Stochastic Modelling & Analysis of Dynamic Human-Resource Allocation (StADy)

Thesis submitted for the degree of
Doctor of Philosophy at the University of
Leicester

by
Adwoa Dansoa Donyina

Hons. BSc.(University of Toronto 2007)
MSc Distinction (University of Leicester 2008)

Department of Computer Science
University of Leicester

April 2011

Abstract

Thesis Title: Stochastic Modelling & Analysis of Dynamic Human-Resource Allocation (StADy)
Author's Full name: Adwoa Dansoa Donyina

Business processes require involvement of technical components as well as humans to achieve their objectives. However, humans are only predictable to a degree of certainty because, while guided by policies and regulations, they retain the freedom to ignore established procedures or positively react to unforeseen events. Since we cannot change people, we have to be able to recognize their unpredictable behaviour by organising processes in such a way as to benefit from the flexibility of their actions and deal with the problems that arise from it. Business processes tend to be a structured sequence of events; however the assignment of humans to scheduled cases is unstructured. Hence, it is difficult to accurately model and simulate the flexibility of human resource allocation without considering the impact of unpredictable human behaviour.

While business processes often have a rigid structure, determining sequences of actions on each individual case, there is flexibility in the selection of cases to be processed as well as in the assignment of human resources. However, such a flexible use of resources poses its own challenges, making process execution difficult to model and predict.

In this thesis I propose a methodology and language to support the modelling and evaluation of business process executions with flexible assignment of human resources. The main idea is to model configurations of a business process as graphs and use graph transformation rules in a UML-like syntax to describe the process execution. This model allows to define conditions to temporarily permit actors to exceed their roles in exceptional (escalated) situations, without causing legal repercussions.

The evaluation of process execution models is supported by the use of stochastic graph transformations, which allow the qualitative analysis of different organizational policies through simulation.

The methodology is presented in four stages of (1) business modelling, (2) process execution design, (3) process encoding and (4) performance evaluation. A case study of a pharmacy process is used to evaluate the approach.

Author's Declaration

I hereby declare that this submission is my own work and that is the result of work done during the period of registration. To the best of my knowledge, it contains no previously published material written by another person. None of this work has been submitted for another degree at University of Leicester or any other University.

Parts of this thesis submission appeared in the following conjoint publications, to each of which I have made substantial contributions:

1. Adwoa Donyina and Reiko Heckel. Modelling Flexible Human Resource Allocation by Stochastic Graph Transformation. Post-Proceedings of the Fifth International Conference on Graph Transformation - Doctoral Symposium (ICGT-DS 2010) ECEASST Journal (1863-2122), Volume 38/2011.

2. Adwoa Donyina and Reiko Heckel. Flexible Behaviour of Human Actors in Distributed Workflows. In proceedings of the 17th Conference on "Communication in Distributed Systems 2011" (KiVS'11) in Kiel, Germany (March, 8-11, 2011). As a special issue of the ECEASST Journal (ISSN 1863-2122), Volume 37/2011, pages 134-145.

3. Adwoa Donyina and Reiko Heckel. Formal Visual Modeling of Human Agents in Service Oriented Systems. In 2009 Fourth South-East European Workshop on Formal Methods (SEEFM'09), pages 25-32, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

Parts of this thesis also appeared in the following independent publication:

1. Adwoa Donyina. Stochastic Modelling and Simulation of Dynamic Resource Allocation. Fifth International Conference on Graph Transformation Doctoral Symposium (ICGT-DS 2010) University of Twente, Enschede, The Netherlands 27 September - 2 October, 2010. Graph Transformations Lecture Notes in Computer Science, 2010, Volume 6372/2010, pages 388-390.

Acknowledgements

First and Foremost the author would like to thank The Lord Jesus Christ for guiding her throughout her studies. Also, this thesis would not have been possible without the support of many people. The author wishes to express her gratitude to her first and second supervisors, Prof. Dr. Reiko Heckel and Prof. Dr. José Fiadeiro who were abundantly helpful and offered invaluable assistance, support and guidance. Deepest gratitude goes to pharmacist/owner Georgina Donyina whose expert pharmacy domain knowledge assisted in a realistic depiction of the project's methodology.

Special thanks also go to her colleagues and friends for their suggestions and editing of this work. Thank you to all the members of the Computer Science Department, University of Leicester, United Kingdom, who became like an extended family, by making her stay in England a home away from home. The author wishes to express her love and gratitude to her beloved family and friends in Canada for their understanding and endless love, through the duration of her studies.

Figure 1: Adwoa Donyina and Her Majesty the Queen of England
Location: University of Leicester, United Kingdom
Date: 4 December 2008
Description: Presenting her MSc thesis work

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Process management is a collection of planning, organising, and controlling activities for the goal-oriented management of a series of administrative activities aimed at identifying opportunities for improvement in quality, time, cost and customer satisfaction factors [46, 35]. Decision-making is an important aspect of process management; hence it is divided into levels depending on the type of decision-making that is involved, such as decisions on process design, resource planning, resource assignment and equipment control, which occur at strategic, tactical, operational and real-time levels respectively [70]. Each of these process management levels go through the following four basic phases to decision-making [70]:

1. Defining the problem.

2. Formulating solution(s) to the problem.

3. Assessing the different solutions.

4. Implementing the best solution.

The scope of this thesis is the strategic and operational levels of process-modelling decisions. Strategic management decisions are process design decisions that occur yearly and have large financial impact, which can remain noticeable for years. Operational management level is concerned with resource assignment decisions. Phases one and two can be accomplished by enabling business experts to describe existing and prospective business processes precisely, using a simple graphical syntax to define the activities and policies for the process design. The third phase can be accomplished through business simulation, which is based on testing the modified business processes using historical data. A simulation model is a particular type of mathematical model of a system, which provides businesses with the means to test scheduling protocols, policies and regulations, prior to employing them in their day-to-day operation. With the goal of increasing business productivity in mind, simulations can also help in the selection of an ideal solution due to simulation tool's analysis facility, which leads to phase four. Simulations are very flexible analytical techniques, which can aid in answering a series of questions. Simulations can provide a good insight of the process being modelled through repeated execution of a process with the aid of a computer. Since the development of a simulation model aids in process-modelling decision making, it can be used to make protocol decisions about human-resource allocation and task management. However it is difficult to model dynamic role allocation accurately because, in contrast to computer systems, human behaviour is only predictable to a degree of certainty. In semi-automated business processes human actors are guided by predetermined policies and regulations, but retain the freedom to react to unforeseen events, or disregard procedures. Business processes are classified as being structured or unstructured [40]. A structured business process is predictable routine work, whereas an unstructured business process is unpredictable knowledge work. While the processes themselves tend to be structured, the execution level is not necessary structured.

BPEL4People is an extension of Web Services Business Process Execution Language *(WS-BPEL) 2.0* [63] that incorporates activities performed by people, using capabilities such as data manipulation. The authors of BPEL4People stress in their whitepages that "the aspect of how people interact with business processes must be properly modelled" [34]. The modelling of the orchestration of people and technical components should incorporate the various roles and responsibilities of all participants involved in the business process properly. These models of orchestrated systems need to take into account the non-deterministic and often non-predictable behaviour of humans. Unfortunately, methods and tools that are currently available within software engineering are not sufficient for addressing these issues. Most formal approaches that have been developed so far in software engineering focus on technical components of a business process by modelling how software and technical components will react to triggers issued by a software component. However, even if humans are instructed to follow a particular protocol, they may or may not do so in practice.

