resource 'DU1' is free now

The BP 'Diagnosis' has finished normally
------ DIAGNOSIS ------
```

It is worth mentioning that the *Simulation Selection* model (partially depicted in Listing 5.16), used along with the *patient diagnosis* model to generate the traces, constrains the simulation such that (1) the resource that takes control of the activities enclosed by the participant *Patient* within the business process *Registration* is named "Alfredo" (lines 9 and 27), and (2) the participant *Doctor* within the business process *Consultation* concludes the patient's health status is good (i.e. $hs = GOOD$) after having performed the atomic activity *checkPatient* (lines). Thus, according to the second constraint, there is no trace where the exception *EX_Admission* is raised and subsequently handled. The deviation that leads this exception to be raised is detected when the patient's health status is bad.

Listing 5.16: *Simulation Selection* model.

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <Simulation_Selection_Model:Enterprise xmi:version="2.0"
3              xmlns:xmi="http://www.omg.org/XMI"
4              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5              xmlns:Simulation_Selection_Model="http://ssm"
6              xsi:schemaLocation="http://ssm␣SSM.ecore" name="Hospital">
7   ...
8   <bps name="registration">
9     <participant name="Patient" candidats="//@resources.0"/>
10  ...
11  </bps>
12  <bps name="consultation">
13    <participant name="Doctor">
14      <activity name="checkPatient">
```

```
15        <variable name="hs" type="//@dttps.0"/>
16      </activity>
17    </participant>
18  </bps>
19  ...
20  <dttps xsi:type="Simulation_Selection_Model:HealthStatus" name="HealthStatus␣Good">
21    <enum value="GOOD"/>
22  </dttps>
23  <dttps xsi:type="Simulation_Selection_Model:HealthStatus" name="HealthStatus␣BAD">
24    <enum value="BAD"/>
25  </dttps>
26  ...
27  <resources name="Alfredo"/>
28 </Simulation_Selection_Model:Enterprise>
```

5.3.1.1 Current limitations of the simulator

There exist certain technical limitations in the simulator that justify further discussion. First, once the Java source files are generated as a result of having performed in sequence the M2M and M2T transformation, a manual intervention is required to implement expressions of type *OclConstraint* that take place in either a control statement as *While, Repeat* or *If* or a *dynamic resource allocation policy*.

The second limitation is related to the *Simulation Selection* model, or more specifically, on the information that it specifies. Rather than describing the boundaries that constrain the domain space of the *DT4BP* model being simulated, the *Simulation Selection* model specifies the instances to use when simulating the model.

6. PERSPECTIVES

Abstract

This Chapter describes possible improvements and extensions to the contributions of this thesis. Some of these extensions may be applied in the short term, whereas others may require the exploration of entirely new research areas. This chapter is divided into three parts: the first focuses on the extensions related to the DT4BP *modelling language; the second, is concerned with extending the Timed-CaaFWrk; the third part of this chapter addresses enhancements of the Timed-CAA-DRIP implementation framework.*

6.1 The DT4BP modelling language

One of the main contributions of this thesis is the *DT4BP* modelling language. Its description here represents the "first" release of this modelling language. Depending on its use, new extensions may be required to improve its "effectiveness" when capturing the domain-specific concerns it addresses, and the role it will play within the software development process where it will be used.

6.1.1 Modelling extensions

This section presents some ideas that should be considered when providing a new version of the modelling language.

- **Manual and automatic activities**
  By definition, a *manual* activity requires the participation of a human being. Conversely, an *automatic* activity is one that can be performed without human intervention. The current version of the language does not provide a means for capturing whether an activity is either manual or automatic. It may be of interest to provide mechanisms that allow the modeller to refine, if required, the kind of activity being modelled.
- **Inter and intra-collaborations**
  The collaborations performed by the different participants enclosed within a same business process are assumed to take part "physically" within the boundaries of organisation that runs the business process, regardless of whether the resources that perform the activities are formally related to the organisation or not . Hence, the third-party resources that the organisation relies on to perform its activities are assumed to be physically within the organisation's structure when participating in the execution of certain business process.

Thus, the *DT4BP* modelling language is suitable for "intra-organisational" business processes. Making *DT4BP* suitable for modelling inter-organisational business processes will required providing mechanisms to differentiate whether a participant "is" within the organization's structure or not. This provision will allow both intra- and inter-collaborations in the business process. Allowing inter-collaborations in the business processes increases the potential range of relationships that may take place during the execution of inter-organisational business processes. Subsequently, potentially abnormal events that have not previously been considered make occur due to communication problems, mismatched information formats or any other number of compatibility-related issues.

- **Timing constraint over the "recovery" part of the business process**
  One of the main features of the *DT4BP* modelling language is its rich set of time-related constructs that can be used to model many different types of time constraints. Hence, it is possible to model, for example, the maximum allowed time an activity, a participant, or the overall business process may take to complete. However, the language does not include time-related constructs specifically oriented to handle the recovery components. It may be necessary to explicitly model the time a business process may be allowed to spend on the recovery of a certain deviation. Furthermore, this information may be useful when performing off-line analysis of the business process in order to guarantee the goal achievement.

- **Deviation due to resource unavailability**
  The modelling language provides means to allow the explicit description of a vast number of deviations. In this thesis, deviations are categorised as functional-related and time-related. While the occurrence of the former depends on the specific activities enclosed by the business process, the latter occurs when a time constraint is missed. However, deviations resulting from missing resources have not been considered. It is assumed that there are always enough resources to execute the business process. Hence, the language does not provide a means to explicitly capture this kind of deviation. Extensions to cover this kind of deviation may be of interest 1) for discriminating whether a business process has aborted or failed due to either a lack of time or resources, and 2) for putting into place appropriate recovery strategies.

- **Data type constructors**
  The current version of the language allows the programmer to define both simple and structured data types. While simple data types allows the programmer to capture the basic concepts that underpin the domain of the business process, structure data types allow the modeller to encapsulate the basic concepts that are required for a higher order concept. However, the number of structured data types that may contain an encapsulated data type is limited to one. Hence, let *sd* be a structured data type composed of $d_1, ..., d_n$ data types, an object of type *sd* contains just only one instance for each d_i data type. Therefore, constructs like *Sequence, Bag* and *Set* should be provided to allow the definition of data types aimed at containing a collection of instances. Notice that these constructs can be used not only when defining a structured data types, but also when defining simple data types.

6.1.2 Graphical concrete syntax

DT4BP has a *textual* concrete syntax. Since business analysts usually feel more comfortable when dealing with a visual language rather a textual one, a desirable extension to the language

would be to provide a *graphical* concrete syntax. As *BPMN* is currently considered by the business process modelling community as the *"de-facto"* standard modelling notation, a good strategy would be to extend the BPMN graphical syntax to support the full set of *DT4BP* constructs. Making the graphical concrete syntax of *DT4BP* close to that of the BPMN's not only eases the ability of new users to adapt to the language, but also increases the chances of getting tool support. Regarding this last point, for example, one possibility would be to take any existing BPMN meta-model available for the Eclipse Modelling Framework (EMF) [BBM03], which means that there exists an Ecore file that implements the BPMN meta-model, and extend it with the specific constructs brought by the *DT4BP* language.

6.1.2.1 Position with respect to BPMN 2.0

Because BPMN is considered the "de-facto" standard modelling notation within the business process management community, it was one of the modelling languages taken as reference for the definition of *DT4BP*. Thus, there is a point that deserves further explanation.

During the development of this thesis, version 1.2 of the BPMN language was considered since the release of the version 2.0 was only achieved in August of 2009. Almost simultaneously, *DT4BP* was conceived. To the (pleasant) surprise of the author of this thesis, the latest version of BPMN gave raise to (1) a major emphasis on the notion of *collaboration*, and (2) the use of multiple diagrams to facilitate the separation of concerns when modelling a business process, in a similar manner as addressed in *DT4BP*. Hence, these coincidences provide some evidence of the suitability of *DT4BP* for the modelling of business processes.

6.1.3 Formalisation

The formalisation of a language is determined by the precision of its semantics. Hence, a language is considered as formal when its semantics are founded on mathematical concepts. Using a mathematically-based language as the semantic domain when defining a (source) language not only brings rigour and precision to the (source) language, but also provides the potential to use existing tools and techniques associated with the formal language chosen as the semantic domain. These tools and techniques are a useful means for finding inconsistencies, ambiguities and missing components over models, which are described in terms of the source language.

The most direct path to achieve the formalisation of the *DT4BP* modelling language would be by making use of the *COALA* [Vac00] formal language as semantic domain. *COALA* is a formal language oriented toward specifying the software systems designed according to the CaaFWrk paradigm. Hence, most of the CaaFWrk concepts, as presented in this thesis, are natively supported by this formal language. Extending *COALA* to fully cover not only CaaFWrk's concepts, but also those forming part of the Timed-CaaFWrk, is the first step. These extensions will result in a *Timed-COALA*, making it possible to easily obtain a semantic mapping, i.e. a model transformation, between *DT4BP* and *COALA*.

When validation must be performed, there does not exist any tool support associated with COALA for such a purposes. Thus, a formal language different from COALA should be used for validation. In this case, the formal language to be used as the semantic domain for *DT4BP* has to match the concerns of interest, i.e. concurrency, dependability and time, but which also comes with tool support for verification. i.e. a model checker, or automated theorem prover. At the time of writing, no formal language with such characteristics was known to exist.

6.1.4 Assessment

There does not exist a standard way for evaluating a business process' modelling notation. However, the *workflow patterns*, as initially presented in [vtKB03] and latter revised and extended by Rusell in [Rus07], define a commonly used benchmark for evaluating the suitability and effectiveness of a particular notation with respect to the modelling of business processes.

Besides the assessment of *DT4BP* concerning other modelling notations (see Section 3.5), it is also be important to asses the suitability and effectiveness of the language concerning the *workflow patterns*.

It is also worth noting, that another manner of assessing *DT4BP* would be by putting it into practice. Hence, its use to model both real and academic cases studies should also be considered as part of the assessment process of the language. Regarding this manner of assessing *DT4BP*, there is some ongoing research being performed within the context of the REACT project [REA10] already.

6.1.5 Use in the software development process

Since *DT4BP* has been engineered considering dependability as a major first-class concern, this modelling language has great potential to be used as the notation in the software development processes that consider dependability issues, particularly in the requirements elicitation phase.

An example of this kind of development process is DREP [MK09]. This development process relies on the notion of *exceptional use-case* [SMK06, SMKD05], which is an extended use-case template adapted to capture not only the expected environment-software system interactions that lead the service being described to provide the expected goal, but also dependability-related concerns such as service-related exceptions and their respective handling. Hence, one potential research direction would be testing the suitability of *DT4BP* models within the DREP methodology as a notation to capture the requirements of the software system under development. Furthermore, since DREP maps exceptional used-cases to DA-Charts and Markov chains to perform dependability analysis by model transformations, part of this research direction would also be to achieve a model transformation that allows the modeller to obtain the same kinds of target models, but taken as input *DT4BP* models.

Another example of a development process that incorporates dependability concerns, is the one presented by Lopatkin et al. in [LIR10]. In this approach, the authors propose a solution to integrate fault tolerance (FT) in a systematic manner during the refinement-based formal stage of the software development in the Event-B method. The proposal is to enrich an Event-B model, which formally describes the requirements under the assumption that the system always operate normally, with an FT view. An FT view describes fault tolerance features that allow the software system under development to recover when faced with errors. Despite the strong basis on which this methodology is founded, nothing is said about how the requirements that lead to the description of Event-B models and FT views are obtained. Notice that this requirement elicitation phase must be especially driven to obtain a clear separation between normal and abnormal behaviour of the software system under development.

An interesting research direction would then be to set up a requirement elicitation phase as initial step of the refinement-based software development process aimed at providing a document that explicitly separates the information concerning the normal behaviour of the software system from the abnormal. This document should be described using the *DT4BP* notation since it

provides, among other features, explicit views for these concerns: the *Process view* captures the requirements concerning with the normal behaviour, whereas the *Dependability view* those related with the abnormal behaviour. The *Process view* would have the required information to described the *Event-B* models; the *Dependability view* the information that allows modellers to describe the FT view.

6.1.6 Tool support

The diffusion and acceptance of a software-related language is tightly related to the available tool set that makes possible its use in a straightforward manner. Nowadays, it is common to find editors with not only highlighting, but also with auto-completion and automatic spell checking facilities that make the use of the software-related language very friendly.

The current *DT4BP* tool support, regarding the editing of models, is the dynamic instance creation feature provided by the Eclipse Modelling Framework (EMF). Since this editor works directly over the meta-model used to define the *abstract syntax* of the language, it is only possible to create correct models. However, its use is very click-intensive, which makes the model creation not only slow, but also hard to accomplish. The solution to this editing problem is to provide a customised editor for the *DT4BP* modelling language. It is planed to develop an editor using the Xtext language development framework [The10]. Once the language is enriched with a *graphical* concrete syntax, a graphical editor could be obtained by means of the the Graphical Eclipse Modelling (GMF) framework.

Regarding the validation of *DT4BP* models, the tool support is achieved by combining the *M2M* and *M2T* transformations that allow to obtain Java Timed-CAA-DRIP-compliant code from a given *DT4BP* model. As previously explained, this Java code, once compiled, is executed to obtain the different traces that describe the behaviour allowed by the *DT4BP* model given as input. It is worth recalling that a *Simulation Selection* model is also required to be provided as input, along with the *DT4BP* model, in order to accomplish the validation process. This *Simulation Selection* model defines the boundaries of the scenario to be simulated. However, the current version of the simulator that supports the validation process of the *DT4BP* models requires some extensions and enhancements to either reduce or ease the intervention of the modeller in the validation process. Required extensions, as well as potential enhancements include:

- *Code generation for OclConstraint expressions*
  Currently, the modeller has to manually complete the generated Java source files as no code is generated for expressions of type *OclConstraint*. The automatic code generation for this kind of expressions may be accomplished by enhancing the M2T transformation. In this case, the transformation must be extended to (1) parse expressions of type *OclConstraints* in order to know how it is made, and (2) map OCL expressions to Java statements. Another solution, it is to extend the *DT4BP* model with built-in OCL primitives, such that *OclConstraints* are defined in a native way, instead of assuming that they are properly defined. Despite the fact that this alternative brings more control to the way *OclConstraints* expressions are defined within a model, its main drawback is that it would lead towards and overwhelming the definition of the *DT4BP* language, as a non-minor part of OCL language would be required to be embedded within it.

- *Automatic instance generator*
  The pre- and post-conditions associated with each atomic activity should be used along

with the information placed within the *Simulation Selection* model in order to automatically obtain the set of possible instances that satisfy the constraints, i.e. pre-condition, post-condition and boundaries. One possibility would be to interface with the Alloy tool [Jac06] as a way to obtain these instances. Hence, the pre-condition, post-condition and boundaries related to a particular atomic activity should be used to generate an *Alloy* model, which allows the modeller to generate the instances of the atomic activity. These instances should be used at run-time when generating the traces that describe the allowed behaviour of the *DT4BP* model according to the specified boundaries.

6.2 The Timed-CaaFWrk

When analysing the original extensions given by Romanosvky et al. [RXR99, RXR98, BRR+98] to the CaaFWrk, a special interest concerning the scheduling of timed-CAAs arose as a means to minimise the number of timing constraints that are missed and the effects of using the pre-emptive recovery scheme over the timing behaviour of a CAA. The motivation of these topics, as well as the potential extensions to the Timed-CaaFWrk allowing it to overcome these issues are discussed here.

6.2.1 Scheduling

The abstract concurrent object-oriented (OO) computation model that underpins the CaaFWrk is defined as a collection of interacting objects, where the processes (or threads) executing concurrently, i.e. these processes may but need not overlap [BA06], corresponds to executions of operations on a group of objects. It is assumed that each object executes just one of its operations at time. This concurrent model leads to the design of a CaaFWrk-compliant software system that may be implemented and deployed on both mono-processor or distributed systems. Notice that, in this context, a distributed system is considered as one where the time interaction between two different processes is not negligible [Lam78].

Let's assume that a certain Timed-CaaFWrk-compliant design D_{caa} is going to be implemented and deployed on a mono-processor system. As every Timed-CaaFWrk-compliant design, D_{caa} is composed of several CAAs, then for the sake of simplicity, all these CAAs are assumed to be timed.

The roles executing within a CAA perform their tasks concurrently. This means that there is no need to specify the exact order in which the roles execute. If the CAA is properly designed, then the functional out,put (i.e. CAA's goal) will be the same regardless of the internal behaviour of its roles. What indeed will vary considerable is the timing behaviour of the CAA depending of how its roles are interleaved. Therefore, the issue is of special interest when CAAs start having timing constraints like E or t_3.