1.2 Problem Statement

There should be a means to specify and analyse dynamic, static and stochastic aspects of human resource re-configurations in a business process. This specification can be used to test the benefits of the following resource allocation features: scheduling policies, assignment policies, load balancing and escalation handling. Scheduling policies provide a means for specifying the order that a task should be allocated to a resource, whereas assignment policies can be used to determine who should be assigned to a particular role. On the hand, load balancing can be used to reduce the businesses workload by globally reassigning work. Additionally, escalation handling can be used for specifying exception beyond typical assignment rules by triggering exceptions if a task is not progressing as expected.

> *Current approaches such as BPMN [47], WS-HumanTask [2] and InConcert [58] capture some aspects of human resource allocation; however they are not intended for business users to test protocols prior to employing them. In this thesis I propose a methodology and language to support the modelling and evaluation of business process with flexible assignment of human resources.*

The creation of a visual modelling language and methodology will encapsulates these requirements outlined in Chapter 2 to model and simulate a business process at different level of abstraction, by providing a means to define stochastic and flexible specifications. Generic performance questions can be defined based on the selection of model features. By formulating and testing performance questions one could validate which feature is beneficial for the business process. Questions 1-3 below illustrate generically defined performance questions.

1. Which scheduling method (by priority/deadline) increases the probability for a case to be completed on time? How does it affect the number of cases completed out of the total received?

2. Does the usage of assignment policies decrease the number of idle workers (not assigned to jobs) at any point in time?

3. Does escalation and/or load balancing increase the percentage of cases that are completed within a given deadline, or reduce the time that cases run past their deadline?

These questions can be used on any business process to detect the timeliness of the service and the efficiency of task allocation. Question 1 tests the alternative scheduling feature, such that it would require the comparison of a model that implements scheduling by priority compared to another model that is scheduled by deadlines. Question 2 tests the optional assignment policy feature, such that the comparison would test the absence and presence of the feature. Similarly, Question 3 tests the optional load balancing and escalation features, such that there are four unique models that could be tested comparing the absence and presence combination for both of these features.

1.3 Solutions

To solve the problem described in Section 1.1, this thesis introduces a visual modelling language which allows us to represent humans as part of flexible workflows using a rule-based approach. In order to permit dynamic non-deterministic decisions, a graph transformations rule-based approach is used, such that the configurations of a business process are represented using graphs and the changing operations to these configurations are defined using graph transformation rules.

To capture the dynamic nature of cases, states are used to describe distinct cases within the system at some instant of time [17] (see Section 7.1). Attribute values are used in some of the pre and post conditions of the transformation rules to define the current state of a particular case. The transformation rules capture the dynamic re-configuration of the system. At the semantic level, graphs are used to represent configurations and stochastic graph transformations are used to model state changes with non-deterministic timing, such as the execution of a business action with a known average delay or the assignment of an actor to a role. This allows us to model semi-structured processes, where actions are not chosen using fixed control flow, but are non-deterministically influenced by deadlines, priorities and escalation events.

At the same time the visual, rule-based approach provides an intuitive notation for structural changes, distinguishing between domain-specific and domain-independent rules. This enables the specification of generic allocation policies, defined using application conditions and constraints [31]. To manage the complexity of such models the support of a dedicated language with domain-specific notation and formal support for analysis is required. Since the operational semantics of stochastic graph transformation (GT) allow for simulation of workflows, a domain specific language (DSL) can be used in a stochastic graph transformation approach for simulating different features on workflows in dynamic commercial applications. The stochastic simulation [68] provides analysis capabilities for service-level metrics. For instance, it can be used to determine the probability of completing a task within a deadline or the degree in which late cases are past the deadline. It also enables us to compare the performance of different policies and the effectiveness of additional features such as load balancing and/or escalation handling.

This approach aims to effectively model human actors in business processes, integrating a flexible rule-based approach with a rigid control flow approach. The language syntax of the DSL is defined using a metamodel. The metamodel is influenced by Role-Based Access Control (RBAC) [57] models while the concrete notation extends that of UML class and use-case diagrams. The domain-specific language incorporates features of organisational modelling [46], such as techniques to assign activities to resources in organisational structures.

1.4 Contributions

The thesis proposes a new approach for **St**ochastic Modelling & **A**nalysis of **Dy**namic Human-Resource Allocation (StADy)[1]. This approach is outlined in a methodology which describes tasks and consistency checks required for developers to employ StADy for quantitative analysis of scheduling protocols, policies and regulations.

This thesis also proposes a new visual framework based on graph transformation for modeling business processes where human beings are involved. This is achieved by satisfying the requirements laid out in Chapter 2. The newly developed StADy language is a combination of two sub languages: configuration modelling language and transformation modelling language. The StADy configuration modelling language is a domain-specific language which consists of a metamodel, models and a graphical notation. The metamodel includes elements such as actor, role, process, artifact, state, case, capability and escalation. The models are extended use-case and class diagrams called StADy's hierarchy and artifact models, which contain additional information in terms of access control, escalation handling and capabilities. On the other hand, the StADy transformation modelling language is a graph transformation system which represents business process concepts and states using type and instance graphs. The StADy metamodel is the type graph for the language's graph transformation system. The instance graphs are presented in a new graphical notation, which can be used to denote things such as a person's availability or a case's escalation level. A catalogue of GT rules which

[1] The StADy approach is pronounced as stad-dee

capture the conceptual requirements (Section 2.1) is provided for future developers to employ. Guidelines for translating design models into simulation models were developed.

The thesis's core contributions include:

1. StADy methodology (Chapter 9)
    (a) Business modelling stage (Section 9.5)
    (b) Process execution design stage (Section 9.6)
    (c) Process encoding stage (Section 9.7)
    (d) Performance evaluation stage (Section 9.8)
2. StADy Language
    (a) StADy configuration modelling Language(Chapter 6)
        i. StADy metamodel (Section 6.1)
        ii. StADy's hierarchy and artifact models (Section 6.3)
        iii. StADy graphical notation (Section 6.4)
    (b) StADy transformation modelling language (Chapter 7)
        i. Dynamic (re)-assignment (Section 7.2.1)
        ii. Scheduling (Section 7.2.2)
        iii. Escalation handling (Section 7.3.1)
        iv. Role promotion/demotion and temporary promotion (Section 7.3.2)
        v. Load balancing (Section 7.3.3)
        vi. Human error & unpredictability (Section 7.3.4)
3. Translation of StADy design models into simulation models (Chapter 8)

1.5 Outline of the Thesis

The remainder of this thesis is organized as follows:

Chapter 2 defines the concept, methodology and language requirements for modelling and analysing business processes with human involvement.

Chapter 3 discusses background concepts for understanding the StADy Configuration Modelling Language such as workflow management and domain specific languages.

Chapter 4 introduces background concepts for understanding the StADy Transformation Modelling Language such as attributed graph transformation and stochastic simulation.

Chapter 5 defines a pharmacy case study which is used throughout the thesis for illustrations and methodology evaluation.

Chapter 6 uses metamodelling to present the StADy Configuration Modelling Language and demonstrates it with the pharmacy case study.

Chapter 7 defines the StADy Transformation Modelling Language.

Chapter 8 illustrates mechanisms for translating StADy designs into VIATRA2 implementations.

Chapter 9 defines the StADy methodology for modelling and analysising dynamic resource allocation.

Chapters 10–13 illustrate the stages of the methodology in detail using the pharmacy case study.

Chapter 14 illustrates an alternative execution of the methodology for testing effectiveness of load balancing and escalation handling.

Chapter 15 evaluates the StADy language and methodology.

Chapter 16 presents related work in comparison of the StADy approach and discusses why existing methods fail to satisfy the requirements.

Chapter 17 concludes the thesis with a discussion of several possibilities for extending this thesis in the future.

The diagram in Figure 1.1 shows major dependencies in the chapters of the thesis. An arrow from chapter 'A' to chapter 'B' means that understanding chapter 'B' depends on some important material in chapter 'A'. This diagram excludes the minor dependencies. The dots at the intersection of lines indicates merges. The conclusion does not have any major dependencies because it is self contained.

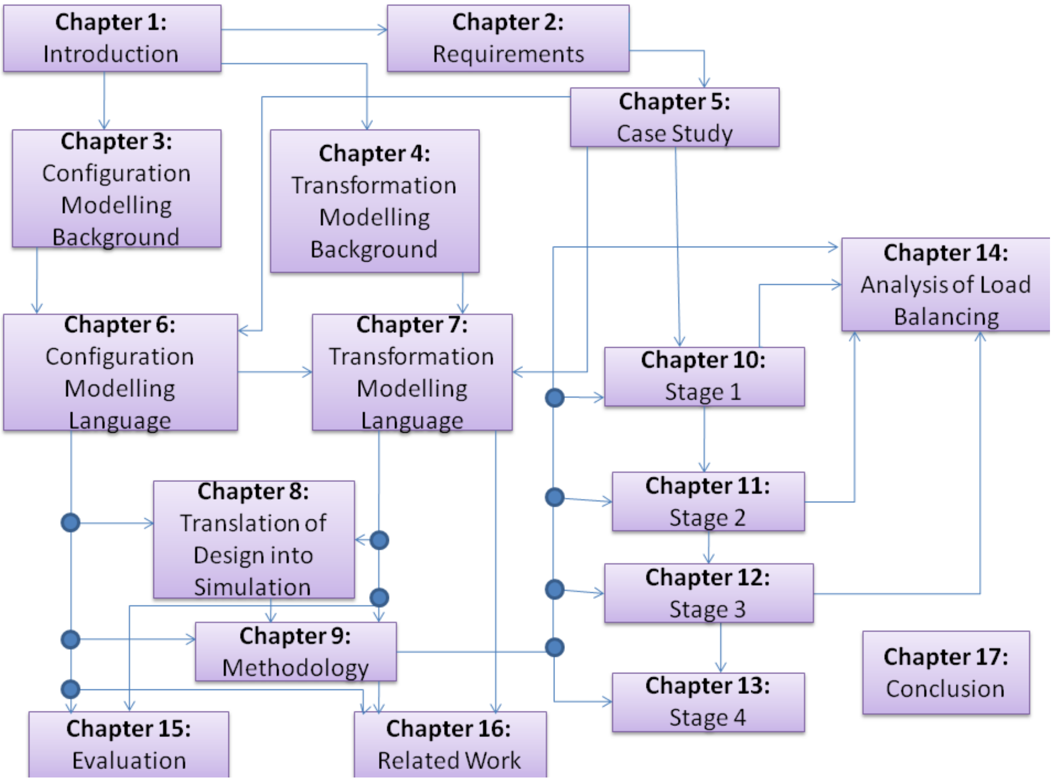


Figure 1.1: Chapter Dependencies

Chapter 2

Requirements

A suitable language for modelling human behaviour will need to account for a variety of requirements. These can be divided into three categories: concept, language and methodology requirements. Figure 2.1 illustrates the dependency relation between the categories [26]. Concepts are the basic semantic entities that are encoded into language syntax such as *cases* and *actors*. Language is a means for representing the concepts and supporting the methodology, whereas the methodology defines when and how to use the language. Each of the requirements is defined and briefly justified. Sections 2.1-2.3 outline the concept, language and methodology requirements, respectively. These requirements are referenced throughout the thesis using a (R#) reference index, such that the # is to their corresponding requirement number.

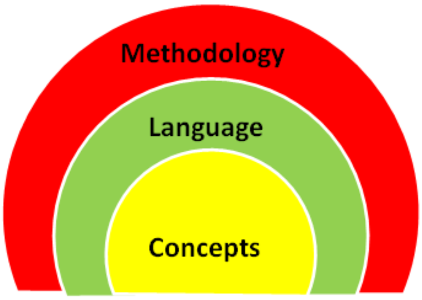


Figure 2.1: Requirement Structure

2.1 Concept Requirements

A business process (BP) is the production of a particular case (product/work-item). Each case involves a process being performed, which consists of a number of tasks that need to be carried out by a human or technical resource based on certain conditions. Resource allocation principles are based on the objective of completing a case as quickly as possible through the ordering of case selection and selection of a resource to perform the case [70]. Hence, the language concept requirements have been subdivided into two groups: features that help the selection of tasks to be performed on cases (task management) and features that help selection of people to perform the tasks (role management). The following concept requirements are composed of general properties that need to be captured in the language.

1. **ROLE MANAGEMENT**

   R1.1 Dynamic (re)-assignment

   - *Explanation:* Tasks can be dynamically (re)assigned to people based on their availability and capabilities. Optionally, assignment policies can be defined. In particular, predefined assignment protocols can be used to determine who should be assigned to a particular role. For example, a policy to assign the least-qualified person is illustrated as follows: if a bank cashier role for processing withdrawal transactions is required to be assigned and both a bank manager and a bank teller are available then the teller should take on that role. However, in a situation where no bank tellers available, the bank manager is flexible enough to take on the cashier role.

- *Justification:* Dynamic re-assignment reflects real world needs because in a typical workflow role assignment is constantly changing. This requirement is of particular interest because it directly relates to the problem being addressed in this thesis which is the ability to model dynamic human-resource allocation accurately.

R1.2 ***Role promotion, demotion and temporary promotion***

- *Explanation:* A promotion occurs once a person acquires capabilities sufficient to be assigned a new role. Demotion means that a person loses permission to take on a particular role. Temporary promotion occurs in response to an escalation for a fixed period of time. This requirement reflects real world needs because people's positions in organisations are constantly changing. For instance, a person can be promoted to a higher position if he or she advances in their field, whereas a person can be demoted if their performance is unsatisfactory.

On the other hand, temporary promotion is dependent on situations, such as the assistant manager temporarily covering duties for an ill manager.

- *Justification:* This requirement enables role allocation to dynamically change within a process. This requirement is beneficial to dynamic re-assignment (R1.1) because assignment is dependent on roles (role-based) and if roles are also dynamic then assignment will become more flexible.