Roles within the same CAA use *internal* objects to share information. These internal objects are passed by message exchange. When a certain role wants to execute some operations over an internal object being shared with another peer role, there must exist a synchronisation between the roles to make such operations appear as atomic. These instructions then define a *critical region* within the role. The synchronisation required to protect a critical section is known as *mutual exclusion*. It is known that the unrestricted use of shared variables, in this case internal objects, is unreliable and unsafe. This is due to multiple update problems, since the update of a variable is not executed as an **atomic** operation. Therefore, internal objects have to be

protected to avoid multiple update problems. Any of the mechanisms used to reach this form of protected shared data (like semaphores or monitors) leads to the possibility of a role being suspended until some necessary future event occurs. Therefore, using synchronisation primitives the internal behaviour of a CAA exhibits non-determinism, and as said in [BW01], page 465: *a real-time systems needs to restrict the non-determinism found within concurrent systems.*

One way to restrict the non-determinism found in a concurrent software systems is by using certain *scheduling* policies. In this case, i.e. interprocess communication by internal objects, a possible solution could be the use of any of the existing *priority ceiling protocols*[1] (PCP), which were originally designed for single processor systems as a modification of the priority inheritance protocol (PIP). As explained in [BW01] page 492, this protocol, which assumes a pre-emptive dispatching[2], is defined as follows:

- each process has a static default priority assigned according to a certain scheme, e.g. rate monotonic priority assignment, which is based on the period of the process,
- each shared resource has a static ceiling value defined, this is the maximum priority of the processes that use it,
- a process has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked.

Now, let Caa_1 be a CAA composed of r_i $(i = 1..n)$ roles, such that its roles exchange information by io_j $(j = 1..m)$ internal objects and the CAA accesses the external objects eo_k $(k = 1..r)$, which are resources shared with other CAAs (i.e. Caa_l $(l = 2..w)$), then to implement the PCP into the Timed-CaaFWrk implies:

- to assign a unique priority to each role r_i $(i = 1..n)$ belonging to Caa_1,
- to assign a ceiling priority to each internal object io_j $(j = 1..m)$ used in Caa_1 to exchange information among its roles. The ceiling priority of the internal object io_j $(j = 1..m)$ will be the maximum priority of the roles r_i $(i = 1..n)$ that use it,
- a role has a dynamic priority that is the maximum of its own static priority and the ceiling values of any local object it has locked
- to assign a unique priority to each CAA Caa_l $(l = 1..w)$,
- to assign a ceiling priority to each external object eo_k $(k = 1..r)$. The ceiling priority of the external object eo_k $(k = 1..r)$ will be the maximum priority of the Caa_l $(l = 1..w)$ that use it.

It must be noted that, on a single processor system, the simple fact of applying PCP, ensures the mutual exclusion required over each shared resources (every internal object used into the CAA, and every external object accessed by the CAA). Thus, no synchronisation primitive would be required to provide mutual exclusion to protect the shared resources. Under the same conditions, i.e. single processor system, there is no possibility of being in deadlock, since the ceiling protocols are a form of deadlock prevention.

[1] This scheduling scheme is known as *static*, since priorities are assigned at pre-run-time.

[2] In pre-emptive dispatching, when a high-priority process is released during the execution of a lower-priority-process there is an immediate switch to the higher-process

In a distributed system, to ensure such properties, the PCP must maintain a global view of the acquired shared resources, which leads to high communication overhead. A solution to cope with this problem is to use a distributed PIP built on top of a deadlock avoidance schema [SSGM06].

6.2.2 Recovery process

The current Timed-CaaFWrk uses a pre-emptive scheme (instead of a blocking scheme) for the recovery process. According to the pre-emptive scheme a role has the right to interrupt any other roles when it has detected an exception. Notice that in a blocking scheme, a role has to reach the end of the CAA, or detect an exception and inform the other peer roles enclosed within the same CAA. It is only afterwards that this role is ready to accept information about the state of other roles. Thus, the pre-emptive scheme is a faster way of notifying the CAA roles of an exceptional situation, which speeds up the recovery process.

Since CAAs can be nested, the pre-emptive scheme embedded in the Timed-CaaFWrk has to provide special features. Let $Caa_{1,2}$ be two different CAAs such that Caa_2 is nested in Caa_1. Then according to the principles of the pre-emptive scheme, Caa_2 can be interrupted by any of the roles belonging to Caa_1 when an exception has been detected in one of them. The interruption of a nested CAA in this way represents its abortion. Therefore, it is up to the pre-emptive scheme to raise the *abort* exception on each role of the nested CAA in order to start the recovery phase. Once this is done, the nested CAA will follow the usual recovery path: exception resolution and then exception handling. In the case that one of the roles does not have a defined handler to deal with the abort exception, then a *failure* exception is returned by the nested CAA to its enclosing context. Once the nested CAAs were aborted, the recovery phase for the enclosing CAA has to be started, as usual. Thus, as argued in [RXR99], to abort the nested CAAs rather than wait for their completion as is done in the blocking scheme, speeds up the overall CAA recovery process, which is very important when CAAs have timing constraints.

However, the fact of aborting a nested CAA will not always speed up the recovery process of the enclosing CAA where the exception took place. For example, consider the case where a nested CAA has almost completed its execution and a role where an exception has been detected tries to interrupt the nested CAA. The process required to abort this nested CAA can take longer than the completion of the CAA. Thus, the right decision of whether a nested CAA has to be aborted or kept running can only be made if the CAA timing information is available. All the necessary timing information regarding the nested CAA and any other time-constrained CAAs would be held by the scheduler that implements the PCP, as explained previously. Thus, information about how much time is left to reach the deadline of the nested CAA (i.e. E or t_3) or how much execution time has consumed any of its roles can be gathered by the scheduler.

The proposal, then, is to extend the Timed-CaaFWrk with a scheme that combines the pre-emptive and blocking schemes. In this case, a role no longer has the right to interrupt the execution of any other roles when it has detected an exception. Instead, the role notifies the scheduler of the event. Once the scheduler is notified of the exception, it will decide which nested CAAs have to be aborted and which must be kept running until completion. Notice that in this context, the scheduler has to be notified of only value-related exceptions since time-related exceptions are automatically detected by the scheduler.

6.3 Timed-CAA-DRIP

Some future plans related to the implementation framework either were already mentioned when the framework was introduced in Section 4.4, e.g. performance analysis. Additional plans or can be deduced from the extensions related to the Timed-CaaFWrk conceptual framework, for which implementation support should be provided. Regarding this latest aspect, it should be mentioned that the required extensions for supporting both the Priority Ceiling Protocol (PCP) protocol and the new recovery scheme combines the blocking and pre-emptive approaches. However, there exist other extensions or potential research directions related to the implementation framework that may also be considered. They include:

6.3.1 Java Real-Time

The idea of including scheduling policies within the conceptual framework, and the fact that the implementation framework is developed in the Java programming language make Java Real-Time System (JRTS) [Ora10] an interesting option in the development of a new version of the implementation framework. Features introduced by this real-time version of the Java platform such as asynchronous transfer of control (ATC), high-resolution time (nanosecond accuracy) and the notion of real-time threads (i.e. threads that cannot be interrupted by the garbage collector) apply directly over Timed-CAA-DRIP since its implementation relies on such characteristics. Hence, the features brought about by JRTS along with the lessons learnt during its development should be exploited as much as possible when re-factoring Timed-CAA-DRIP to obtain a newer version with a better programming interface and run-time performance.

6.3.2 Built-in transactional support

It is assumed that the existence of an external transactional support such that the transaction-related properties brought by the CaaFWrk are achieved. The Timed-CAA-DRIP implementation framework has been developed using this hypothesis. Hence, engineers or programmers must choose which transactional support must be used. Once such a decision has been made, the next step is to insert the chosen transactional support into the implementation framework.

Despite the flexibility that this strategy brings, engineers and programmers may be interested in getting an implementation framework with "out-of-the-box" support. Hence, extending the Timed-CAA-DRIP to provide built-in transactional support is an improvement that would be welcome by the community. It is worth noting that this built-in support should be incorporated into the implementation framework in a way that it could be easily replaced by one chosen by the customer.

6.3.3 Distributed support

In the current version of the implementation framework, the participants playing the roles of any CAA execute within the same computer. This means that the whole software system has to be deployed along with the Timed-CAA-DRIP on the same computer such that all the application's and implementation framework's components share the same memory space.

In a distributed setting, both the roles and the objects (either external or internal) involved in the execution of a CAA may reside in different processing nodes. Since the Java Remote

Method Invocation (Java RMI) is the technology that allows Java-based applications to run in a distributed fashion, one possibility is to extend Timed-CAA-DRIP by adding the RMI support such that not only the roles but also the objects can be spread over potentially different processing nodes.

Another alternative for achieving distribution could be to re-engineer the implementation framework according to the Service Oriented (SO) paradigm. Some early attempts at combining the CaaFWrk principles with the SO paradigm can be found in [TIR02, BCG08]. Following this alternative, "web services" is the technological framework to be used as the means to attain distribution. Hence, both the roles that compose a CAA, and access to the external objects can be performed as web-services. However, it must be noted that nothing forbids using RMI also, such that some parts of the distribution are achieved by RMI-related means, whereas others by SO-related means. For example, the roles that compose a CAA may be distributed objects that are accessed by the participants in charge of their execution by RMI means, whereas external objects are accessed by a service-oriented interface.

7. CONCLUSION

This thesis has attempted to address the problems that arise when modelling business processes that are composed of multiple participants, are required to fulfil certain time-related constraints, and whose failures are not unacceptably frequent or severe from some stakeholders' viewpoint. Business process with these characteristics, within the context of this thesis, were referred to as *Dependable, Collaborative, Timed-constrained (DCTC)* business processes. In particular, the thesis investigated the problems that arise when describing information related to these concerns in a way such that the resulting model is not only comprehensible both for those that were involved in its creation (i.e. modellers), as well as for those who will make use of it later (e.g. software engineers and programmers), but also accurate with respect to the stakeholders' expectations.

In order to capture this domain-specific information such that the resulting model allows people to retrieve the same insights it was meant to provide, a modelling language with primitives that address the modelling of dependability, collaboration and time-related concerns both in a single and integrated manner is required. Moreover, this modelling language should come with tools that allow modellers to check the correctness of the business model with respect to the customer's expectations, i.e. to validate the model. Currently available business process modelling languages fall short when modelling dependability, collaboration and time aspects in an integrated way. While some of the existing languages fall short in considering at least one of these dimensions, those that cover the three dimensions only do so only partially. For this reason, a new a business process modelling language called *DT4BP* has been engineered to model DCTC business processes.

Each of the dimensions that define the business process domain space were deeply reviewed to extract the key concepts modellers need to manipulate during the business process description. The *DT4BP* primitives used to represent these key concepts were carefully chosen such that the business process definition remains easy to read and understand for people involved not only in the modelling process, but also in its latter use. This is achieved by structuring a business process definition as a set of different models, each of them targeting a specific concern. The feasibility of *DT4BP* for modelling the targeted business processes has been assessed by modelling a very simple (but complete) case study. Whether *DT4BP* is desirable for modelling dependable collaborative time-constrained business processes can only be determined from future practical experience gained by modelling more complex case studies.

In particular, reviewing of the notion of dependability was considered using the Coordinated Atomic Action conceptual framework (CaaFWrk) as a reference. This was done due to the fact that those concepts brought by the CaaFWrk are, for a major part, abstractions of the concepts included in collaborative business processes. The CaaFWrk however, had to be augmented with time-related extensions such that all the concerns being addressed at the level of the modelling language were included in a common conceptual framework, which could be used as the semantic domain for defining the modelling language. These time-related extensions, along with the other required extensions, were the result of using CaaFWrk as the semantic domain for the *DT4BP* modelling language led to a new conceptual framework named *Timed-CaaFWrk*. This new conceptual framework thus provides the opportunity of dealing with time-related aspects when designing a software system following the Coordinated Atomic Actions paradigm.

Following the software development process, once a software system is designed it must be implemented. Thus, an implementation framework named *Timed-CAA-DRIP* aimed at providing support to implement software systems designed according to Coordinated Atomic Paradigm has been developed. This implementation framework offers programmers a clear interface such that those concepts brought by the paradigm could be implemented in an easier way and using the same jargon. The support offered by this implementation framework is also at the run-time level, since it plays an active role in accomplishing the software system execution.

In particular, the Timed-CAA-DRIP implementation framework has been used as the executional semantics to achieve the enactment of *DT4BP* models and then allows for their validation by simulation. In fact, the enactment of the*DT4BP* models has been achieved by combining the semantic mapping that allows *DT4BP* concepts to be mapped over *Timed-CaaFWrk* concepts, with a model-to-text (M2T) transformation. This M2T transformation has been engineered to automatically generate the Java source code that implements a particular Timed-CaaFWrk design. It is worth underlining that the combinations of the semantic mapping with the M2T transformation was not only possible, but also straightforward due to the alignment with the Model-Driven Engineering principles followed along the development of this thesis.

BIBLIOGRAPHY

[AGM92] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Trans. Database Syst.*, 17(3):513–560, 1992.

[AK03] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Softw.*, 20(5):36–41, 2003.

[AKM08] Farhad Arbab, Natallia Kokash, and Sun Meng. Towards Using Reo for Compliance-Aware Business Process Modeling. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 108–123. Springer, 2008.

[AL81] Tom Anderson and Peter A. Lee. *Fault Tolerance: Principles and Practice, Second Edition.* Prentice-Hall, 1981.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[Ant65] Robert Anthony. *Planning and Control Systems: A Framework for Analysis.* Harvard University Graduate School of Business Management, Cambridge, MA., 1965.

[App98] Andrew W. Appel. *Modern compiler implementation in Java.* Cambridge University Press, New York, NY, USA, 1998.

[ARGP06] Elvira Rolón Aguilar, Francisco Ruiz, Félix García, and Mario Piattini. Applying software metrics to evaluate business process models. *CLEI Electron. J.*, 9(1), 2006.

[ARI09] ARIS Community. ARIS Express 1.0, 2009. Available from: http://www.ariscommunity.com/aris-express.

[ATL10a] ATL team. ATL Transformation Language, 2010. Available from: http://www.eclipse.org/atl/.

[ATL10b] ATL team. ATL/User Guide, 2010. Available from: http://wiki.eclipse.org/ATL/User_Guide.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[Avi85] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11(12):1491–1501, 1985.

[BA06] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd Edition) (Prentice-Hall International Series in Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[BBG+63] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language algol 60. *Commun. ACM*, 6(1):1–17, 1963.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[BBM03] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BCG08] Florencia Balbastro, Alfredo Capozucca, and Nicolas Guelfi. Analysis and framework-based design of a fault-tolerant web information system for m-health. *Service Oriented Computing and Applications*, 2(2-3):111–144, 2008.

[BCT05] Alan W. Brown, Jim Conallen, and Dave Tropeano. Introduction: Models, modeling, and model-driven architecture (mda). In Sami Beyeda, Matthias Book, and Volker Gruhn, editors, *Model-Driven Software Development*, pages 1–16. Springer-Verlag, 2005.

[BD07] Lindsay Bradford and Marlon Dumas. Getting Started with YAWL, 2007. Available from: http://yawlfoundation.org/yawldocs/GettingStartedWithYAWL.pdf.

[Béz04] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *UPGRADE*, 5(2):21–24, 2004.

[BM00] Peter A. Buhr and W.Y. R. Mok. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, 26(9):820–836, 2000.

[BM07] Artur Boronat and José Meseguer. Algebraic Semantics of EMOF/OCL Metamodels. Technical report, Technical Report UIUCDCS-R-2007-2904, Computer Science Department, University of Illinois at Urbana-Champaign, USA, 2007.

[BNT07] Benoit Baudry, Clementine Nebut, and Yves Le Traon. Model-Driven Engineering for Requirements Analysis. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, Washington, DC, USA, 2007. IEEE Computer Society.

[Bor10] Borland. Borland Together, 2010. Available from: http://www.borland.com.

[BRR+98] Alan Burns, Brian Randell, Alexander Romanovsky, Robert J. Stroud, Andrew J. Wellings, and Jie Xu. Temporal Constraints and Exception Handling in Object-Oriented Distributed Systems. Technical report, Design for Validation (DeVa) - Third Year Report, Esprit LTR Project 20072, 1998.

[Bt98] Alistair P. Barros and Arthur H.M. ter Hofstede. Towards the construction of workflow-suitable conceptual modeling techniques. *Information Systems Journal*, 8:313–337, 1998.

[Bur91] Alan Burns. Scheduling Hard Real-Time Systems: A Review. *Software Engineering Journal*, 6:116–128, may 1991.

[BW01] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CBS04] Jorge Cardoso, Robert P. Bostrom, and Amit Sheth. Workflow management systems and erp systems: Differences, commonalities, and applications. *Inf. Technol. and Management*, 5(3-4):319–338, 2004.

[CCGT09] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on semantics definition in mde - an instrumented approach for model verification. *Journal of Software*, 4(9), 2009.

[CF94] Flaviu Cristian and Christof Fetzer. Probabilistic internal clock synchronization. In *In Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, pages 22–31, Los Alamitos, CA, USA, Oct 1994. IEEE Computer Society.

[CGP+06] Alfredo Capozucca, Nicolas Guelfi, Patrizio Pelliccione, Alexander Romanovsky, and Avelino F. Zorzo. CAA-DRIP: a framework for implementing Coordinated Atomic Actions. *International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 385–394, 2006.