R1.3 *Role-based access control*

- *Explanation:* Access control is a mechanism that grants or revokes the right to perform activities and/or access data. In role-based access control assignment permissions are associated with roles and users assigned to roles. Access control can be used for an authority to specify a person's allowable roles and access rights to artifacts.
- *Justification:* This requirement is needed because role-based access control can be used to specify which human resources are permitted to perform certain roles and what data the human resource can access while taking on a role. The use of role-based access control would strengthen dynamic re-assignment (R1.1), role promotion/demotion and temporary promotion (R1.2) specifications.

2. **TASK MANAGEMENT**

R2.1 ***Scheduling***

- *Explanation:* Scheduling is a means of allocating resources to tasks. The general structure of scheduling consists of a set of tasks with dependencies and resource requirements. The output of scheduling is a sequence of executions. Ordering tasks by deadline or priority are two common scheduling policies. Scheduling by deadline can minimise the amount of time it takes for tasks to be completed past their deadlines, whereas scheduling by priority could give preference to particular special cases.
- *Justification:* Scheduling can be used for human resources for assignment to tasks with the intention of optimising an organisation's criteria.

R2.2 ***Non-deterministic duration of tasks***

- *Explanation:* The duration to complete a task varies from person to person because it is influenced by various factors such as the individual's expertise, the difficulty of the task and external factors that go beyond the scope of the business process models. For instance, a highly trained worker is more likely to complete a task faster then an un-trained worker.
- *Justification:* In order to accurately emulate unpredictable task durations it is important for the system to take into account of the non-deterministic time it takes for human re-

sources to complete tasks.

R2.3 ***Temporal escalation handling***

- *Explanation:* Tasks can be modelled to express the expectation that the task is to be started or completed within a certain time frame. If a task is not progressing as expected then an escalation mechanism is required [34]. Product deadlines should trigger escalations enabling resources to be diverted to the escalated product. Procedures should be in place specifying how to handle escalations so that they comply with legal requirements and people are allowed to react efficiently.
- *Justification:* Escalation handling is a mechanism for achieving flexibility in human resource allocation because it can be used for specifying exceptions beyond typical assignment rules.

R2.4 ***Load balancing***

- *Explanation:* In a distributed setting, workflows can balance the load between different organisations, thus increasing flexibility, performance and reliability. Load balancing permits external involvement to help reduce the businesses workload.
- *Justification:* Tasks can be dynamically (re)assigned globally to other sites (locations). This strategy is dynamic reassignment (R1.1) at a higher level.

R2.5 ***Human error and unpredictability***

- *Explanation:* The human unpredictability feature should reflect that even if people are correctly instructed, they may or may not perform their allocated tasks. Backtracking may be required to eliminate human error.
- *Justification:* In order to capture human behaviour realistically in a business process negative aspects and their consequences need to considered.

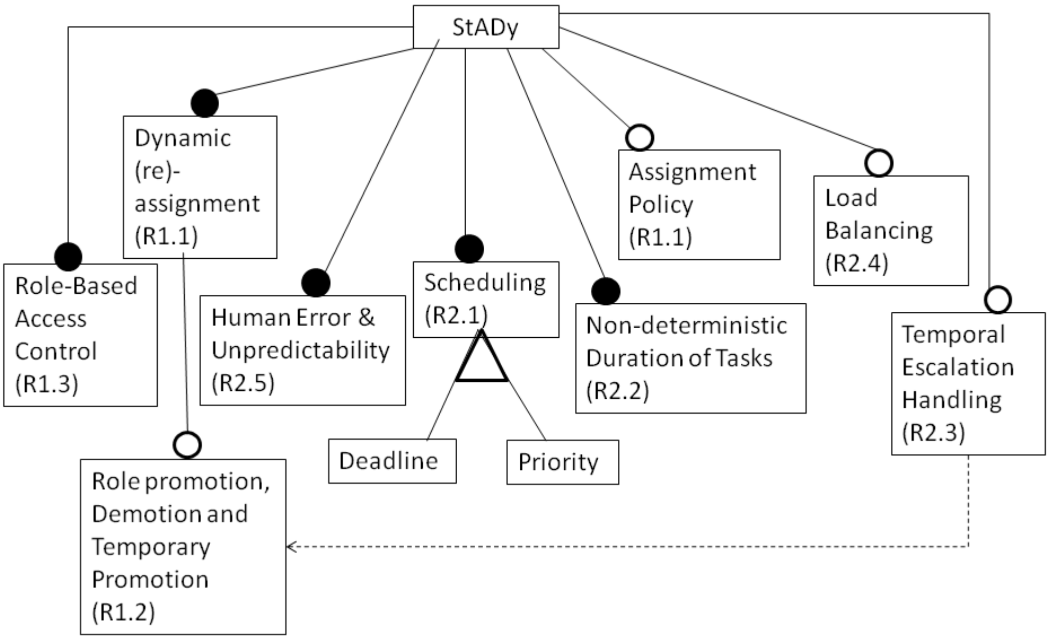

StADy
Dynamic (re)-assignment (R1.1)
Role-Based Access Control (R1.3)
Human Error & Unpredictability (R2.5)
Scheduling (R2.1)
Non-deterministic Duration of Tasks (R2.2)
Assignment Policy (R1.1)
Load Balancing (R2.4)
Temporal Escalation Handling (R2.3)
Deadline
Priority
Role promotion, Demotion and Temporary Promotion (R1.2)

Figure 2.2: Feature Diagram

A feature diagram is used to summarize the concept requirements as shown in Figure 2.2. A feature model [38, 75] provides a compact representation of features, whereas a feature diagram is a tree-like visual notation of a feature model. The relationships between the parent and child features are categorized into mandatory, optional, alternative features. The mandatory features, such as access control and human error, are denoted with a black circle. The optional features, such as assignment policy and escalation are denoted with a white circle. Sub-features deadline and priority have an alternative (xor) relationship under the mandatory scheduling feature. The dependency implication is denoted with the dashed arrow, which implies that if the source feature (i.e., escalation feature) is selected then the target feature (i.e., role promotion/demotion feature) must be selected as well. Some of the features are optional and alternative to enable the testing of distinct protocols such as the effectiveness of an assignment policy, by comparing simulation models with or without the feature. This feature diagram can be used to decide optional and alternative features to test as illustrated in Section 1.2. The concept requirements are further validated in Chapter 5.

2.2 Language Requirements

This section presents the language requirements for the development of a domain-specific business process language. The language should be composed of static, dynamic and stochastic aspects for modelling and specifying dynamic re-configuration of resources in business processes. The language requirements outline syntax and semantics.

3. **LANGUAGE**

R3.1 ***Visual representation***

- *Explanation:* Visual representation is the use of diagrammatic notation to represent the language. Business users can intuitively and visually understand diagrammatic notation with minimal amount of training.
- *Justification:* This is important because the models being developed are intended for business experts who need to design and validate strategies for resource allocation.

R3.2 ***Stochastic specification of task selection and duration***

- *Explanation:* Stochastic models allow to represent actual or apparent randomness of actions and their durations using probability distribution, thus supporting the modelling of unpredictability of humans (R2.5) and time (R2.2).
- *Justification:* Unpredictability of behaviour is a key feature of processes executed by humans and needs to be taken into account when evaluating different strategies.

R3.3 ***Flexible specification of unstructured aspects***

- *Explanation:*

  The language should permit dynamic non-deterministic decisions. The purpose of a flexible language is to reflect people's spontaneous choices by providing options without a fixed order i.e., a person has the option to choose the task to work on first. It is also in contrast to classical workflow modelling languages based on control-flow oriented or net-like diagrams.