[CGP+09] Alfredo Capozucca, Nicolas Guelfi, Patrizio Pelliccione, Alexander Romanovsky, and Avelino F. Zorzo. Frameworks for designing and implementing dependable systems using coordinated atomic actions: A comparative study. *Journal of Systems and Software*, 82(2):207–228, 2009.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[CLV02] Albertas Caplinskas, Audrone Lupeikiene, and Olegas Vasilecas. A framework to analyse and evaluate information systems specification languages. In *ADBIS '02: Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, pages 248–262, London, UK, 2002. Springer-Verlag.

[CR86] Roy H. Campbell and Brian Randell. Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826, 1986.

[Cri89] Flaviu Cristian. Exception Handling. *Dependability of Resilient Computers (ed. T.Anderson)*, pages 68–97, 1989.

[CRR09] Fernando Castor Filho, Alexander Romanovsky, and Cecília M. F. Rubira. Improving reliability of cooperative concurrent systems with exception flow analysis. *J. Syst. Softw.*, 82(5):874–890, 2009.

[CY06] Jinjun Chen and Yun Yang. Key research issues in grid workflow verification and validation. In *ACSW Frontiers '06: Proceedings of the 2006 Australasian workshops on Grid computing and e-research*, pages 97–104, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[Cyr10] Cyril Faucher and Mickael Clavreul. Kermeta Emitter Template (KET)-Reference Manual, 2010. Available from: http://www.kermeta.org/documents/user_doc/ket_manual.

[dALB05] George M. de A. Lima and Alan Burns. Scheduling Fixed-Priority Hard Real-Time Tasks in the Presence of Faults. In *Proceedings of the Second Latin-American Symposium - LADC 2005*, pages 154–173, Berlin, Heidelberg, 2005. Springer.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Inf. Softw. Technol.*, 50(12):1281–1294, 2008.

[DG98] Wolfgang Deiters and Volker Gruhn. Process management in practice applying the funsoft netapproach to large-scale processes. *Automated Software Engg.*, 5(1):7–25, 1998.

[DT410] DT4BP Project. DT4BP project web site. http://wiki.lassy.uni.lu/Projects/DT4BP, September 2010.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[Ecl10] Eclipse Modeling Framework (EMF) project. EMF project web site. http://www.eclipse.org/modeling/emf/, September 2010.

[EFK89] M. Evangelist, N. Francez, and S. Katz. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15(11):1417–1426, 1989.

[EP98] Hans-Erik Eriksson and Magnus Penker. *Business Modeling With UML: Business Patterns at Work*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[EPR99] Johann Eder, Euthimios Panagos, and Michael Rabinovich. Time constraints in workflow systems. In *CAiSE 99: Proceedings of the 11th International Conference on Advanced Information Systems Engineering*, pages 286–300, London, UK, 1999. Springer-Verlag.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[FHL+96] E. D. Falkenberg, W. Hesse, P. Lindgreen, B. E. Nilsson, J. L. H. Oei, C. Rolland, R. K. Stamper, F. J. M. Van Assche, A. A. Verrijn-Stuart, and K. Voss. Frisco: A framework of information system concepts. Technical report, The IFIP WG 8. 1 Task Group FRISCO, 1996.

[FLM+04] Ariel Fuxman, Lin Liu, John Mylopoulos, Marco Pistore, Marco Roveri, and Paolo Traverso. Specifying and analyzing early requirements in tropos. *Requir. Eng.*, 9(2):132–150, 2004.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJM02] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, second edition, September 2002.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[GL07] Volker Gruhn and Ralf Laue. *Technologies for Business Information Systems*, chapter Approaches for Business Process Model Complexity Metrics, pages 13–24. Springer Netherlands, 2007.

[Goo75] John B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[GPR+06] Félix García, Mario Piattini, Francisco Ruiz, Gerardo Canfora, and Corrado A. Visaggio. FMESP: Framework for the modeling and evaluation of software processes. *Journal of Systems Architecture*, 52(11):627–639, 2006.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[Gra81] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981.

[GRS08] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Model-driven language engineering: The asmeta case study. In *ICSEA '08: Proceedings of the 2008 The Third International Conference on Software Engineering Advances*, pages 373–378, Washington, DC, USA, 2008. IEEE Computer Society.

[GTP07] Pau Giner, Victoria Torres, and Vicente Pelechano. Bridging the gap between bpmn and ws-bpel. m2m transformations in practice. In Nora Koch, Antonio Vallecillo, Geert J. Houben, Nora Koch, Antonio Vallecillo, and Geert J. Houben, editors, *MDWE*, volume 261 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[GZ89] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3bsd. *IEEE Trans. Softw. Eng.*, 15(7):847–853, 1989.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Har97] A.F. Harmsen. *Situational Method Engineering*. PhD thesis, University of Twente, 1997.

[Hen90] Matthew Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[HLMSR74] James J. Horning, Hugh C. Lauer, Peter M. Melliar-Smith, and Brian Randell. A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium*, pages 171–187, London, UK, 1974. Springer-Verlag.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[HOP05] Armin Haller, Eyal Oren, and Simeon Petkov. Aspects in Workflow Management. Technical report, Digital Enterprise Research Institute, April 2005.

[HR00] Bart-Jan Hommes and Victor Van Reijswoud. Assessing the quality of business process modeling techniques. *Hawaii International Conference on System Sciences*, 1:1007, 2000.

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.

[IFI] IFIP WG 10.4 on Dependable Computing and Fault Tolerance. Ifip wg 10.4. http://www.dependability.org/wg10.4/.

[Ins90] Institute of Electrical and Electronics Engineers (IEEE). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology.* 1990.

[Int] International Organization for Standardization (ISO). *ISO/TR 9007:1987, Information processing systems – Concepts and terminology for the conceptual schema and the information base.* ISO, Geneva, Switzerland.

[Int98] International Organization for Standardization (ISO). ISO/IEC 8859-1:1998, 1998. Available from: http://www.iso.org.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[Jav] Java Platform Standard Ed. 6, http://download.oracle.com/javase/6/docs/api/.

[JB96] Stefan Jablonski and Christoph Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation.* International Thomson Computer Press, September 1996.

[JBA06] Frédéric Jouault, Jean Bézivin, and Atlas Team. Km3: a dsl for metamodel specification. In *In proc. of 8th FMOODS, LNCS 4037*, pages 171–185. Springer, 2006.

[JBC+06] Frédéric Jouault, Jean Bézivin, Charles Consel, Ivan Kurtev, and Fabien Latry. Building dsls with amma/atl, a case study on spl and cpl telephony languages. In *In: Proceedings of the 1st ECOOP Workshop on DomainSpecific Program Development (DSPD), July 3rd*, 2006.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin / Heidelberg, 2006.

[Ken02] Stuart Kent. Model driven engineering. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 286–298, London, UK, 2002. Springer-Verlag.

[KK94] Hermann Kopetz and Kane H. Kim. A real-time object model rto.k and an experimental investigation of its potentials. *IEEE Computer Society's 1994 Int'l Computer Software and Applications Conf.*, January 1994.

[Kle06] Anneke Kleppe. MCC: A Model Transformation Environment. In Arend Rensink and Jos Warmer, editors, *Model Driven Architecture – Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 173–187. Springer Berlin / Heidelberg, 2006.

[Kle07] Anneke Kleppe. A language description is more than a metamodel. In *Fourth International Workshop on Software Language Engineering, Nashville, USA*, October 2007.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels.* Addison-Wesley Professional, 2008.

[KLV05] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.

[Ko09] Ryan K. L. Ko. A computer scientist's introductory guide to business process management (bpm). *Crossroads*, 15(4):11–18, 2009.

[KPR04] Raman Kazhamiakin, Marco Pistore, and Marco Roveri. Formal Verification of Requirements using SPIN: A Case Study on Web Services. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 406–415, Washington, DC, USA, 2004. IEEE Computer Society.

[KS97] Kane H. Kim and Chittur Subbaraman. Fault-tolerant real-time objects. *Commun. ACM*, 40(1):75–82, 1997.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley-IEEE Computer Society Pr, March 2008.

[KTK09] Juha Kärnä, Juha-Pekka Tolvanen, and Steven Kelly. Evaluating the Use of Domain-Specific Modeling in Practice. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, New York, NY, USA, 2009. ACM.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LaT] LaTeX project team. LaTeX. http://www.latex-project.org.

[LIR10] Ilya Lopatkin, Alexei Iliasov, and Alexander Romanovsky. On Fault Tolerance Reuse during Refinement. Technical Report CS-TR No 1188, School of Computing Science, Newcastle University, 2010.

[LK06] Beate List and Birgit Korherr. An evaluation of conceptual business process modelling languages. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1532–1539, New York, NY, USA, 2006. ACM.

[LLK09] Stephen S.G. Lee, Eng Wah Lee, and Ryan K. L. Ko. Business process management (bpm) standards: a survey. *Business Process Management Journal*, 15(5):744–791, 2009.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[LZRT08] X. Liu, W. J. Zhang, R. Radhakrishnan, and Y. L. Tu. Manufacturing perspective of enterprise application integration: the state of the art review. *International Journal of Production Research*, 46(16):4567–4596, 2008.

[Met10] MetaCase. MetaEdit+, 2010. Available from: http://www.metacase.com/.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer Berlin / Heidelberg, 2005.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[MK09] Sadaf Mustafiz and Joerg Kienzle. Drep: A requirements engineering process for dependable reactive systems. In Michael Butler, Cliff Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 220–250. Springer Berlin / Heidelberg, 2009.

[MKB08] Sadaf Mustafiz, Joerg Kienzle, and Andrey Berlizev. Addressing degraded service outcomes and exceptional modes of operation in behavioural models. In *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, pages 19–28, New York, NY, USA, 2008. ACM.

[MRv+09] Ronny S. Mans, Nicholas C. Russell, Wil M. P. van der Aalst, Arnold J. Molenman, and Piet J.M. Bakker. Schedule-Aware Workflow Managment Systems. In *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE 09)*, pages 81–96, Hamburg, Germany, 2009. University of Hamburg, Department of Informatics.

[MV06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[Net09] Network Time Protocol. NTP. www.ntp.org, 2009.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[No 10] No Magic Inc. MagicDraw, 2010. Available from: http://www.magicdraw.com/.

[OAS07] OASIS. WS-BPEL 2.0, 2007. Available from: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[Obj] Object Management Group (OMG). Available from: http://www.omg.org.

[Obj01] Object Management Group (OMG). OMG/Model Driven Architecture - A Technical Perspective, 2001. OMG Document: ormsc/01-07-01.

[Obj06] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification version 2.0. Technical Report 06-01-01, OMG, January 2006.

[Obj08] Object Management Group (OMG). MOF Query/View/Transformation specification v1.0, 2008. Available from: http://www.omg.org/spec/QVT/1.0/PDF/.

[Obj09a] Object Management Group (OMG). Business Process Modeling Notation (BPMN), Version 1.2, 2009. http://www.omg.org/spec/BPMN/1.2/PDF.

[Obj09b] Object Management Group (OMG). Unified Modeling Language Superstructure specification version 2.2, 2009. http://www.omg.org/cgi-bin/doc?formal/09-02-02.pdf.

[OHNAEE+07] Javier Ortiz-Hernández, Erika M. Nieto-Ariza, Hugo Estrada-Esquivel, Guillermo Rodríguez-Ortiz, and Azucena Montes-Rendon. A theoretical evaluation for assessing the relevance of modeling techniques in business process modeling. In *SOQUA '07: Fourth international workshop on Software quality assurance*, pages 102–107, New York, NY, USA, 2007. ACM.

[Ora10] Oracle. Java Real-Time System. http://java.sun.com/javase/technologies/realtime, September 2010.

[Pet96] C. A. Petri. Nets, time and space. *Theor. Comput. Sci.*, 153(1-2):3–48, 1996.

[PJ08] Kapil Pant and Matjaz Juric. *Business Process Driven SOA using BPMN and BPEL*. Packt Publishing, Birmingham, UK, 2008.

[PRP08] Jose Manuel Perez, Francisco Ruiz, and Mario Piattini. MDE for BPM: A Systematic Review. In Joaquim Filipe, Boris Shishkov, and Markus Helfert, editors, *Software and Data Technologies*, volume 10 of *Communications in Computer and Information Science*, pages 127–135. Springer Berlin Heidelberg, 2008.

[Pv07] Michael P. Papazoglou and Willem-Jan van den Heuvel. Business process development life cycle methodology. *Commun. ACM*, 50(10):79–85, 2007.

[PW06] Frank Puhlmann and Mathias Weske. M.: Investigations on soundness regarding lazy activities. In *Proceedings of the 4th International Conference on Business Process Management (BPM 2006), volume 4102 of LNCS*, pages 145–160. Springer Verlag, 2006.

[Ran75] Brian Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering. IEEE Press*, SE-1(2):220–232, 1975.

[RDV09] José E. Rivera, Francisco Durán, and Antonio Vallecillo. Formal specification and analysis of domain specific models using maude. *Simulation*, 85(11-12):778–792, 2009.

[RE01] Alexander Romanovsky and Paul Ezhilchelvan. Conversations with fixed and potential participants. *J. Syst. Archit.*, 47(2):193–196, 2001.

[REA10] REACT Project. REACT project web site. http://wiki.lassy.uni.lu/Projects/DT4BP, September 2010.

[RJK+06] Davide Di Ruscio, Frédéric Jouault, Ivan Kurtev, Jean Bézivin, and Alfonso Pierantonio. A practical experiment to give dynamic semantics to a DSL for telephony services development. Technical report, Technical Report 06.03, LINA, University of Nantes, France, 2006.

[RK07] Brian Randell and Maciej Koutny. *Theoretical Aspects of Computing - ICTAC 2007*, chapter Failures: Their Definition, Modelling and Analysis, pages 260–274. Springer, Berlin, Heidelberg, 2007.

[Rom07] Alexander Romanovsky. CAA, 1995-2007. Available from: http://homepages.cs.ncl.ac.uk/alexander.romanovsky/home.formal/caa.html.

[RPZ03] Alexander Romanovsky, Panos Periorellis, and Avelino F. Zorzo. On Structuring Integrated Web Applications for Fault Tolerance. In *Proceedings of ISADS 2003*, pages 99–106, April 2003.

[RRS+97] Alexander Romanovsky, Brian Randell, Robert J. Stroud, Jie Xu, and Avelino F. Zorzo. Implementation of blocking coordinated atomic actions based on forward error recovery. *J. Syst. Archit.*, 43(10):687–699, 1997.

[Rus07] Nicholas C. Russell. *Foundations of process-aware information systems.* PhD thesis, Faculty of Information Technology, Queensland University of Technology, 2007.

[RWD+08] Changrui Ren, Wei Wang, Jin Song Dong, Hongwei Ding, Bing Shao, and Qinhua Wang. Towards a flexible business process modeling and simulation environment. In *WSC '08: Proceedings of the 40th Conference on Winter Simulation*, pages 1694–1701. Winter Simulation Conference, 2008.

[RXR98] Alexander Romanovsky, Jie Xu, and Brian Randell. Exception handling in object-oriented real-time distributed systems. In *ISORC '98: Proceedings of the The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 32, Washington, DC, USA, 1998. IEEE Computer Society.

[RXR99] Alexander Romanovsky, Jie Xu, and Brian Randell. Coordinated Exception Handling in Real-Time Distributed Object Systems. *Computer Systems Science and Engineering*, 14(4):197–208, 1999.

[Sar00] Robert G. Sargent. Verification, validation, and accreditation: verification, validation, and accreditation of simulation models. In *WSC '00: Proceedings of the 32nd conference on Winter simulation*, pages 50–59, San Diego, CA, USA, 2000. Society for Computer Simulation International.

[Sch98] August-Wilhelm W. Scheer. *Aris-Business Process Frameworks.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[Sch00] August-Wilhelm W. Scheer. *Aris-Business Process Modeling.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[SGD03] Pnina Soffer, Boaz Golany, and Dov Dori. ERP modeling: a comprehensive approach. *Inf. Syst.*, 28(6):673–690, 2003.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003.

[SMK06] Aaron Shui, Sadaf Mustafiz, and Joerg Kienzle. Exception-aware requirements elicitation with use cases. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 221–242. Springer Berlin / Heidelberg, 2006.

[SMKD05] Aaron Shui, Sadaf Mustafiz, Joerg Kienzle, and Christophe Dony. Exceptional use cases. In *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 568–583. Springer Berlin / Heidelberg, 2005.

[Sof99] Software Technology Group, Technische Universität Dresden. Dresden OCL2 Toolkit for Eclipse, 1999. Available from: http://dresden-ocl.sourceforge.net/.

[SSGM06] César Sánchez, Henny B. Sipma, Christopher D. Gill, and Zohar Manna. Distributed priority inheritance for real-time and embedded systems. In Alex Shvartsman, editor, *Proceedings of the 10th International Conference On Principles Of Distributed Systems (OPODIS'06)*, volume 4305 of *LNCS*, Bordeaux, France, 2006. Springer-Verlag.

[Tel94] Gerard Tel. *Introduction to distributed algorithms*. Cambridge University Press, New York, NY, USA, 1994.

[The10] The Eclipse Foundation. Xtext - Language Development Framework. http://www.eclipse.org/Xtext/, September 2010.

[TIR02] Ferda Tartanoglu, Valérie Issarny, and Alexander Romanovsky. Dependability in the web service architecture. In *In Proceedings of the ICSE Workshop on Architecting Dependable Systems*, pages 89–108. Springer-Verlag, 2002.

[TLIR] Ferda Tartanoglu, Nicole Levy, Valérie Issarny, and Alexander Romanovsky. Using the B Method for the Formalization of Coordinated Atomic Actions. In *Proc. ICSE 2003 Workshop on Software Architectures for Dependable System*.

[Toh99] K. T. K. Toh. The realization of reference enterprise modelling architectures. *International Journal of Computer Integrated Manufacturing*, 12(5):403–417, 1999.

[Tri10a] Triskell team. Kermeta, 2010. Available from: http://www.kermeta.org/.

[Tri10b] Triskell team. Kermeta Emitter Template (KET), 2010. Available from: http://www.kermeta.org/mdk/ket.

[tvAR09] Arthur H.M. ter Hofstede, Wil M. P. van der Aalst, Michael Adams, and Nicholas C. Russell. *Modern Business Process Automation*. Springer-Verlag New York, Inc., December 2009.

[Vac00] Julie Vachon. *COALA : a design language for reliable distributed systems*. PhD thesis, Swiss Federal Institute of Technology Lausanne (Thesis no 2302), 2000.

[van99] Wil M. P. van der Aalst. Formalization and verification of event-driven process chains. *Information & Software Technology*, 41(10):639–650, 1999.

[van02] Wil M. P. van der Aalst. *Workflow management: models, methods, and systems.* MIT Press, Cambridge, MA, USA, 2002.

[van04] Wil M. P. van der Aalst. Business process management: A personal view. *Business Process Management Journal*, 10(2), April 2004.

[VD06] Martin Vasko and Schahram Dustdar. A view based analysis of workflow modeling languages. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 293–300, Washington, DC, USA, 2006. IEEE Computer Society.

[vtKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[WG08] Peter Y. H. Wong and Jeremy Gibbons. A process semantics for bpmn. In *ICFEM '08: Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, pages 355–374, Berlin, Heidelberg, 2008. Springer-Verlag.

[WG09] Peter Y. H. Wong and Jeremy Gibbons. A relative timed semantics for bpmn. *Electron. Notes Theor. Comput. Sci.*, 229(2):59–75, 2009.

[Wie10] Federico Wiecko. An Evaluation of MDE tools in the context of M2M transformations. Technical Report TR-LASSY-10-04, University of Luxembourg, 2010.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[WM08] Stephen A. White and Derek Miers. *BPMN Modeling and Reference Guide: Understanding and Using BPMN.* Future Strategies Inc., Lighthouse Pt, FL, USA, 2008.

[Wor] Workflow Management Coalition. www.wfmc.org.

[Wor99] Workflow Management Coalition. Terminology & Glossary. Document Number WFMC-TC-1011. Issue 3.0, 1999. Available from: http://www.wfmc.org/Download-document/WFMC-TC-1011-Ver-3-Terminology-and-Glossary-English.html.

[Wor08] Workflow Management Coalition. XPDL 2.1, 2008. Available from: http://www.wfmc.org/xpdl.html.

[XR97] Ming Xiong and Krithi Ramamritham. *Real-Time Database and Information Systems: Research Advances*, chapter Specification and Analysis of Transactions in Real-Time Active Databases, pages 327–354. Kluwer Academic Publishers, 1997.

[XRR+95] Jie Xu, Brian Randell, Alexander Romanovsky, Cecília M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. *Proceedings of the 25 International Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.

[XRR98] Jie Xu, Alexander Romanovsky, and Brian Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 12, Washington, DC, USA, 1998. IEEE Computer Society.

[XRR+02] Jie Xu, Brian Randell, Alexander Romanovsky, Robert J. Stroud, Avelino F. Zorzo, Ercument Canver, and Friedrich von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Trans. Comput.*, 51(2):164–179, 2002.

[YAW09a] YAWL Foundation. YAWL. www.yawl-system.com, 2008-2009.

[YAW09b] YAWL Foundation. YAWL User Manual 2.0, 2009. Available from: http://www.yawlfoundation.org/yawldocs/YAWLUserManual2.0.pdf.

[Zac99] John A. Zachman. A framework for information systems architecture. *IBM Syst. J.*, 38(2-3):454–470, 1999.

[ZRX+99] Avelino F. Zorzo, Alexander Romanovsky, Jie Xu, Brian Randell, Robert J. Stroud, and I. S. Welch. Using coordinated atomic actions to design safety-critical systems: a production cell case study. *Softw. Pract. Exper.*, 29(8):677–697, 1999.

AUTHOR INDEX

Deiters, Wolfgang *17*

Dijkman, Remco M. *13*

Ding, Hongwei *17*

Dong, Jin Song *17*

Dony, Christophe *200*

Dori, Dov *10*

DT4BP Project *146, 170*

Dumas, Marlon *13, 41*

Durán, Francisco *26*

Dustdar, Schahram *15*

Eclipse Modeling Framework (EMF) project *167, 168, 255*

Eder, Johann *34*

Elnozahy, E. N. (Mootaz) *20*

Eriksson, Hans-Erik *11*

Erl, Thomas *11*

Estrada-Esquivel, Hugo *16*

Evangelist, M. *110*

Ezhilchelvan, Paul *128, 130*

Falkenberg, E. D. *13*

Fetzer, Christof *34*

Fleurey, Franck *168*

France, Robert *23, 27*

Francez, N. *110*

Fuxman, Ariel *18*

Gamma, Erich *154*

García, Félix *16*

Garcia-Molina, Hector *21*

Gargantini, Angelo *23, 25*

Garoche, Pierre-Loïc *26*

Ghezzi, Carlo *17*

Gibbons, Jeremy *13*

Gill, Christopher D. *203*

Giner, Pau *24*

Golany, Boaz *10*

Goodenough, John B. *116*

Gosling, James *26*

Gray, Jim *20, 21, 71, 89, 110*

Green, J. *25*

Gruhn, Volker *16, 17*

Guelfi, Nicolas *146, 151, 206*

Gusella, R. *34*

Haller, Armin *15*

Harel, David *24, 121, 173*

Harmsen, A.F. *13*

Helm, Richard *154*

Helsen, S. *28*

Hennessy, Matthew *24, 52*

Hesse, W. *13*

Hoare, C. A. R. *13*

Hommes, Bart-Jan *15*

Horning, James J. *20*

IFIP WG 10.4 on Dependable Computing and Fault Tolerance *18*

Iliasov, Alexei *200*

Institute of Electrical and Electronics Engineers (IEEE) *13, 17*

International Organization for Standardization (ISO) *13, 14, 49, 61, 237*

Issarny, Valérie *125, 206*

Jablonski, Stefan *15, 16*

Jackson, Daniel *201*

Jazayeri, Mehdi *17*

Jézéquel, Jean-Marc *168*

Johnson, David B. *20*

Johnson, Ralph *154*

Jouault, Frédéric *25, 26, 174, 255*

Joy, Bill *26*

Juric, Matjaz *11, 12*

Kärnä, Juha *27*

Katz, C. *25*

Katz, S. *110*

Kazhamiakin, Raman *18*

Kelly, Steven *27, 31*

Kent, Stuart *23*

Kienzle, Joerg *89, 200*

Kiepuszewski, Bartek *15, 40, 200*

Kim, Kane H. *134*

Kleppe, Anneke *xvii, 24–26, 28, 29, 49, 52, 56, 113*

Klint, Paul *25*

Ko, Ryan K. L. *9, 12, 13*

Kokash, Natallia *13*

Kopetz, Hermann *134*

Korherr, Birgit *16*

Koutny, Maciej *18*

Kozaczynski, Wojtek *28*

Kühne, Thomas *23*

Kurtev, Ivan *25, 26, 174, 255*

Lämmel, Ralf *25*

Lamport, Leslie *20, 21, 202*

Landwehr, Carl E. *18, 19*

Laprie, Jean-Claude *18, 19*

LaTeX project team *27*

Latry, Fabien *26*

Laue, Ralf *16*

Lauer, Hugh C. *20*

Lee, Eng Wah *12, 13*

Lee, Peter A. *129*

Lee, Stephen S.G. *12, 13*

Levy, Nicole *125*

Lindgreen, P. *13*

Lipow, M. *16*

List, Beate *16*

Liu, Lin *18*

Liu, X. *11, 106*

Lopatkin, Ilya *200*

Lupeikiene, Audrone *15*

Mandrioli, Dino *17*

Manna, Zohar *203*

Mans, Ronny S. *45*

McCarthy, J. *25*

Melliar-Smith, Peter M. *20*

Meng, Sun *13*

Mens, Tom *28*

Merks, Ed *23, 27, 168, 174, 199, 255*

Meseguer, José *25*

MetaCase *27*

Meyer, Bertrand *85*

Miers, Derek *24, 33, 39, 40*

Milner, R. *13*

Mok, W.Y. R. *20, 97, 116*

Molenman, Arnold J. *45*

Montes-Rendon, Azucena *16*

Muller, Pierre-Alain *168*

Mustafiz, Sadaf *89, 200*

Mylopoulos, John *18*

Naamad, Amnon *121*

Nebut, Clementine *168*

Network Time Protocol *34*

Nielson, Flemming *25*

Nielson, Hanne Riis *25*

Nieto-Ariza, Erika M. *16*

Nilsson, B. E. *13*

No Magic Inc. *27*

OASIS *12, 24*

Object Management Group (OMG) *12, 23–25, 27, 28, 35, 36, 168*

Oei, J. L. H. *13*

Oracle *205*

Oren, Eyal *15*

Ortiz-Hernández, Javier *16*

Ouyang, Chun *13*

Panagos, Euthimios *34*

Pant, Kapil *11, 12*

Papazoglou, Michael P. *11, 88*

Pease, Marshall *21*

Pelechano, Vicente *24*

Pelliccione, Patrizio *146, 151*

Penker, Magnus *11*

Perez, Jose Manuel *24*

Periorellis, Panos *128*

Perlis, A. J. *25*

Petkov, Simeon *15*

Petri, C. A. *13*

Piattini, Mario *16, 24*

Pierantonio, Alfonso *25, 26*

Pistore, Marco *18*

Puhlmann, Frank *13*

Rabinovich, Michael *34*

Radhakrishnan, R. *11, 106*

Ramamritham, Krithi *133*

Randell, Brian *18–22, 103, 110, 113, 117, 119, 127–130, 134, 142, 144, 202, 204*

REACT Project *200*

Reijswoud, Victor Van *15*

Ren, Changrui *17*

Reuter, Andreas *20, 71, 89, 110*

Riccobene, Elvinia *23, 25*

Rivera, José E. *26*

Rodríguez-Ortiz, Guillermo *16*

Rolland, C. *13*

Romanovsky, Alexander *22, 110, 111, 113, 117, 119, 125, 127–130, 134, 142, 144, 146, 151, 200, 202, 204, 206*

Roveri, Marco *18*

Rubira, Cecília M. F. *22, 110, 111, 129*

Ruiz, Francisco *16, 24*

Rumpe, Bernhard *23, 24, 27, 173*

Ruscio, Davide Di *25, 26*

Russell, Nicholas C. *13, 15, 18, 41, 45, 107, 200*

Rutishauser, H. *25*

Samelson, K. *25*

Sánchez, César *203*

Sargent, Robert G. *18*

Scandurra, Patrizia *23, 25*

Scheer, August-Wilhelm W. *12, 13*

Sendall, Shane *28*

Shao, Bing *17*

Sheth, Amit *10*

Shostak, Robert *21*

Shui, Aaron *200*

Sipma, Henny B. *203*

Soffer, Pnina *10*

Software Technology Group, Technische Universität Dresden *54*

Stamper, R. K. *13*

Steele, Guy *26*

Stroud, Robert J. *22, 110, 113, 127–129, 134, 144, 202*

Subbaraman, Chittur *134*

Tartanoglu, Ferda *125, 206*

Tel, Gerard *21*

ter Hofstede, Arthur H. M. *15, 40, 200*

ter Hofstede, Arthur H.M. *13, 18*

The Eclipse Foundation *201*

Thirioux, Xavier *26*

Toh, K. T. K. *11, 106*

Tolvanen, Juha-Pekka *27, 31*

Torres, Victoria *24*

Traon, Yves Le *168*

Traverso, Paolo *18*

Triskell team *27, 30, 167, 168*

Tropeano, Dave *23*

Tu, Y. L. *11, 106*

Ullman, Jeffrey D. *24, 25, 173, 231*

Vachon, Julie *132, 199*

Vallecillo, Antonio *26*

van den Heuvel, Willem-Jan *11, 88*

van der Aalst, Wil M. P. *9, 11, 13, 15, 17, 18, 40, 44, 45, 55, 200*

Van Gorp, Pieter *28*

van Wijngaarden, A. *25*

Vasilecas, Olegas *15*

Vasko, Martin *15*

Vauquois, B. *25*

Verhoef, Chris *25*

Verrijn-Stuart, A. A. *13*

Visaggio, Corrado A. *16*

Vlissides, John *154*

von Henke, Friedrich *113*

Voss, K. *13*

Wang, Qinhua *17*

Wang, Wei *17*

Wang, Yi-Min *20*

Warmer, Jos *28, 49, 56, 113*

Wegstein, J. H. *25*

Welch, I. S. *127*

Wellings, Andrew J. *20, 22, 110, 133, 134, 144, 202, 203*

Weske, Mathias *13*

White, Stephen A. *24, 33, 39, 40*

Wiecko, Federico *30*

Wong, Peter Y. H. *13*

Woodger, M. *25*

Workflow Management Coalition *9, 10, 12, 32*

Wu, Zhixue *22, 110, 129*

Xiong, Ming *133*

Xu, Jie *22, 110, 113, 117, 119, 127–129, 134, 142, 144, 202, 204*

Yang, Yun *17*

YAWL Foundation *13, 36, 41, 44*

Zachman, John A. *11*

Zatti, S. *34*

Zhang, W. J. *11, 106*

Zorzo, Avelino F. *113, 127, 128, 146, 151*

APPENDIX

A. DT4BP TEXTUAL CONCRETE SYNTAX

This Appendix provides the syntactic specification[1] of the *DT4BP* business process modelling language. The Context-free grammar or grammar, for short, is also sometimes called BNF (Backus-Naur Form) [AU77]. The grammer is the common notation used to specify the syntax of a language. Listed below is the grammar for the *DT4BP* language.

A.1 Notational conventions

1. boldface symbols such as **if** or **do** denote terminals
2. quoted symbols such as '+' or '{' denote terminals
3. plain symbols denote nonterminals
4. ϵ is a regular expresssion denoting the empty string
5. a is a regular expression denoting string a
6. a b is a regular expression denoting the string ab formed by appending b to a
7. $(\alpha \mid \beta)$ is a regular expression which denotes either α or β
8. $(\alpha)^*$ is a regular expression which denotes zero or more instances of α
9. $(\alpha)^+$ is a regular expression which denotes one or more instances of α
10. $[\alpha]$ is a regular expression which denotes zero or one instance of α
11. the left side of the first production rule is the *start symbol*

A.2 Production rules

businessProcess → processModel resourceModel [dataModel] [dependabilityModel]

[1] The rules that determine whether a string is a valid *DT4BP* business process or not.