- *Justification:* The control flow approach is suitable for defining structured business processes; however it is not suitable for unstructured business process aspects such as role assignment and scheduling, because in order to capture these non-deterministic decisions the control flow approach would have to break down into various complex exceptions. Therefore a flexible approach is necessary for specifying a unstructured business process. Since many processes include both structured and unstructured aspects, a control flow and a flexible approach is required.

The static aspect of the language requirements is defined in R3.1, because a visual representation provides a means to define configurations of a process. R3.3 represents the dynamic aspect of a business process by defining it in a flexible way. On the other hand, R3.2 is the stochastic aspect by providing a means for analysing quantitative properties.

2.3 Methodology Requirements

This section presents the methodology requirements which are procedures and techniques that should be employed.

4. **METHOD**

R4.1 ***Integrate with standard business modelling***

- *Explanation:* Integration means to use appropriate existing models and extend them where necessary or to provide mappings with standard notations. Integration to existing standardized modelling notations such as Unified Modeling Language (UML) will provide the ability to represent the model at different levels of abstraction.
- *Justification:* There is a potential to reuse existing models. The use of standard notions and methods would improve understandability and readability of the overall stochastic modelling and analysis of dynamic human-resource allocation approach.

R4.2 ***Simulation mechanism***

- *Explanation:* Simulation is the repeated execution of a system with the aid of a computer. The system is represented using a simulation model such as a stochastic simulation model. Stochastic simulation models use random variables as inputs and outputs estimations of the true characteristics of a system [55]. Also simulations provide a flexible analysis technique for testing protocols prior to employing them in day to day routine.

- *Justification:* Simulation is a means for analysing resource allocation protocols on a business process. With the goal of increasing business productivity in mind, simulation can help in the selection of a resource allocation protocol.

2.4 Summary

This chapter introduced concept, language and methodology requirements for modelling business processes involving humans. These requirements will be referenced throughout the thesis using a (R#) reference index, such that the # is to their corresponding requirement number. The next two chapters (Chapters 3-4) will introduce relevant background information to help with the understanding of the StADy approach.

Chapter 3

Organisational & Process Modelling

In order to understand the Configuration Modelling Language, definitions of organisational and process modelling concepts are necessary. Basic notions of workflow management, role-based access control (RBAC), domain specific language (DSL) and unified modelling language (UML) are introduced. Each section specifies the relevance of the corresponding background information. This chapter concludes with a summary section that summarizes the background and locates where particular concepts are used in the thesis.

3.1 Workflow Management

Background information on workflow management is relevant for understanding the configuration modelling language in Chapter 6. A *business process (BP)* is defined as the production of a particular *case (work-item/product/*

service). Each *case* involves a *process* being performed, which consists of a number of *tasks* that need to be carried out by a *resource* based on conditions. A process can be subdivided into three categories [70]: primary (production), secondary (support), and tertiary (managerial). The primary process is customer-oriented by focusing on the case. The secondary process supports the primary process by maintaining the production. The managerial process directs and coordinates the primary and secondary processes in terms of required resources and process allocation. This classification is used in Chapter 7.

A resource is defined as "the generic name for a person, machine or group of persons or machines that can perform specific tasks" [70]. The performance of a task by a resource is called an *activity*. Some tasks can be performed without human involvement, whereas others require human involvement. Unlike machines when a person is allocated a task they have the choice to accept it. A *role* (function/qualification) is a group of resources with complementary skills [70].

Resource allocation is very important to the efficiency and effectiveness of a workflow [70]. Therefore, allocation principles are based on the objective of completing a case as quickly as possible, through the ordering of cases and selection of the resources performing the cases. Some of the common queuing disciplines that can be used for ordering of cases are [70]: *First-in First-out (FIFO)*, *Last-In First-Out (LIFO)*, *Shortest Processing Time (SPT)*, *Shortest Rest-Processing Time (SRPT)*, and *Earliest Due Date (EDD).* FIFO is an allocation rule in which cases are allocated in the order they arrived, whereas LIFO is the opposite ordering. Queuing is influenced by temporal aspects in SPT, SRPT and EDD methods in terms of processing time and required completion times. The order that a case should be performed is closely associated with the selection of the resource because if the task on the case can be carried out by more than one resource then there should be considerations made for the selection [70]. The options for allocating a resource to a case include assignment to: a specialized resource, resources that recently fulfilled similar tasks or the least qualified person [70]. If the least qualified permitted person is selected, this can benefit the workflow by leaving the higher qualified person available to perform other specialized duties in the future [70]. Such choices must be continually made during allocation of cases to resources which is further illustrated in Section 7.2.1. It is also important to note that, even if the workflow engine takes on an advisory role in formulating allocation decisions, humans still retain the freedom to deviate from these suggestions.

3.2 Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) is a predominant model for advanced access control, which was formalized in 1992 by David Ferraiolo and Rick Kuhn [23]. It is an access control mechanism that promotes central administration of an organisational specific security policy. RBAC policy bases decisions on the functions a user is allowed to perform within an organisation [23]. By 1994, various IT vendors such as IBM and Siemens began developing products based on the RBAC model. In 1998 Ravi Sandhu introduced a family of models called RBAC96, which encompasses the relationship between four models: RBAC$_0$, RBAC$_1$, RBAC$_2$ and RBAC$_3$ [57]. In 2000 David Ferraiolo, Rick Kuhn and Ravi Sandhu unified the RBAC models to create NIST RBAC [56] and later developed an ANSI/INCITS standard in 2004.

The StADy language metamodel (Section 6.1) is influenced by Sandhu's RBAC$_3$ model, because the central notion of RBAC96 is that assignment permissions are associated with roles and users assigned to roles. Roles represent competency, authority or responsibility to do a specific task, which formulates the access control policy [57]. The RBAC$_3$ model consists of sessions/objects (S) in addition to a combination of RBAC$_0$, RBAC$_1$ and RBAC$_2$ as shown in Figure 3.1.

The base model (RBAC$_0$) specifies the minimum requirements for a system to fully support RBAC [57]. These minimum requirements include three sets of entities called users (U), roles (R), and permissions (P). RBAC$_1$ and RBAC$_2$ are both extensions of RBAC$_0$ with distinct additional features. RBAC$_1$ adds role hierarchy (RH) feature whereas RBAC$_2$ adds constraints to the model. A user is defined as a human-being. A role is a job function or job title within the organization with additional semantics regarding authority and responsibility. Permission is defined as approval of a particular mode of access to one or more objects in the system. Figure 3.1 shows the *many-to-many* relationship between user assignment (UA) and permission assignment (PA). Hence, a user can be a member of many roles and a role can have many users. Similarly, a role can have many permissions and the same permission can be assigned to many roles. Users are not directly connected to permissions because it weakens access control; however the placement of role