A.2.1 Process Model

processModel → **business process** procName ‘(’ [params] ‘)’
[start] [when] [last] [period] [pre] ‘{’
(participant)$^{+}$
‘}’ [normal]

participant → **participant** partName ‘{’
stmts
‘}’ [workFor]

stmts → statement ‘;’ statement
| statement ‘;’ stmts

statement → execute [in] [within]
| control
| objDecl

execute → block [activityDeviations]
| activity
| send
| receive

control → ifThen
| ifThenElse
| while
| repeat
| split
| spawn

block → **do** statement
| **do** ‘{’ stmts ‘}’

activity → (composite | atomic | nested) ‘(’ [args] ‘)’
[activityDeviations]

composite → **composite** activityName [‘[’resourceAllocations‘]’]

atomic → activityName
[pre]
[normal]

nested → **nested** partName

send → **send** (msg | msg ‘(’ localObjects ‘)’) **to** partName [**block**]

receive → **receive** (msg | msg ‘(’ localObjects ‘)’) **from** partName

localObjects → objName
| objName ‘,’ localObjects

ifThen → **if** oclConstraint **then** statement ’
| objDecl **if** oclConstraint **then** ‘{’ stmts ‘}’

ifThenElse → **if** oclConstraint **then** statement **else** statement
| **if** oclConstraint **then** ‘{’ stmts ‘}’ **else** statement
| **if** oclConstraint **then** statement **else** ‘{’ stmts ‘}’
| **if** oclConstraint **then** ‘{’ stmts ‘}’ **else** ‘{’ stmts ‘}’

while → **while** oclConstraint **do** statement
| **while** oclConstraint **do** ‘{’ stmts ‘}’

repeat → **repeat** statement **until** oclConstraint
| **repeat** ‘{’ stmts ‘}’ **until** oclConstraint

split → **split** activity ‘,’ activity
| **split** ‘{’ (activity ‘,’)$^+$ activity ‘}’ ‘,’ ‘{’ (activity ‘,’)$^+$ activity ‘}’

spawn → **spawn** activity ‘,’ activity
| **spawn** ‘{’ (activity ‘,’)$^+$ activity ‘}’ ‘,’ ‘{’ (activity ‘,’)$^+$ activity ‘}’

objDecl → localObject

localObject → (primitive | datatype) objName [**expire** ‘(’ timeExp‘)’]

primitive → dInteger | dString | dBoolean | dFloat

dInteger → **Integer**

dString → **String**

dBoolean → **Boolean**

dFloat → **Float**

datatype → typeName

params → **in** parameter
| **out** parameter
| **in** parameter ‘;’ **out** parameter

parameter → object
| object ‘,’ parameter

object → typeName objName

when → **when** ‘(’ eventName ‘)’

eventName → name
| **clock** ‘(’ calendarTime ‘)’

start → **start** timeExp
| **start** '[' timeExp ',' '_' ']'
| **start** '[' '_' ',' timeExp ']' [**deviation** '[' **timeout** ']']
| **start** '[' timeExp ',' timeExp ']' [**deviation** '[' **timeout** ']']

last → **last** timeExp [timeDeviation]
| **last** '[' timeExp ',' '_' ']' [timeDeviation]
| **last** '[' '_' ',' timeExp ']' [timeDeviation]
| **last** '[' timeExp ',' timeExp ']' [timeDeviation]

period → **every** timeExp **until** timeExp

pre → **pre** '[' oclConstraint ']'

normal → **post** '[' oclConstraint ']'

in → **in** timeExp

within → **within** '[' timeExp ',' '_' ']' [timeDeviation]
| **within** '[' '_' ',' timeExp ']' [timeDeviation]
| **within** '[' timeExp ',' timeExp ']' [timeDeviation]

workFor → **workFor** '[' timeExp ',' '_' ']' [timeDeviation]
| **workFor** '[' '_' ',' timeExp ']' [timeDeviation]
| **workFor** '[' timeExp ',' timeExp ']' [timeDeviation]

resourceAllocations → resourceAllocation
| resourceAllocation ',' resourceAllocations

resourceAllocation → [assignAlloc] (reference | dynamic | static | onDemand)

reference → **alloc** '(' resourceVar ')'

dynamic → **alloc** '(' oclConstraint | '_' ')'

static → **alloc** '(' resourceName ')'

onDemand → **alloc** '(' **new** partName ')'

assignAlloc → resourceVar '='

activityDeviations → activityDeviation [pHandler]
| activityDeviation **or** activityDeviations
| activityDeviation **and** activityDeviations

timeDeviations → timeDeviation [pHandler]
| timeDeviation [pHandler] ',' timeDeviation [pHandler]

activityDeviation → **deviation** '[' exceptionName | condition ']'

condition → FailureDeviation
| AbortDeviation
| Predicate
| DataDurationDeviation

FailureDeviation → **failed**

AbortDeviation → **aborted**

Predicate → oclConstraint

DataDurationDeviation → **dataExpired** '(' objNames ')'

objNames → objName
| objName ',' objNames

timeDeviation → **deviation** '[' timeoutType ']'

timeoutType → **timeout.min**
| **timeout.max**
| **timeout**

pHandler → '{' stmts '}'

args → **in** arg
| **out** arg
| **in** arg ';' **out** arg

arg → objName
| objName ',' arg

A.2.2 Resource Model

resourceModel → **resources** '{' (resource)$^+$ '}'

resource → partName ';'
| partName '=' candidats

candidats → candidat ';'
| candidat ',' candidats

candidat → resourceName
| resourceName '{' capabilities '}'

capabilities → '(' capability ')'
| '(' capability ')' ',' capabilities

capability → name ',' int

A.2.3 Data Model

dataModel → (dataTypes)$^+$

dataTypes → dataType
| dataType ';' dataTypes

dataType → **type** typeName
| **type** typeName ('{' attributes [invariant] '}' | enumerationLiteral)

attributes → attribute
| attribute ';' attributes

attribute → typeName attrName

invariant → **where** oclConstraint

enumerationLiteral → '=' **enum** '{' literals '}'

literals → '=' name
| name ';' literals

A.2.4 Dependability Model

dependabilityModel → resolution recovery

resolution → **resolution** '{' exceptionList '− >' handlerName '}'

exceptionList → exceptionName
| exceptionName **and** exceptionList

recovery → **recovery** '{' (handlers)$^+$ '}'

handlers → handlerName '{'
(participant)$^+$
'}' outcome

outcome → **Normal** | degraded | **Failed** | **Aborted**

degraded → **degraded** '[' oclConstraint ']'

A.2.5 Extras

letter → ‘A’ | ‘B’ | ... | ‘Z’ | ‘a’ | ‘b’ | ... | ‘z’
digit → ‘0’ | ‘1’ | ... | ‘9’
symbol → ‘_’ | ‘-’
name → letter (letter | digit | symbol)*

timeUnit → **year** | **month** | **week** | **day** | **hs.** | **min.** | **sec.**
timeExp → nat timeUnit

calendarTime → year ‘-’ month ‘-’ day ‘:’ hour ‘-’ min
year → [1970..9999]
month → [1..12]
day → [1..31]
hour → [0..23]
min → [0..59]

oclConstraint → string

procName → name
partName → name
activityName → name
typeName → name | **Integer** | **Float** | **String** | **Boolean**
objName → name
exceptionName → name
| **Abort**
| **TimeoutMinBPLast**
| **TimeoutMaxBPLast**

resourceName → name
resourceVar → name
attrName → name
handlerName → name
msg → name

Notes:

1. The terminal *string* represents a sequence of characteres of the ISO/IEC 8859-1:1998 [Int98] (aka Latin-1) character encoding.

2. The terminal *nat* represents the possitive integers.

3. The predicate *oclConstraint* is written using the *OCL* language. Since the grammar rules of the OCL language are not included in the *DT4BP* grammar, they are assumed to be syntanctically compliant with the *OCL* language. Thus, the nonterminals *oclConstraint* is defined as strings in the DT4BP grammar rules.

B. DT4BP META-MODEL

This Appendix joins in one consolidated figure all the concepts and relationships that form part of the *DT4BP* meta-model.

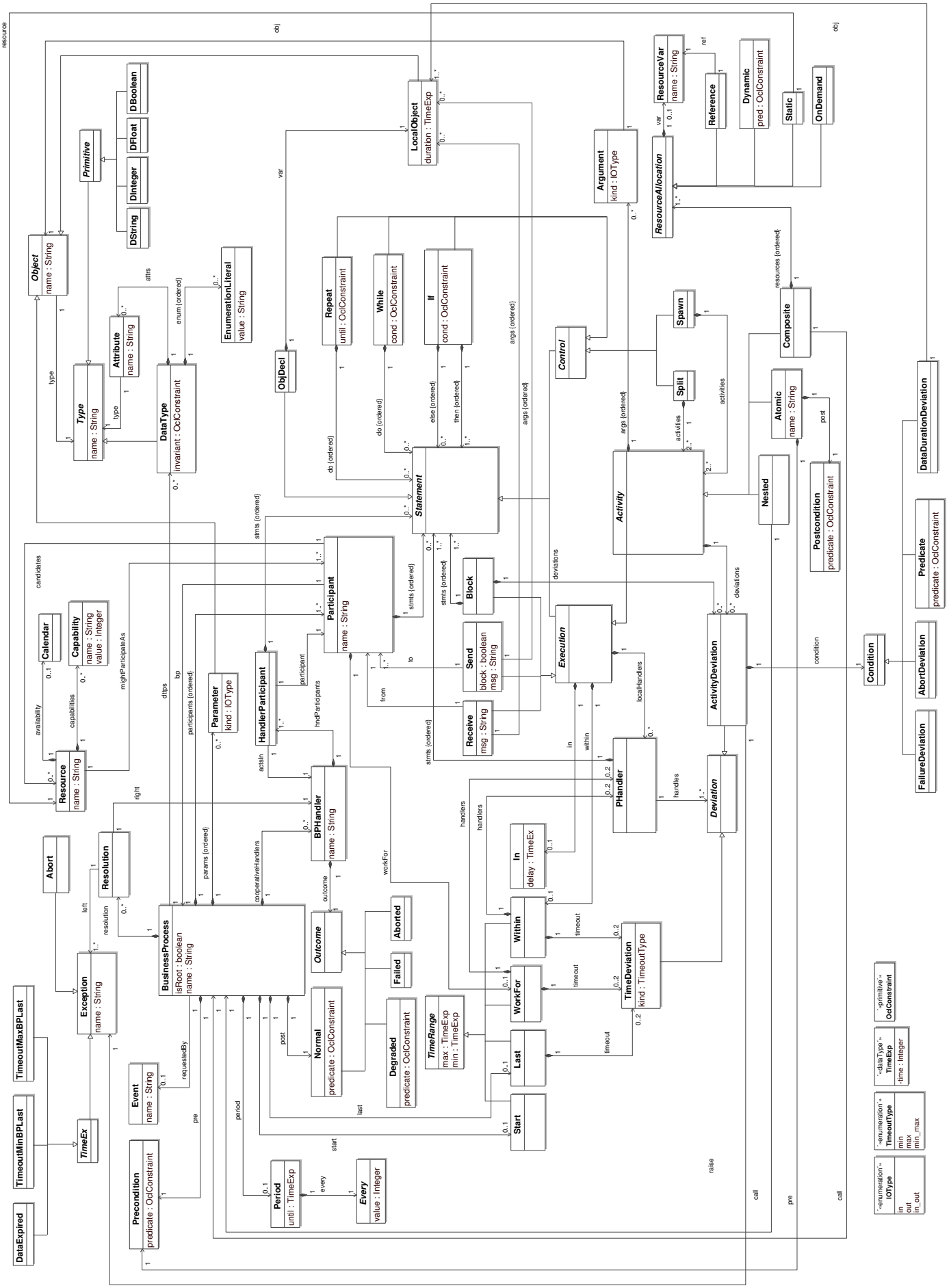

Fig. B.1: DT4BP Meta-Model.

C. THE PATIENT DIAGNOSIS RUNNING EXAMPLE IN DT4BP

C.1 Diagnosis business process definition

C.1.1 Process Model

```
;;*************************************************************
business process diagnosis(out PatientSheet ps) when(patientArrives) last[_,2hs.] {
;;*************************************************************

  participant DiagnosisUnit {
     FireAlarm fa;
     do{
        Person prs;
        composite registration[_,p = alloc(new Patient)](out prs) within[_,15min.];

        Temperature t expire(1hs.);
        BloodPressure bp expire(1hs.);

        do{split composite examination[p,_,_](out t,bp),
                  composite makeDocument[_](in prs; out ps);
        }within[_,30min.];

        composite consultation[p,_](in ps,t,bp; out ps) within[15min.,1hs.]
                        deviation[EX_Admission|ps.d.oclIsUndefined() and
                                  not ps.p.t.oclIsUndefined()];

        composite giveInformation[p,_](in ps) within[_,15min.];

     }deviation[EX_Fire| fa = #ON)];
  }

} post[not ps.d.oclIsUndefined() and not ps.p.oclIsUndefined()]
```

C.1.2 Data Model

```
type  Person{
        String name;
        String surname;
        Calendar birthday;
        String address;
        String city;
        String country;
        Integer ssn;
}

type Calendar;

type Temperature { Integer t where
        (30 <= t) and (t <= 50)}

type BloodPressure { Integer bp where
```

```
        (50 <= bp) and (bp <= 250)}

type Treatment, Medicine, Diagnosis, MedicalHistory;

type PatientSheet{
        Integer ssn;
        String name;
        String surname;
        String address;
        String city;
        String country;
        MedicalHistory mh;
        Diagnosis d;
        Prescription p;
}

type Prescription{
        Treatment t;
        Medicine m;
}

type FireAlarm = enum{ON,OFF}
```

C.1.3 Resource Model

```
resources{
        DiagnosisUnit = Unit1;
}
```

C.1.4 Dependability Model

```
;;Binding EXEC-HANDLER

resolution{
        EX_Fire -> Evacuation;
        EX_Admission -> PatientAdmission;
}

;;HANDLER definition

recovery{
        Evacuation(){
                participant DiagnosisUnit{
                        evacuateDiagnosisUnit();
                }
        }Failed

        PatientAdmission(){
                participant DiagnosisUnit{
                        admit(in ps);
                }
        }degraded[ps.d.oclIsUndefined() and not ps.p.t.oclIsUndefined()]

}
```

C.2 Registration business process definition

C.2.1 Process Model

```
;;************************************************************
business process registration(out Person p) last[_,15min.]{
;;************************************************************

  participant Secretary{
    send reqSSCard to Patient block;
    SocialSecurityCard ssCard;
    receive reqSSCard(ssCard) from Patient;
    Hospital h;

    checkSSCard(in ssCard; out h)
                post[ssCard.country = h.country]
                deviation[EX_Foreigner|ssCard.country <> h.country];

    String address,city;
    getPersonAddress(in ssCard; out address,city);
                post[address<>'' and city <>'']

    send reqAddress_and_City to Patient block;
    String p_address,p_city;
    receive dataRequested(p_address,p_city) from Patient;

    validatePerson(in p_address,p_city,address,city)
                post[p_address=address and
                     p_city=city]
                deviation[Abort|p_address<>address or
                                p_city<>city]

    registerPerson(in address,city,ssCard;out p)
                post[p|p.oclIsNew() and
                     p.name = ssCard.name and
                     p.surname = ssCard.surname and
                     p.birthday = ssCard.birthday and
                     p.address = address and
                     p.city = city and
                     p.country = ssCard.country and
                     p.ssn = ssCard.number];
  }

  participant Patient{
    receive reqSSCard from Secretary;
    SocialSecurityCard ssCard;
    searchSScard(out ssCard) post[not ssCard.oclIsUndefined()];
                             deviation[EX_NotSSCard|ssCard.oclIsUndefined()]

    send reqSSCard(ssCard) to Secretary;
    receive reqAddress_and_City from Secretary;
    String p_address,p_city;
    send dataRequested(p_address,p_city) to Secretary;
  }
} post[not p.oclIsUndefined()]
```

C.2.2 Data Model

```
type  Person{
        String name;
        String surname;
        Calendar birthday;
        String address;
        String city;
        String country;
        Integer ssn;
```

```
}

type Calendar;

type  SocialSecurityCard{
        Integer number;
        String name;
        String surname;
        Calendar birthday;
        Calendar expirationDate;
        Integer cardID;
        String country;
}

type  Hospital{
        String name;
        String address;
        String city;
        String country;
}

type ForeignerForm{
        String address;
        String city;
        }
```

C.2.3 Resource Model

```
resources{
        Secretary = Ann;
        Patient;
}
```

C.2.4 Dependability Model

```
;;Binding EXEC-HANDLER

resolution{
        EX_NotSSCard -> AdminOffice;
        EX_Foreigner -> FillSpecialForm;
        Abort -> AdminOffice;
}

;;HANDLER definition

recovery{

   AdminOffice {
        participant Secretary{
                send goToAdminOffice to Patient block;
        }

        participant Patient{
                receive goToAdminOffice from Secretary;
        }
   }Aborted


   FillSpecialForm {
        participant Secretary{
                ForeignerForm ff;
                createForeignerForm(out ff)post[ff|ff.oclIsNew()];
                send askFillForm(ff) to Patient block;
                receive filledForm(ff) from Patient;
```

```
                registerPerson(in ff,ssCard;out p)
                        post[p|p.oclIsNew() and
                                p.name = ssCard.name and
                                p.surname = ssCard.surname and
                                p.birthday = ssCard.birthday and
                                p.address = ff.address and
                                p.city = ff.city and
                                p.country = ssCard.country and
                                p.ssn = ssCard.number];];
        }

        participant Patient{
                ForeignerForm ff;
                receive askFillForm(ff) from Secretary;
                fillOutForm(in ff; out ff)
                        post[ff.address <> '' and ff.city <> ''];
                send filledForm(ff) to Secretary block;
        }
    }Normal
}
```

C.3 Examination business process definition

C.3.1 Process Model

```
;;*************************************************************
business process examination(out Temperature temp,
                                 BloodPressure bp)
                              last[_,30min.]{
;;*************************************************************

  participant Patient{
    receive reqProblemExplanation from Nurse;
    explainWhatIsWrong();
    spawn receive getTemp from Nurse,
          receive getBP from Assistant;
  }

  participant Nurse{
    send reqProblemExplanation to Patient;
    send getTemp to Patient block;
    repeat{
        checkTemp(out temp)
           post[temp|temp.oclIsNew() and temp < 40];
           deviation[EX_HighTemp|(temp|temp.oclIsNew() and temp > 40)]
           deviation[EX_MalfunctionThermometer|temp.oclIsUndefined()]{
                    findNewThermometer();
           }
    until(not temp.oclIsUndefined());
  }

  participant Assistant{
    send getBP to Patient block;
    repeat{
        checkBP(out bp)
                post[bp|bp.oclIsNew() and bp < 200];
                deviation[EX_HighBP|(bp|bp.oclIsNew() and bp > 200)]
                deviation[EX_MalfunctionBPMonitor|bp.oclIsUndefined()]{
                        findNewBPMonitor();
                }
    until(not bp.oclIsUndefined());
  }
}post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
```

C.3.2 Data Model

```
type Temperature { Integer t where
        (30 <= t) and (t <= 50)
}

type BloodPressure { Integer bp where
        (50 <= bp) and (bp <= 250)
}
```

C.3.3 Resource Model

```
resources{
        Nurse = Sue, Rose;
        Patient;
        Assistant = Jane;
}
```

C.3.4 Dependability Model

```
;; Binding EXEC-HANDLER
resolution{
        EX_HighTemp -> Undress;
        EX_HighBP -> CardiacUnit;
        EX_HighTemp and EX_HighBP -> UndressANDCardiacUnit;
}
;;HANDLER definition
recovery{
   Undress {
        participant Patient{
                receive reqUndress from Nurse;
                Undress();
        }

        participant Nurse{
                send reqUndress to Patient block;
        }

        participant Assistant{}

   }Normal


   CardiacUnit {
        participant Patient{
                receive reqChangeRoom from Assistant;
                goToSpecialUnit();
        }

        participant Nurse{
                receive reqChangeRoom from Assistant;
                takePatientToSpecialUnit();
        }

        participant Assistant{
                send reqChangeRoom to Patient,Nurse;
                notifyNewRoomPatient();
        }

   }Normal


   UndressANDCardiacUnit {
        participant Patient{
                receive reqChangeRoom from Assistant;
                goToSpecialUnit();
                receive reqUndress from Nurse;
                Undress();
        }

        participant Nurse{
                receive reqChangeRoom from Assistant;
                takePatientToSpecialUnit();
                send reqUndress to Patient block;
        }

        participant Assistant{
                send reqChangeRoom to Patient,Nurse;
                notifyNewRoomPatient();
        }

   }Normal
}
```

C.4 MakeDocument business process definition

C.4.1 Process Model

```
;;*************************************************************
business process makeDocument(in Person prs;
                              out PatientSheet ps) last[_,30min.]{
;;*************************************************************
  participant Nurse{
     ;;Hx = abbr. Medical History
    MedicalHistory hx;
    getHxPatient(in prs; out hx)
                post[not hx.oclIsUndefined()];
                deviation[EX_undefHx|hx.oclIsUndefined()]{
                        createtHx(in prs; out hx)
                                post[hx|hx.oclIsNew()];
                }
    makePatientSheet(in hx,prs; out ps);
                post[ps.ssn = prs.ssn and
                     ps.name = prs.name and
                     ps.surname = prs.surname and
                     ps.addres = prs.address and
                     ps.city = prs.city and
                     ps.country = prs.country and
                     ps.mh = hx]
  }
}
```

C.4.2 Data Model

```
type  Person{
        String name;
        String surname;
        Calendar birthday;
        String address;
        String city;
        String country;
        Integer ssn;
}

type Treatment,Medicine,Diagnosis,MedicalHistory,Calendar;

type PatientSheet{
        Integer ssn;
        String name;
        String surname;
        String address;
        String city;
        String country;
        MedicalHistory mh;
        Diagnosis d;
        Prescription p;
}

type Prescription{
        Treatment t;
        Medicine m;
}
```

C.4.3 Resource Model

```
resources{
        Nurse = Sue, Rose;
}
```

C.5 Consultation business process definition

C.5.1 Process Model

```
;;************************************************************
business process consultation(
          in PatientSheet ps, Temperature t, BloodPreasure bp;
          out PatientSheet ps) last[15min.,1hs.]{
;;************************************************************

  participant Patient{
    receive chek from Doctor;

    Diagnosis d;
    Prescription p;
    receive news(d,p) from Doctor;
  }

  participant Doctor{
    evaluateExaminationResults(in t,bp,ps);

    send check to Patient block;

    HealthStatus hs;
    checkPatient(out hs)
        post[hs = #GOOD]
        deviation[EX_Admission | hs = #BAD];

    Diagnosis d;
    diagnosePatient(out d)
        post[not d.oclIsUndefined()];

    Prescription p;
    prescribeTreatment(out p)
        post[not p.oclIsUndefined()];

    send news(d,p) to Patient block;

    fillPatientSheet(in ps,p,d; out ps)
        post[ps.d = d and ps.p = p]
  }
}post[not ps.d.oclIsUndefined() and not ps.p.oclIsUndefined()]
```

C.5.2 Data Model

```
type Temperature { Integer t where
        (30 <= t) and (t <= 50)}

type BloodPressure { Integer bp where
        (50 <= bp) and (bp <= 250)}

type HealthStatus = enum{GOOD,BAD};

type Treatment,Medicine,Diagnosis,MedicalHistory;

type PatientSheet{
        Integer ssn;
        String name;
        String surname;
        String address;
        String city;
        String country;
        MedicalHistory mh;
        Diagnosis d;
        Prescription p;
}
```

```
type Prescription{
        Treatment t;
        Medicine m;
}
```

C.5.3 Resource Model

```
resources{
        Patient;
        Doctor = Marc{(cost,50)}, Nick{(cost,80)};
}
```

C.5.4 Dependability Model

```
;;Binding EXEC-HANDLER
```

```
resolution{
        EX_Admission -> HospitalAdmission;
}
```

```
;;HANDLER definition
```

```
recovery{

   HospitalAdmission {
        participant Patient{
                receive reqAdmission from Doctor;
        }

        participant Doctor{
                send reqAdmission to Patient;
                notifyAdministration(in ps);
                Treatment t;
                prescribeAdmission(out t)
                        post[not t.oclIsUndefined()];
                fillPatientSheet(in ps; out ps)
                        post[not ps.p.t.oclIsUndefined()]
        }
   }degraded[ps|ps.d.oclIsUndefined() and not ps.p.t.oclIsUndefined()]

}
```

C.6 GiveInformation business process definition

C.6.1 Process Model

```
;;*************************************************************
business process giveInformation(in PatientSheet ps)
                pre[not ps.p.oclIsUndefined()]
                last[_,15min.] {
;;*************************************************************
  participant Patient{
    receive treatmentDetails from Nurse;
    receive medicineDetails from Nurse;
  }

  participant Nurse{
    checkTreatment(in ps.p.t);
    send treatmentDetails to Patient block;
    checkMedicine(ps.p.m);
    send medicineDetails to Patient block;
  }
}
```

C.6.2 Data Model

```
type PatientSheet{
        Integer ssn;
        String name;
        String surname;
        String address;
        String city;
        String country;
        MedicalHistory mh;
        Diagnosis d;
        Prescription p;
}

type Treatment,Medicine,Diagnosis,MedicalHistory;

type Prescription{
        Treatment t;
        Medicine m;
}
```

C.6.3 Resource Model

```
resources{
        Nurse = Sue, Rose;
        Patient;
}
```

D. TIMED-CAAFWRK META-MODEL

This Appendix joins in one consolidated figure all the concepts and relationships that form part of the *Timed-CaaFWrk* meta-model.

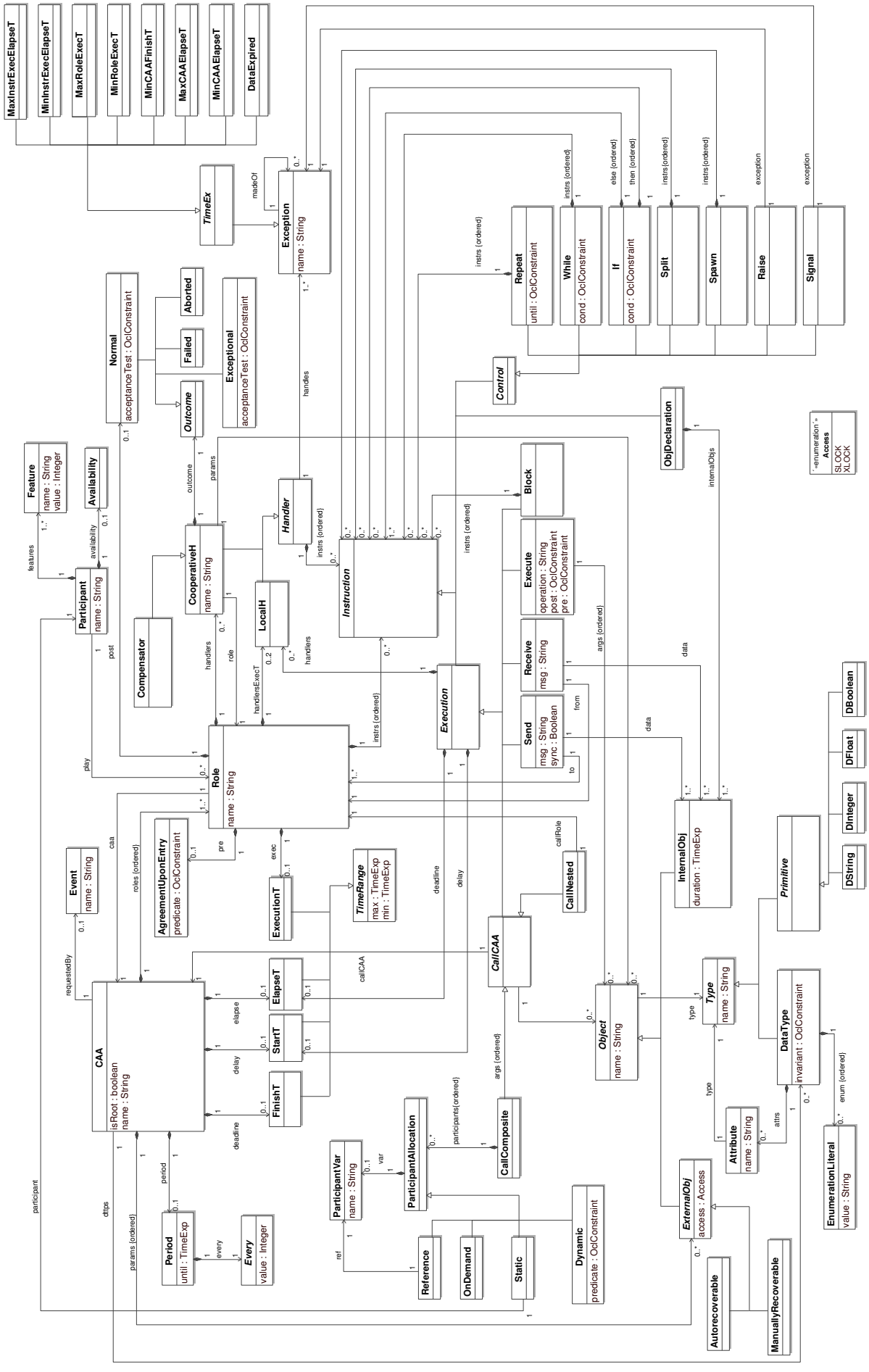

Fig. D.1: Timed-CaaFWrk Meta-Model.

E. ATL IMPLEMENTATION OF DT4BP TO TIMED-CAAFWRK TRANSFORMATION

This Appendix provides the complete list of rules that compose the model-to-model (M2M) transformation that allows the compliant *DT4BP* model to automatically obtain a compliant Timed-CaaFWrk model. These rules have been implemented using the *ATLAS Transformation Language (ATL)* [ATL10a, JK06, JABK08] (version: 3.1.0.v201005030322, build id: 201005030322).

The *DT4BP* and *Timed-CaaFWrk* meta-models on which this M2M transformation relies have been implemented as Ecore models within the Eclipse Modelling Framework (EMF) [BBM03, Ecl10] (version: 2.5.0.v200906151043, build id: 200906151043). The Ecore implementation of *DT4BP* and *Timed-CaaFWrk* meta-models are shown in Figures E.1 and E.2 that appear at the end of this Appendix.

To facilitate locating the different rules in this appendix, rules are grouped according to the overall source meta-model concern they address. They are subsequently listed according to the particular source meta-model concept each of them applys.

E.1 Header

```
module DT4BP_to_TimedCaaFWrk;

create OUT : TimedCAAFWrk from IN : DT4BP;
```

E.2 Main

```
rule Enterprise {
from
        s : DT4BP!Enterprise

to
        t : TimedCAAFWrk!CAADesign(
                name <- s.name,
                events <- s.events,
                exceptions <- OrderedSet {
                                s.exceptions,
                                tMaxInstrExecElapseT,
                                tMinInstrExecElapseT,
                                tMaxRoleExeT,
                                tMinRoleExeT
                              }->union(
                                  let res: Set(DT4BP!Resolution) =
                                    s.bps->collect(e|e.resolution)->flatten()
                                  in
                                    res->iterate(e ;
```

```
                                                excs: OrderedSet(TimedCAAFWrk!Exception) =
                                                    OrderedSet{} | excs.append(e))),
                participants <- s.resources,
                predefinedTypes <- s.basicTypes,
                dttps <- s.dttps,
                caas <- s.bps
        ),
        tMaxInstrExecElapseT : TimedCAAFWrk!MaxInstrExecElapseT(
                name <- 'maxInstrExecElapseT'
        ),
        tMinInstrExecElapseT : TimedCAAFWrk!MinInstrExecElapseT(
                name <- 'minInstrExecElapseT'
        ),
        tMaxRoleExeT : TimedCAAFWrk!MaxRoleExecT(
                name <- 'maxRoleExecT'
        ),
        tMinRoleExeT : TimedCAAFWrk!MinRoleExecT(
                name <- 'minRoleExecT'
        )
}
```

E.3 Business Processes

E.3.1 BusinessProcess

```
rule BusinessProcess {
        from
                s : DT4BP!BusinessProcess
        to
                t : TimedCAAFWrk!CAA(
                        name <- s.name,
                        requestedBy <- s.requestedBy,
                        delay <- s.start,
                        elapse <- s.last,
                        period <- s.period,
                        params <- s.params,
                        roles <- s.participants,
                        dttps <- s.dttps,
                        isRoot <- s.isRoot
                )
}
```

E.3.2 Participant

E.3.2.1 Helper: check whether a participant is the first in the BP definition

```
helper context DT4BP!Participant def: isFirstParticipant(): Boolean =
        let
                first: DT4BP!Participant = self.bp.participants->first()
        in
                self = first
        ;
```

E.3.2.2 First Participant

```
rule ParticipantFirst {
  from
      s : DT4BP!Participant(s.isFirstParticipant())
```

```
  to
      t : TimedCAAFWrk!Role(
        name <- s.name,
        caa <- s.bp,
        pre <- s.bp.pre,
        instrs <- s.stmts,
        exec <- s.workFor
        post <- s.bp.post,

        handlers <- s.bp.cooperativeHandlers->
                      collect(e|e.hndParticipants)->
                        flatten()-> select(e|e.participant=s),

        handlersExecT <- if (not s.workFor.oclIsUndefined()) then
                           OrderedSet {s.workFor.handlers}
                         else
                           OrderedSet {}
                         endif
      )

}
```

E.3.2.3 Non-First Participant

```
rule ParticipantNonFirst {
    from
        s : DT4BP!Participant (not s.isFirstParticipant())
    to
        t : TimedCAAFWrk!Role(
          name <- s.name,
          caa <- s.bp,
          instrs <- s.stmts,
          exec <- s.workFor

          handlers <- s.bp.cooperativeHandlers->
                        collect(e|e.hndParticipants)->
                          flatten()->select(e|e.participant=s),
          handlersExecT <- if (not s.workFor.oclIsUndefined()) then
                             OrderedSet {s.workFor.handlers}
                           else
                             OrderedSet {}
                           endif,
      )
}
```

E.3.3 Precondition

```
rule Precondition {
  from
      s : DT4BP!Precondition(s.refImmediateComposite().oclIsTypeOf(DT4BP!BusinessProcess))
  to
      t : TimedCAAFWrk!AgreementUponEntry(
        predicate <- s.predicate.toString()
      )
}
```

E.3.4 HandlerParticipant

E.3.4.1 Helper: check whether a *HandlerParticipant* deals with *Abort* exception

```
helper context DT4BP!HandlerParticipant def: dealsWithAbort(): Boolean =
  let
      res: Set(DT4BP!Resolution)=self.actsIn.refImmediateComposite().resolution->
                      select(r|r.right = self.actsIn)
  in
      res-> exists(r | r.left-> exists(ex|ex.oclIsTypeOf(DT4BP!Abort)));
```

E.3.4.2 Handler not dealing with *Abort* exception

```
rule HandlerParticipant {
 from
  s : DT4BP!HandlerParticipant(not s.dealsWithAbort())
 to
  t : TimedCAAFWrk!CooperativeH(
      name <- s.actsIn.name,
      role <- s.participant,
      outcome <- s.actsIn.outcome,
      instrs <- s.stmts,
      handles <- let res: Set(DT4BP!Resolution) =
                    s.actsIn.refImmediateComposite().resolution->
                                        select(r|r.right = s.actsIn)
                 in
                 res->iterate(e; excs:OrderedSet(TimedCAAFWrk!Exception) = OrderedSet{}|
                           if(res->exists(ex | ex.left.size() > 1)) then
                                   excs.append(thisModule.resolveTemp(e, 't'))
                           else
                                   excs.append(res->collect(ex2| ex2.left))
                           endif),
      params <- s.participant.bp.params->
                      union(s.participant.stmts->
                          select(e|e.oclIsTypeOf(DT4BP!ObjDecl))->collect(e|e.var))
    )
}
```