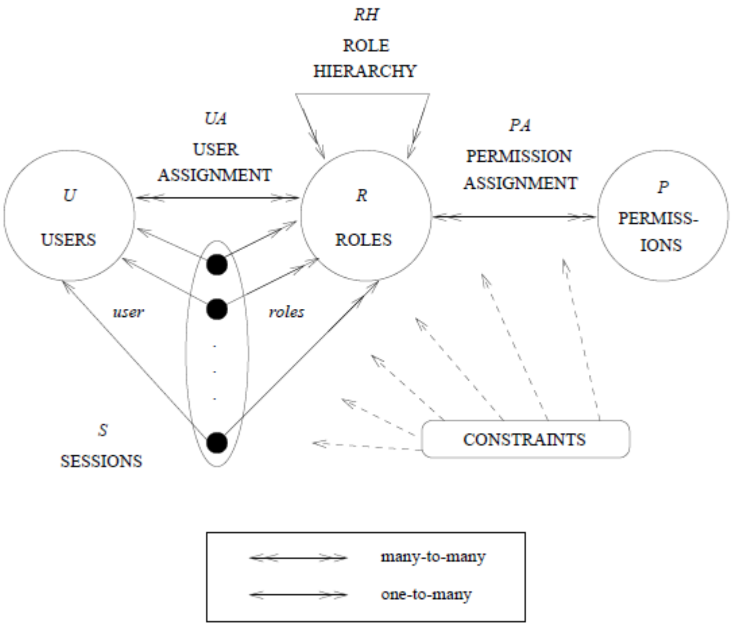


Figure 3.1: RBAC$_3$ [57]

between user and permission provides a greater control over access configuration [57]. A subject (or *session*) is defined as a unit of access control, and a user may have multiple subjects active at the same time. *Constraints* is a mechanism used for enforcing "acceptable" or "unacceptable" organisational policies. The StADy metamodel relation to Sandhu's RBAC$_3$ model will be further discussed in Section 6.1.

3.3 Domain-Specific Language (DSL)

This section is relevant for understanding the creation of the StADy Configuration Modelling Domain Specific Language (DSL), which is introduced in Chapter 6. A *DSL* is composed of various concepts, which are outlined in Figure 3.2, such as the domain and the metamodel. The *domain* is the bounded field of interest or knowledge, for example the pharmacy business process to be discussed in Chapter 5. The metamodel is composed of the abstract syntax, the concrete syntax and the static semantics of the language which is illustrated in Sections 6.2 and 6.3, respectively. The abstract syntax denotes the structure and grammatical rules of a language, whereas the concrete syntax denotes the graphical notation and the language representation [39]. The static semantics of a language is used to validate well-formedness. A *DSL* allows key aspects of a domain to be formally expressed and modelled by using syntax and semantic constructs [74].

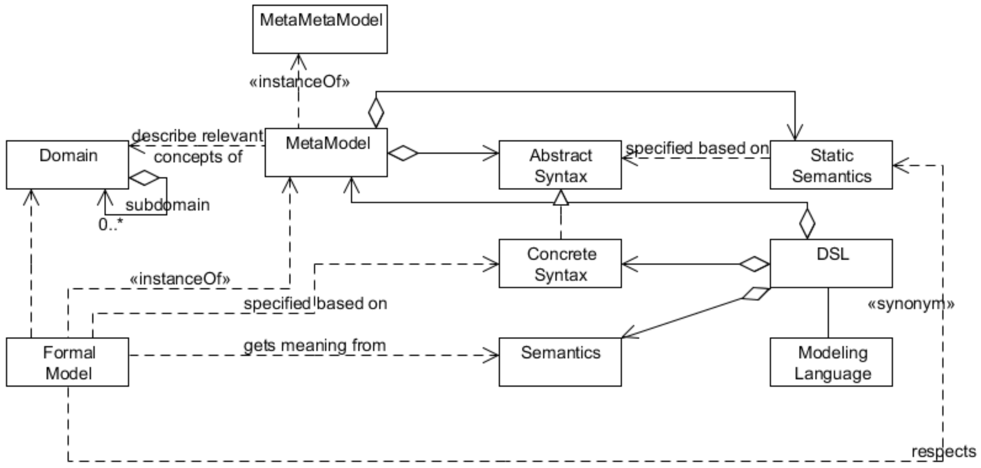


Figure 3.2: DSL Modelling Concepts [74]

Metamodelling is defined as "the act and science of creating metamodels, which are qualified variants of models" [28]. A *metamodel* is a model that defines the components of a conceptual model, process, or system. There are two types of metamodels: ontological and linguistic. The ontological (or logical) metamodel follows the Object Management Group (OMG) strict metamodelling hierarchy technique, which enforces the rule that the "instance-of" relationship between instance (object) and type is only permissible between a neighbouring pair of layers. The linguistic (or physical) metamodel focuses on the alignment of the metalevels with the notion of abstraction, using the "instance-of" relationship across and within levels [28]. Figures 3.3 and 3.4 outline the difference between these metamodeling techniques. A linguistic metamodel is used to define the StADy configuration modelling language in Section 6.1.

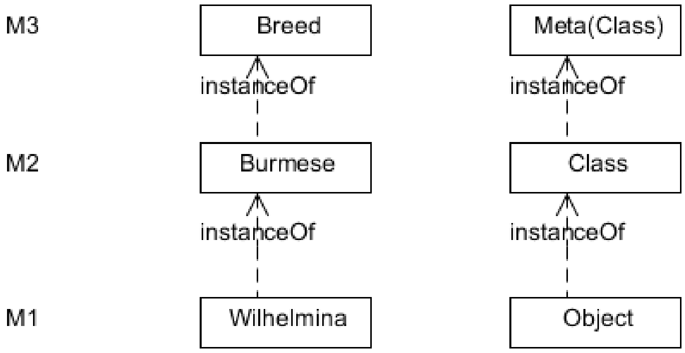


Figure 3.3: Logical (Ontological) Metamodel [28]

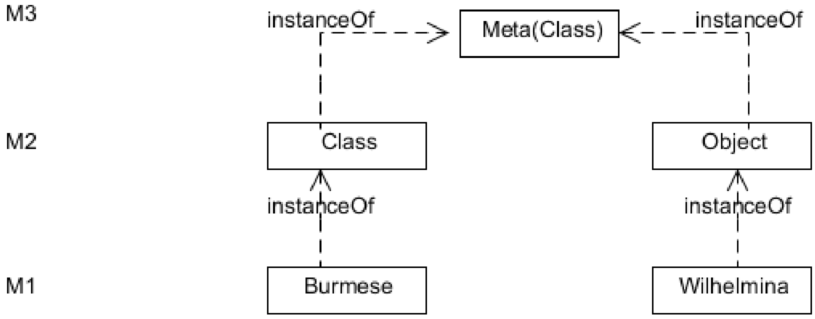


Figure 3.4: Physical (Linguistic) Metamodel [28]

3.4 Unified Modeling Language (UML)

The Unified Modeling Language (UML) section is important because it is used in the first stage of the methodology (Chapter 9) and some of the diagrams are extended by the StADy configuration modelling language in Section 6.3.

The common meta-language of the *Meta-Object Facility (MOF)* [66] defines all the foundational concepts required to build the *Unified Modelling Language (UML)* [67], hence model elements in UML are instantiated from model elements defined in MOF. Figure 3.5 provides a visual representation of the relationship between the UML and MOF metamodels.