E.3.4.3 Handler dealing with *Abort* exception

```
rule HandlerParticipant4Abort {
 from
  s : DT4BP!HandlerParticipant(s.dealsWithAbort())
 to
  t : TimedCAAFWrk!Compensator(
      name <- s.actsIn.name,
      role <- s.participant,
      outcome <- s.actsIn.outcome,
      instrs <- s.stmts,
      handles <- let res: Set(DT4BP!Resolution) =
                      s.actsIn.refImmediateComposite().resolution->
                                        select(r|r.right = s.actsIn)
                 in
                 res->iterate(e ;excs:OrderedSet(TimedCAAFWrk!Exception) = OrderedSet{}|
                                   if(res->exists(ex | ex.left.size() > 1)) then
                                           excs.append(thisModule.resolveTemp(e, 't'))
                                   else
                                           excs.append(res->collect(ex2| ex2.left))
                                   endif),
      params <- s.participant.bp.params->
                    union(s.participant.stmts->
                        select(e|e.oclIsTypeOf(DT4BP!ObjDecl))->collect(e|e.var))
    )
}
```

E.3.5 Resolution

```
rule ConcurrentExceptions {
  from
      s: DT4BP!Resolution (s.left->size() > 1)
  to
      t: TimedCAAFWrk!Exception (
              name <- s.left->
                        iterate(e; res : String = 'CC_EX' |
                          res.concat('_'.concat(
                                        e.name.toString().regexReplaceAll('EX_|CC_EX_','')
                                      )
                                   )
                                ),
              madeOf <- s.left
      )
}
```

E.3.6 Event

```
rule Event {
        from
                s : DT4BP!Event
        to
                t : TimedCAAFWrk!Event(
                        name <- s.name
                )
}
```

E.3.7 Resource

```
rule Resource {
        from
                s : DT4BP!Resource
        to
                t : TimedCAAFWrk!Participant(
                        name <- s.name,
                        features <- s.capabilities,
                        play <- s.mightParticipateAs,
                        availability <- s.availability
                )
}
```

E.3.8 Capability

```
rule Capability {
        from
                s : DT4BP!Capability
        to
                t : TimedCAAFWrk!Feature(
                        name <- s.name,
                        value <- s.value
                )
}
```

E.3.9 Calendar

```
rule Calendar {
        from
                s : DT4BP!Calendar
        to
                t : TimedCAAFWrk!Availability(
                        diary <- s.agenda
                )
}
```

E.4 Objects

E.4.1 Object

```
abstract rule Object {
        from
                s : DT4BP!Object
        to
                t : TimedCAAFWrk!Object(
                        name <- s.name,
                        type <- s.type
                )
}
```

E.4.2 LocalObject

```
rule LocalObject extends Object {
        from
                s : DT4BP!LocalObject
        to
                t : TimedCAAFWrk!InternalObj(
                        duration <- s.duration
                )
}
```

E.4.3 Parameter

```
rule Parameter extends Object {
        from
                s : DT4BP!Parameter
        to
                t : TimedCAAFWrk!ManuallyRecoverable(
                        access <- if(s.kind.toString() = 'in') then
                                        'SLOCK'
                                  else
                                        'XLOCK'
                                  endif
                        )
}
```

E.5 Statements

E.5.1 ObjDecl

```
rule ObjDecl {
        from
                s : DT4BP!ObjDecl
        to
                t : TimedCAAFWrk!ObjDeclaration(
                        internalObjs <- s.var
                )
}
```

E.5.2 Repeat

```
rule Repeat {
        from
                s : DT4BP!Repeat
        to
                t : TimedCAAFWrk!Repeat (
                        until <- s.until.toString(),
                        instrs <- s.do
                )
}
```

E.5.3 While

```
rule While {
        from
                s : DT4BP!While
        to
                t : TimedCAAFWrk!While (
                        cond <- s.cond.toString(),
                        instrs <- s.do
                )
}
```

E.5.4 If

```
rule If {
        from
                s : DT4BP!If
        to
                t : TimedCAAFWrk!If (
                        cond <- s.cond.toString(),
                        then <- s.then,
                        else <- s.else
                )
}
```

E.5.5 Split

```
rule Split {
        from
                s : DT4BP!Split
        to
                t : TimedCAAFWrk!Split (
                        instrs <- s.activities
                )
}
```

E.5.6 Spawn

```
rule Spawn {
        from
                s : DT4BP!Spawn
        to
                t : TimedCAAFWrk!Split (
                        instrs <- s.activities
                )
}
```

E.5.7 Block

```
rule Block {
        from
                s : DT4BP!Block
        to
                t : TimedCAAFWrk!Block (
                        instrs <- OrderedSet {s.stmts, s.deviations},
                        handlers <- s.localHandlers,
                        delay <- s.in,
                        deadline <- s.within,
                        isTry <- true
                )
}
```

E.5.8 Receive

```
rule Receive {
        from
                s : DT4BP!Receive
        to
                t : TimedCAAFWrk!Receive(
                        from <- s.from,
                        msg <- s.msg,
                        data <- s.args
                )
}
```

E.5.9 Send

```
rule Send {
        from
                s : DT4BP!Send
        to
                t : TimedCAAFWrk!Send(
                        to <- s.to,
                        msg <- s.msg,
                        data <- s.args,
                        sync <- s.block
                )
}
```

E.5.10 Atomic

E.5.10.1 Atomic with Postcondition

```
rule AtomicWithPost {
  from
          s : DT4BP!Atomic(not s.post.oclIsUndefined())
  to
          t : TimedCAAFWrk!Block(
                  instrs <- OrderedSet {t2,t3},
                  isTry <- false
          ),
          t2 : TimedCAAFWrk!Execute(
                  operation <- s.name,
                  args <- s.args-> collect (e | e.obj),
                  delay <- s.in,
                  deadline <- s.within,
                  pre <- if(not s.pre.oclIsUndefined()) then
                            s.pre.predicate
                         else
                            'true'
                         endif,
                  post <- s.post.predicate,
                  handlers <- if (not s.within.oclIsUndefined()) then
                                OrderedSet {s.localHandlers, s.within.handlers}
                              else
                                OrderedSet {s.localHandlers}
                              endif
          ),
          t3 : TimedCAAFWrk!If(
                  cond <- 'not('.concat(s.post.predicate.toString().concat(')')),
                  then <- OrderedSet {s.deviations,t4}
          ),
          t4 : TimedCAAFWrk!Execute(
                  operation <- 'System.exit()',
                  pre <- 'true',
                  post <- 'true'
          )
}
```

E.5.10.2 Atomic without Postcondition

```
rule AtomicWithoutPost {
  from
        s : DT4BP!Atomic(s.post.oclIsUndefined())
  to
        t : TimedCAAFWrk!Execute(
            operation <- s.name,
            args <- s.args-> collect (e | e.obj),
            delay <- s.in,
            deadline <- s.within,
            pre <- if(not s.pre.oclIsUndefined()) then
                      s.pre.predicate
                   else
                      'true'
                   endif,
            post <- 'true',
            handlers <- if (not s.within.oclIsUndefined()) then
                          OrderedSet {s.localHandlers, s.within.handlers}
                        else
                          OrderedSet {s.localHandlers}
                        endif
        )
}
```

E.5.11 Composite

E.5.11.1 Composite with Postcondition

```
rule CompositeWithPost {
  from
          s : DT4BP!Composite(not s.call.post.oclIsUndefined())
  to
          t : TimedCAAFWrk!Block(
                  instrs <- OrderedSet {t2,t3},
                  isTry <- false
          ),
          t2 : TimedCAAFWrk!CallComposite(
                  callCAA <- s.call,
                  participants <- s.resources,
                  args <- s.args-> collect (e | e.obj),
                  delay <- s.in,
                  deadline <- s.within,
                  handlers <- if (not s.within.oclIsUndefined()) then
                                  OrderedSet {s.localHandlers, s.within.handlers}
                              else
                                  OrderedSet {s.localHandlers}
                              endif
          ),
          t3 : TimedCAAFWrk!If(
                  cond <- 'not('.concat(s.call.post.predicate.toString().concat(')')),
                  then <- OrderedSet {s.deviations,t4}
          ),
          t4 : TimedCAAFWrk!Execute(
                  operation <- 'System.exit()',
                  pre <- 'true',
                  post <- 'true'
          )
}
```

E.5.11.2 Composite without Postcondition

```
rule CompositeWithoutPost {
  from
        s : DT4BP!Composite(s.call.post.oclIsUndefined())
  to
        t : TimedCAAFWrk!CallComposite(
                callCAA <- s.call,
                participants <- s.resources,
                args <- s.args-> collect (e | e.obj),
                delay <- s.in,
                deadline <- s.within,
                handlers <- if (not s.within.oclIsUndefined()) then
                                OrderedSet {s.localHandlers, s.within.handlers}
                            else
                                OrderedSet {s.localHandlers}
                            endif
        )
}
```

E.5.12 Nested

E.5.12.1 Nested with Postcondition

```
rule NestedWithPost {
  from
        s : DT4BP!Nested(not s.call.post.oclIsUndefined())
```

```
  to
        t : TimedCAAFWrk!Block(
                instrs <- OrderedSet {t2,t3},
                isTry <- false
        ),
        t2 : TimedCAAFWrk!CallNested(
                callCAA <- s.call,
                callRole <- s.call.participants->
                    select(p | p.name=s.refImmediateComposite().name),
                args <- s.args-> collect (e | e.obj),
                delay <- s.in,
                deadline <- s.within,
                handlers <- if (not s.within.oclIsUndefined()) then
                                OrderedSet {s.localHandlers, s.within.handlers}
                            else
                                OrderedSet {s.localHandlers}
                            endif
        ),
        t3 : TimedCAAFWrk!If(
                cond <- 'not('.concat(s.call.post.predicate.toString().concat(')')),
                then <- OrderedSet {s.deviations,t4}
        ),
        t4 : TimedCAAFWrk!Execute(
                operation <- 'System.exit()',
                pre <- 'true',
                post <- 'true'
        )
}
```

E.5.12.2 Nested without Postcondition

```
rule NestedWithoutPost {
  from
          s : DT4BP!Nested(s.call.post.oclIsUndefined())
  to
          t : TimedCAAFWrk!CallNested(
                  callCAA <- s.call,
                  callRole <- s.call.participants->
                      select(p | p.name=s.refImmediateComposite().name),
                  args <- s.args-> collect (e | e.obj),
                  delay <- s.in,
                  deadline <- s.within,
                  handlers <- if (not s.within.oclIsUndefined()) then
                                  OrderedSet {s.localHandlers, s.within.handlers}
                              else
                                  OrderedSet {s.localHandlers}
                              endif
          )
}
```

E.6 Exceptions

E.6.1 Exception

```
rule Exception {
        from
                s : DT4BP!Exception
        to
                t : TimedCAAFWrk!Exception(
                        name <- s.name
                )
}
```

E.6.2 TimeEx

```
abstract rule TimeEx extends Exception {
        from
                s : DT4BP!TimeEx
        to
                t : TimedCAAFWrk!TimeEx()
}
```

E.6.3 DataExpired

```
rule DataExpired extends TimeEx {
        from
                s : DT4BP!DataExpired
        to
                t : TimedCAAFWrk!DataExpired()
}
```

E.6.4 TimeoutMinBPLast

```
rule TimeoutMinBPLast extends TimeEx {
        from
                s : DT4BP!TimeoutMinBPLast
        to
                t : TimedCAAFWrk!MinCAAElapseT()
}
```

E.6.5 TimeoutMaxBPLast

```
rule TimeoutMaxBPLast extends TimeEx {
        from
                s : DT4BP!TimeoutMaxBPLast
        to
                t : TimedCAAFWrk!MaxCAAElapseT()
}
```

E.7 Outcomes

E.7.1 Outcome

```
abstract rule Outcome  {
    from
        s : DT4BP!Outcome
    to
        t : TimedCAAFWrk!Outcome()
}
```

E.7.2 Normal

```
rule Normal {
    from
        s : DT4BP!Normal
    to
        t : TimedCAAFWrk!Normal(
            acceptanceTest <- s.predicate
        )
}
```

E.7.3 Degraded

```
rule Degraded extends Outcome {
    from
        s : DT4BP!Degraded
    to
        t : TimedCAAFWrk!Exceptional(
            acceptanceTest <- s.predicate.toString()
        )
}
```

E.7.4 Aborted

```
rule Aborted extends Outcome {
    from
        s : DT4BP!Aborted
    to
        t : TimedCAAFWrk!Aborted()
}
```

E.7.5 Failed

```
rule Failed extends Outcome {
    from
        s : DT4BP!Failed
    to
        t : TimedCAAFWrk!Failed()
}
```

E.8 Deviations

E.8.1 PHandler

E.8.1.1 PHandler belonging to an *Activity*

```
rule PHandlerForActivity {
  from
        s : DT4BP!PHandler(s.refImmediateComposite().oclIsKindOf(DT4BP!Activity))
  to
        t : TimedCAAFWrk!LocalH(
                instrs <- s.stmts,
                handles <- s.handles->collect(dev|dev.raise)
        )
}
```

E.8.1.2 PHandler belonging to a time-related constraint *Within*

```
rule PHandlerForWithin {
  from
        s : DT4BP!PHandler(s.refImmediateComposite().oclIsTypeOf(DT4BP!Within))
  to
        t : TimedCAAFWrk!LocalH(
              instrs <- s.stmts,
              handles <- if(s.refImmediateComposite().timeout
                              -> exists(timeDev | timeDev.kind.toString() = 'min_max'))
                         then OrderedSet {TimedCAAFWrk!MinInstrExecFinishT.
                                          allInstances()
                                          -> asSequence()
                                          -> first(),
                                        TimedCAAFWrk!MaxInstrExecFinishT.
                                          allInstances()
                                          -> asSequence()
                                          -> first()}
                         else
                              if(s.refImmediateComposite().timeout
                                  -> exists(timeDev | timeDev.kind.toString() = 'min'))
                              then TimedCAAFWrk!MinInstrExecFinishT.
                                          allInstances()
                                          -> asSequence()
                                          -> first()
                              else TimedCAAFWrk!MaxInstrExecFinishT.
                                          allInstances()
                                          -> asSequence()
                                          -> first()
                              endif
                         endif
        )
}
```

E.8.1.3 PHandler belonging to a time-related constraint *WorkFor*

```
rule PHandlerForWorkFor {
  from
        s : DT4BP!PHandler(s.refImmediateComposite().oclIsTypeOf(DT4BP!WorkFor))
  to
        t : TimedCAAFWrk!LocalH(
              instrs <- s.stmts,
              handles <- if(s.refImmediateComposite().timeout
                              -> exists(timeDev | timeDev.kind.toString() = 'min')
                         and
                              s.refImmediateComposite().timeout
                              -> exists(timeDev | timeDev.kind.toString() = 'max') )
                         then OrderedSet {TimedCAAFWrk!MinRoleExecT.
                                          allInstances()
                                          -> asSequence()
                                          -> first(),
                                        TimedCAAFWrk!MaxRoleExecT.
                                          allInstances()
                                          -> asSequence()
                                          -> first()}
                         else if(s.refImmediateComposite().timeout
                                  -> exists(timeDev | timeDev.kind.toString() = 'min'))
                              then TimedCAAFWrk!MinRoleExecT.
                                          allInstances()
                                          -> asSequence()
                                          -> first()
                              else TimedCAAFWrk!MaxRoleExecT.
                                          allInstances()
                                          -> asSequence()
                                          -> first()
                              endif
                         endif
```

```
        )
}
```

E.8.2 ActivityDeviation

```
rule ActivityDeviation {
  from
        s : DT4BP!ActivityDeviation
  to
        t : TimedCAAFWrk!If(
                cond <- if s.condition.oclIsTypeOf(DT4BP!Predicate) then
                                s.condition.predicate.toString()
                        else
                            if s.condition.oclIsTypeOf(DT4BP!AbortDeviation) then
                                'outcome=aborted'
                            else
                                if s.condition.oclIsTypeOf(DT4BP!FailureDeviation) then
                                    'outcome=failed'
                                else
                                    'dataExpired'
                                endif
                            endif
                        endif,
                then <- t2
        ),
        t2 : TimedCAAFWrk!Raise(
                exception <- s.raise
        )
}
```

E.9 Resources

E.9.1 ResourceAllocation

```
abstract rule ResourceAllocation {
        from
                s : DT4BP!ResourceAllocation
        to
                t : TimedCAAFWrk!ParticipantAllocation(
                        var <- s.var
                )
}
```

E.9.2 ResourceVar

```
rule ResourceVar {
        from
                s : DT4BP!ResourceVar
        to
                t : TimedCAAFWrk!ParticipantVar(
                        name <- s.name
                )
}
```

E.9.3 Reference

```
rule Reference extends ResourceAllocation {
        from
                s : DT4BP!Reference
        to
                t : TimedCAAFWrk!Reference(
                        ref <- s.ref
                )
}
```

E.9.4 Dynamic

```
rule Dynamic extends ResourceAllocation {
        from
                s : DT4BP!Dynamic
        to
                t : TimedCAAFWrk!Dynamic(
                        predicate <- s.pred.toString()
                )
}
```

E.9.5 Static

```
rule Static extends ResourceAllocation {
        from
                s : DT4BP!Static
        to
                t : TimedCAAFWrk!Static(
                        participant <- s.resource
                )
}
```

E.9.6 OnDemand

```
rule OnDemand extends ResourceAllocation {
        from
                s : DT4BP!OnDemand
        to
                t : TimedCAAFWrk!OnDemand()
}
```