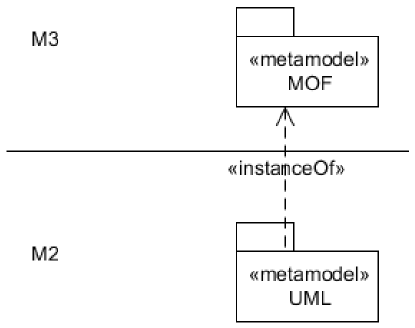


Figure 3.5: Relationship Between UML and MOF

UML is a graphical modeling language that defines a notation and a meta-model. The UML meta-model defines the concepts of the language through its abstract syntax. UML is a standard object modeling language controlled by the Object Management Group (OMG). The OMG forms standards that support interpretability in object-oriented systems [24]. UML uses various graphical notations to create models for object-oriented analysis and design methods [73]. UML diagrams are used for either static or dynamic modeling. *Dynamic model*s describe the behavior of a system, such as use-case and activity diagrams, whereas *static model*s describe the structural aspect of a business process, such as class and object diagrams.

The UML diagrams relevant to this thesis are the following: class, object, use-case and activity diagrams. Figure 3.6 shows the relevant diagrams as a subset of the classification of UML diagram types. Class, use-case and activity diagrams are illustrated in Section 5.2.

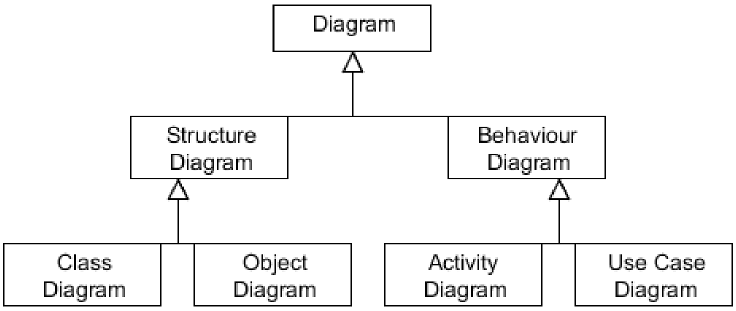


Figure 3.6: Relevant UML Diagrams

A *class diagram* describes types of objects in a system and their static relationships [24]. The metamodel of a class diagram defines it to be composed of properties and generalizations. The *properties* represent the structural features of a class and are shown as the attributes and associations in the diagram. *Generalization* defines inheritance between classes in a diagram. Figure 3.7 is an example of a simple class diagram.

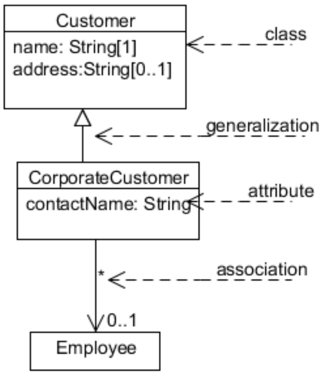


Figure 3.7: Basic Class Diagram

Class diagrams contain associations which are relationships between two classes, the specialized forms are called Aggregation and Composition association [24]. They are distinguished by class parent–child life cycle and ownership relation. Aggregation is defined as the *"part-of"* relationship, where all child classes are owned by the parent class (i.e. the child class that cannot belong to another parent class); however each child class has its own life cycle. On the other hand, composition is defined as a *"has-a"* relationship and is a strong type of aggregation, where the child class does not have its own life cycle; hence, if a parent class terminates then all its child classes will also be terminated.

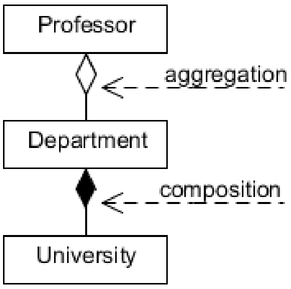


Figure 3.8: Association: Aggregation vs Composition

Figure 3.8 illustrates the notation for aggregation and composition associations. The example *department–professor* aggregation relationship states that a single *professor* cannot belong to multiple *departments*. Due to the aggregation nature, if the *department* class is terminated, then the connecting *professor* classes will not also terminate. On the other hand, the *university–department* composition relationship states that the *university* can contain multiple departments and a *department* object can only belong to one university. Due to the composition nature if the *university* class is terminated, then all its connecting *departments* will automatically delete.

Object diagrams denote snapshots of objects in a system at a point of time [24]. The elements' names are underlined and are usually in the following form: <u>*instance name: class name*</u>. Figure 3.9 illustrates an instance (object diagram) of the simple class diagram defined in Figure 3.7.

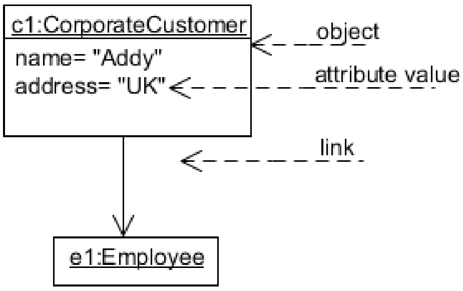


Figure 3.9: Object Diagram

Use-Case Diagrams are composed of the following six modelling elements: actors, use-cases, association, generalization, *"includes"* relationships and *"extends"* relationships. An *actor* is a role that a user, external to the system, plays in relation to the system [48]. The actor's description may be refined using *generalization*, as used in class diagrams. Figure 3.10 displays the basic graphical syntax of a use-case diagram. Role hierarchies can be represented in use-case diagrams. Use-case descriptions are used to specify the use-case content, hence each use-case in the diagram is described using the templates defined in Table 3.1 to 3.3.

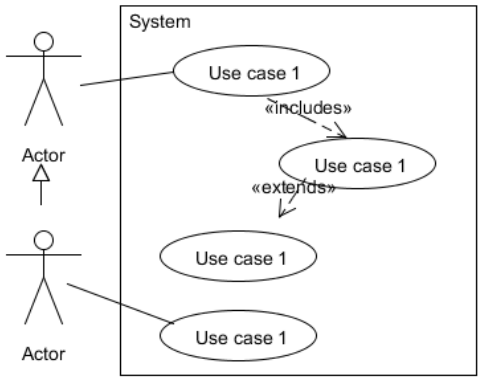


Figure 3.10: Use-Case Diagram's Graphical Syntax

Name:	
Primary Actor:	
Goal (of the User):	
Precondition:	
Postcondition (successful execution):	
Trigger Event:	
System (implementing it):	
Participating Actor:	

Table 3.1: Case Details

Step	**User**	**Action**
1		
2		
..		

Table 3.2: Case Scenario

Step	**Condition for alternative**	**Alternative Action**
	..	..
	..	..

Table 3.3: Case Additional Exceptions

Activity Diagrams are used to describe procedural logic, business process and work flow [24]. Partitions or *swim lanes* may be used to show who carries out particular actions. Also coherent object flow can be defined by using *pins* at the edge of each *activity* node with labels stating the type of input and output object. Also activities can be decomposed into sub activity diagrams which can be denoted by using a *rake* symbol inside the corresponding activity. Figure 3.11 below displays some of the graphical syntax of an activity diagram [24].