E.10 Data types

E.10.1 Type

```
abstract rule Type {
        from
                s : DT4BP!Type
        to
                t : TimedCAAFWrk!Type(
                        name <- s.name
                )
}
```

E.10.2 DInteger

```
rule DInteger extends Type {
        from
                s: DT4BP!DInteger
        to
                t: TimedCAAFWrk!DInteger()
}
```

E.10.3 DString

```
rule DString extends Type {
        from
                s: DT4BP!DString
        to
                t: TimedCAAFWrk!DString()
}
```

E.10.4 DBoolean

```
rule DBoolean extends Type {
        from
                s: DT4BP!DBoolean
        to
                t: TimedCAAFWrk!DBoolean()
}
```

E.10.5 DFloat

```
rule DFloat extends Type {
        from
                s: DT4BP!DFloat
        to
                t: TimedCAAFWrk!DFloat()
}
```

E.10.6 DataType

```
rule DataType extends Type {
    from
        s: DT4BP!DataType
    to
        t: TimedCAAFWrk!DataType (
                --name <- s.name,
                attrs <- s.attrs,
                enum <- s.enum,
                invariant<- if not s.invariant.oclIsUndefined()then
                                        s.invariant.toString()
                                else
                                        ''
                                endif
        )
}
```

E.10.7 EnumerationLiteral

```
rule EnumerationLiteral {
   from
                s: DT4BP!EnumerationLiteral
   to
                t: TimedCAAFWrk!EnumerationLiteral (
                        value <- s.value
                )
}
```

E.10.8 Attribute

```
rule Attribute {
   from
                s: DT4BP!Attribute
   to
                t: TimedCAAFWrk!Attribute (
                        name <- s.name,
                        type <- s.type
                )
}
```

E.11 Time

E.11.1 TimeRange

```
abstract rule TimeRange {

        from
                s : DT4BP!TimeRange
        to
                t : TimedCAAFWrk!TimeRange(
                        min <- s.min,
                        max <- s.max
                )
}
```

E.11.2 Start

```
rule Start extends TimeRange {
        from
                s : DT4BP!Start
        to
                t : TimedCAAFWrk!StartT()
}
```

E.11.3 Last

```
rule Last {
        from
                s : DT4BP!Last
        to
                t1:TimedCAAFWrk!FinishT(
                        min <- s.min,
```

```
                        max <- s.max
                ),
                t2 : TimedCAAFWrk! ElapseT (
                        min <- s.min ,
                        max <- s.max
                )
}
```

E.11.4 WorkFor

```
rule WorkFor extends TimeRange {
        from
                s : DT4BP! WorkFor
        to
                t : TimedCAAFWrk! ExecutionT (
                )
}
```

E.11.5 Within

```
rule Within extends TimeRange {
        from
                s : DT4BP! Within
        to
                t : TimedCAAFWrk! ElapseT ()
}
```

E.11.6 In

```
rule In extends TimeRange {
        from
                s : DT4BP! In
        to
                t : TimedCAAFWrk! StartT (
                        min <- s.delay ,
                        max <- s.delay
                )
}
```

E.11.7 Period

```
rule Period {
        from
                s : DT4BP! Period
        to
                t : TimedCAAFWrk! Period (
                        until <- s.until ,
                        every <- s.every
                )
}
```

E.11.8 Every

```
abstract rule Every {
        from
                s : DT4BP!Every
        to
                t : TimedCAAFWrk!Every(
                        value <- s.value
                )
}
```

E.11.9 Second

```
rule Second extends Every {
        from
                s : DT4BP!Second
        to
                t : TimedCAAFWrk!Second()
}
```

E.11.10 Hour

```
rule Hour extends Every {
        from
                s : DT4BP!Hour
        to
                t : TimedCAAFWrk!Hour()
}
```

E.11.11 Day

```
rule Day extends Every {
        from
                s : DT4BP!Day
        to
                t : TimedCAAFWrk!Day()
}
```

E.11.12 Week

```
rule Week extends Every {
        from
                s : DT4BP!Week
        to
                t : TimedCAAFWrk!Week()
}
```

E.11.13 Month

```
rule Month extends Every {
        from
                s : DT4BP!Month
        to
                t : TimedCAAFWrk!Month()
}
```

E.11.14 Year

```
rule Year extends Every {
        from
                s : DT4BP!Year
        to
                t : TimedCAAFWrk!Year()
}
```

E.11.15 Monday

```
rule Monday extends Every {
        from
                s : DT4BP!Monday
        to
                t : TimedCAAFWrk!Monday()
}
```

E.11.16 Tuesday

```
rule Tuesday extends Every {
        from
                s : DT4BP!Tuesday
        to
                t : TimedCAAFWrk!Tuesday()
}
```

E.11.17 Wednesday

```
rule Wednesday extends Every {
        from
                s : DT4BP!Wednesday
        to
                t : TimedCAAFWrk!Wednesday()
}
```

E.11.18 Thursday

```
rule Thursday extends Every {
        from
                s : DT4BP!Thursday
        to
                t : TimedCAAFWrk!Thursday()
}
```

E.11.19 Friday

```
rule Friday extends Every {
        from
                s : DT4BP!Friday
        to
                t : TimedCAAFWrk!Friday()
}
```

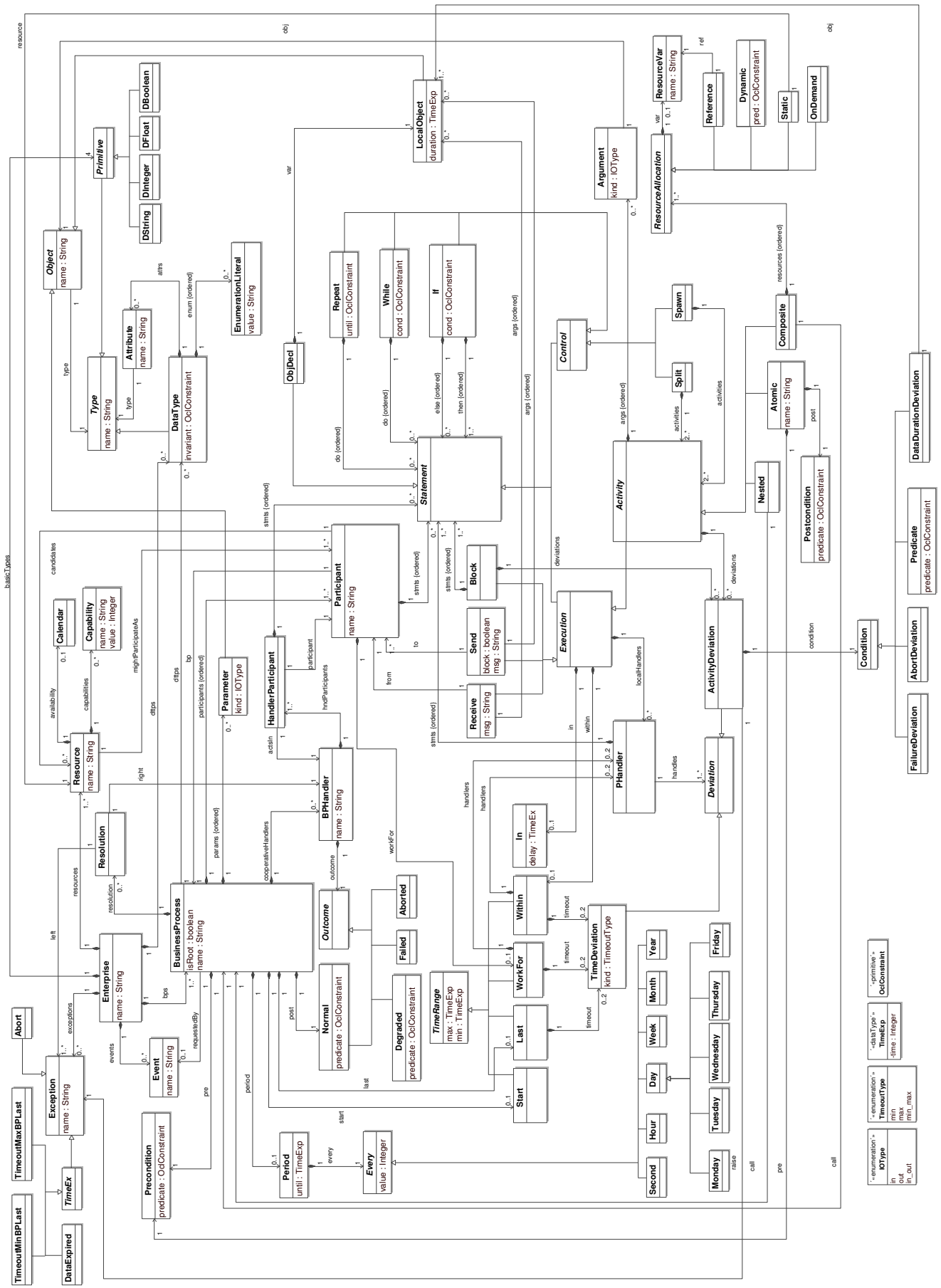

Fig. E.1: Ecore DT4BP meta-model used as input by the ATL transformation.

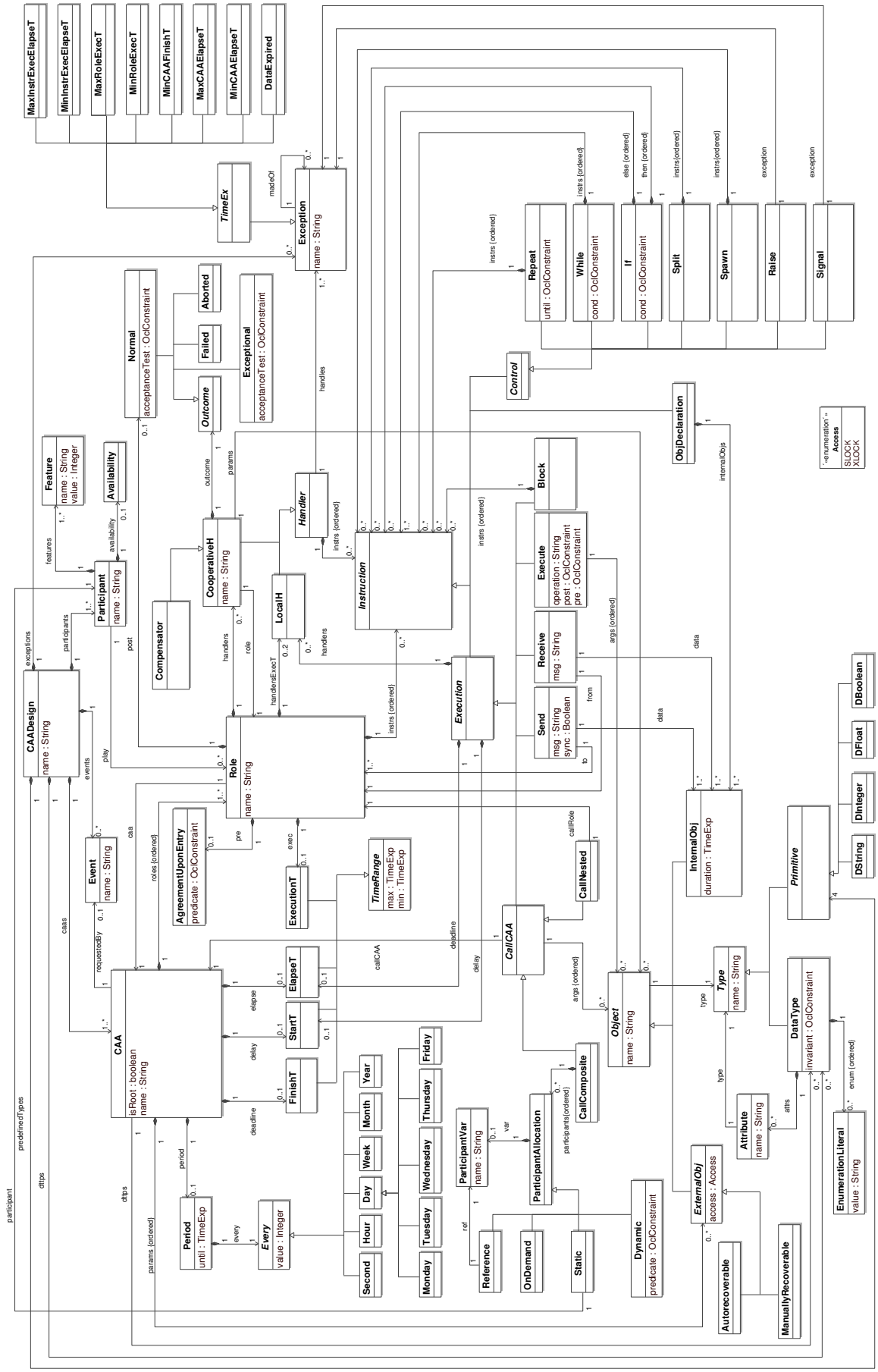

Fig. E.2: Ecore Timed-CAA meta-model used as output by the ATL transformation.

F. CURRICULUM VITAE

Alfredo Capozucca, born on the 13th of November, 1976 in Casilda, Santa Fe, Argentina.

Education

1. **January 2007 - December 2010**: PhD thesis in the field of dependability, collaboration, real-time and business processes modelling.
2. **March 1995 - December 2003**: Licenciado en Ciencias de la Computación (Ms. in Computer Science, five-year degree with final project, final mark: 7.94/10), Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Universidad Nacional de Rosario, Argentina.

Employment Experience

1. *July 2004 - December 2006*
   LASSY - Laboratory for Advanced Software Systems, Faculty of Science, Technology and Communication of the University of Luxembourg.

   Scientific collaborator in the research project CORRECT. The main role comprises designing and developing case studies that experiments with fault tolerance mechanisms like CA actions, in the context of web services in e-health domain.

2. *March 2003 - June 2004*
   Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Universidad Nacional de Rosario, Argentina

   Teaching assistant in Systems Analysis and Software Engineering. These subjects are delivered in the first and the second semesters respectively, in the fourth year of the course Licenciatura en Ciencias de la Computación.

3. *June 2001 - June 2002*
   ATX Software S.A., Lisbon, Portugal

   Analysis, design and development of systems for web applications using Java, JSP, ASP, VBScript and using Apache-Tomcat and Internet Information Server (IIS) as web servers. Member of a testing team for re-engineering tool development with Java. This tool was used to adapt Cobol programs for the Bank Espíritu Santo (Portugal's first bank) due to the incorporation of the Euro as CEE official currency, since Portugal is a member.

4. *October 1998 - March 2001*
   GPL Systems, Rosario, Argentina

   Development of bank systems as programmer and project leader. Responsible for internal training of personnel on tools and procedures use in the company for the development and maintenance systems.

Publications

1. A. Capozucca, N. Guelfi; *Modelling dependable collaborative time-constrained business processes*, in Enterprise Information Systems, Volume 4, Issue 2, pp. 153-214, May 2010.

2. A. Capozucca, N. Guelfi; *Designing reliable real-time concurrent object-oriented software systems*; in Proceedings of the 2009 ACM Symposium on Applied Computing SAC '09, pp. 1996-1997, Honolulu, Hawaii, USA, March 2009. **Best Poster Award (out of 96)**.

3. A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky, A. F. Zorzo; *Frameworks for designing and implementing dependable systems using Coordinated Atomic Actions: A comparative study*; in Journal of Systems and Software, 82, 2, pp. 207-228, February 2009.

4. F. Balbastro, A. Capozucca, N. Guelfi; *Analysis and Framework-based Design of a Fault-Tolerant Web Information System for m-Health*; Special Issue on Service Intelligence & Service Science, Service Oriented Computing and Applications Journal, 2 (Numbers 2-3), pp. 111-144, Springer London, 2008.

5. F. Balbastro, A. Capozucca, N. Guelfi; *On the Integration of Mobility in a Fault-Tolerant e-HealthWeb Information System*; International Workshop on Web and Mobile Information Services (WAMIS), Niagara Falls , Canada, IEEE CS Press as IEEE 21st AINA Workshops Proceedings, 2007.

6. A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky, A. Zorzo; *CAA-DRIP: a framework for implementing Coordinated Atomic Actions*; The 17th International Symposium on Software Reliability, Raleigh, North Carolina, USA, 2006.

7. A. Capozucca, N. Guelfi, P. Pelliccione; *The Fault-Tolerant Insulin Pump Therapy*; book chapter in Rigorous Development of Complex Fault-Tolerant Systems, LNCS 4157, Lecture Notes in Computer Sciences, Springer-Verlag, Berlin Heidelberg 2006, pp. 59-79, 2006.

8. A. Capozucca, B. Gallina, N. Guelfi, P. Pelliccione; *Modeling Exception Handling: a UML2.0 Platform Independent Profile for CAA*; in proceedings of the ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems, Glasgow (Scotland), Department of Computer Science. LIRMM; Montpellier-II University, 2005.

9. A. Capozucca, N. Guelfi, P. Pelliccione; *The Fault-Tolerant Insulin Pump Therapy*; in proceedings of the Workshop on Rigorous Engineering of Fault Tolerant Systems Event Information, in conjunction with Formal Methods 2005, University of Newcastle upon Tyne, UK, 2005.