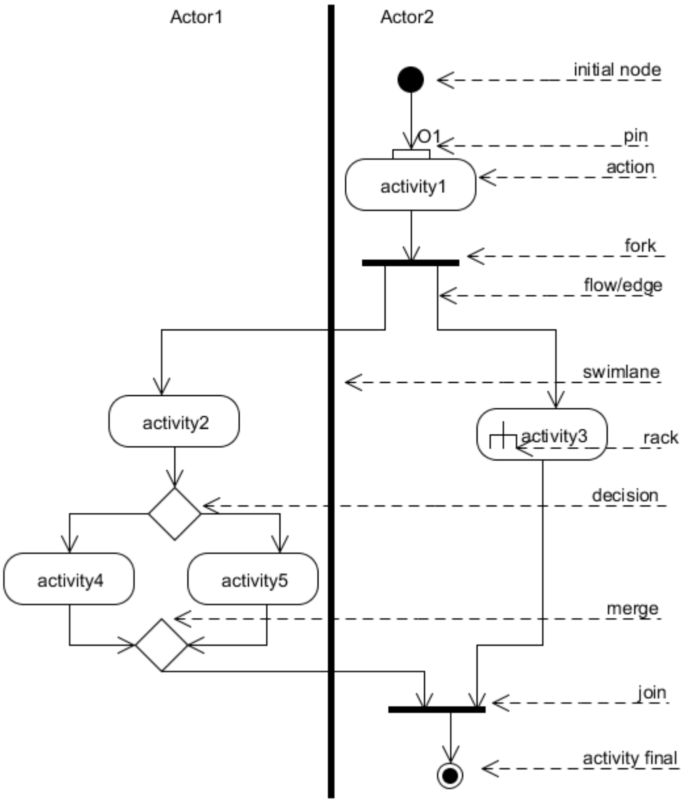


Figure 3.11: Activity Diagram's Graphical Syntax

3.5 Summary

In this chapter, definitions for the relevant concepts of workflow management, Role-Based Access Control (RBAC), Domain Specific Language (DSL) and Unified Modelling Language (UML) were introduced.

The workflow management section presents relevant terms, principles, and classifications that are used for the development of the modelling language in Chapter 6. Some of the definitions introduced included: business process, case, role and resource. These terms are used as elements in the StADy configuration modelling language metamodel in Section 6.1; however 'actor' is used instead of 'resource' to place emphasis on human resources in the model. Some of the important principles that were introduced consisted of assignment options and common queuing disciplines for resource allocation. The assignment options are further illustrated and discussed in Section 7.2.1 in terms of assignment policies, whereas EDD queuing discipline is illustrated in Section 7.2.2 for scheduling cases by deadline. This section also describes how a business process can be subdivided into production, support, and managerial levels. This three-level classification is used to subdivide the StADy transformation modelling language's GT rules, which is further discussed in Section 7.1 and illustrated in Sections 7.2 and 7.3.

The RBAC section discusses how to model access control between users, roles and objects, which is used in the development of the StADy language's metamodel. Section 6.1 describes the mapping of RBAC structures and discusses the association of individuals to permissions via roles to permissions, which forms the basis of the metamodel (Figure 6.2).

The DSL section discussed a metamodel approach for developing domain models. This section is relevant for understanding the creation of the newly developed StADy DSL which will be introduced in Chapter 6. Important metamodel distinguishing features are made to ensure that a clear understanding of linguistic metamodelling is provided, in order to understand the defined linguistic metamodel in Section 6.1. This section also differentiates between abstract and concrete syntax, which is further illustrated in Sections 6.2 and 6.3 respectively. The pharmacy business process (Chapter 5) domain is illustrated in the configuration modelling language (Chapter 6).

The UML section shows the basic structure of existing standard modelling techniques. UML is revisited in Chapter 5 to visually represent a pharmacy business process (Section 5.2). UML class, use-case and activity diagrams are also used in the first stage of the methodology as described in Chapter 9.5. UML class and use-case diagrams are extended by the StADy configuration modelling language, which is described in stage two of the methodology (Chapter 9.6) and illustrated in Section 6.3. The next chapter will present background information required for understanding the StADy transformation modelling language.

Chapter 4

Graph Transformation & Stochastic Simulation

In order to understand the StADy Transformation Modelling Language, definitions of the relevant concepts are necessary. Basic notions of graph transformation (GT), advanced notions of typed attributed graph transformation (TAGT), stochastic graph transformation systems (SGTS) and stochastic simulation are introduced. The introduction of each section specifies the relevance of the corresponding background information. This chapter concludes with a summary section that summarizes the background and locates where particular concepts are used in the thesis.

4.1 Graph Transformations (GT)

Graph transformations (GT) [21] are used in the approach to permit dynamic non-deterministic decisions, as described in Section 7.1 and illustrated in

Sections 7.2 and 7.3.

GT is a rule-based approach which represents procedural knowledge in terms of a set of “IF-THEN” *rules.* These rules define preconditions and effects of the basic activities. This form of modelling is in contrast to the *control-flow* approach, which focusses on the expected ordering of events. The rule-based models are more suitable for flexible processes because of the inherent non-determinism in selecting a rule. The following formal definitions appear in [21].

Definition 4.1.1. ***Graph (G)***
A graph is a tuple $G = (V, E, s, t)$ *where* V *is a set of nodes (vertices),* E *is a set of edges and* $s, t : E \to V$ *associate, respectively, a source and target node for each edge in* E.

Given graphs G_1, G_2 *with* $G_i = (V_i, E_i, s_i, t_i)$ *for i= 1,2, a graph morphism* $f : G_1 \to G_2$, *f=*(f_V, f_E) *consists of two functions* $f_V : V_1 \to V_2$ *and* $f_E : E_1 \to E_2$ *that preserve the source and target functions, i.e.* $f_V \circ s_1 = s_2 \circ f_E$ *and* $f_V \circ t_1 = t_2 \circ f_E$.

Definition 4.1.2. ***Type Graph (TG) and Instance Graph (G)***
A type graph is a tuple $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$ *where* V_{TG} *is a vertex type alphabet and* E_{TG} *is a edge type alphabet. A TG-typed instance graph consists of a graph* G *and a typing morphism* $type : G \to TG$.

Figure 4.1 illustrates a type graph, whereas Figure 4.2 is an instance graph of the type graph. The type graph defines a basic organisational structure consisting of people (P), their role (R) assignments and cases (C). The instance graph is the illustration of two people (P_1,P_2), P_2 assigned to a role for a particular case whereas P_1 is free. The set of all sequences of consecutive transformation steps using rules denotes a system's behaviour.

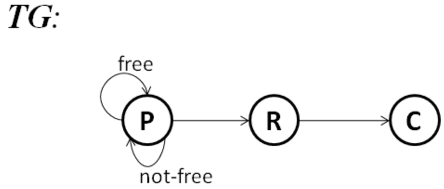


Figure 4.1: Type Graph

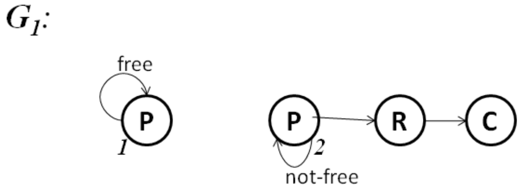


Figure 4.2: Instance Graph

Definition 4.1.3. ***Graph Transformation Rule (Production)***
A typed graph rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ consists of three typed graphs the left-hand side L, the gluing graph K and the right hand side R and two injective (typed) graph morphisms l and r. The left-hand side represents the precondition of the rule, while the right-hand side represents the postcondition of the rule.

Figure 4.3 illustrates a graph transformation rule which corresponds to the type graph defined in Figure 4.1. This GT rule is for the assignment of a person to role for a particular case.

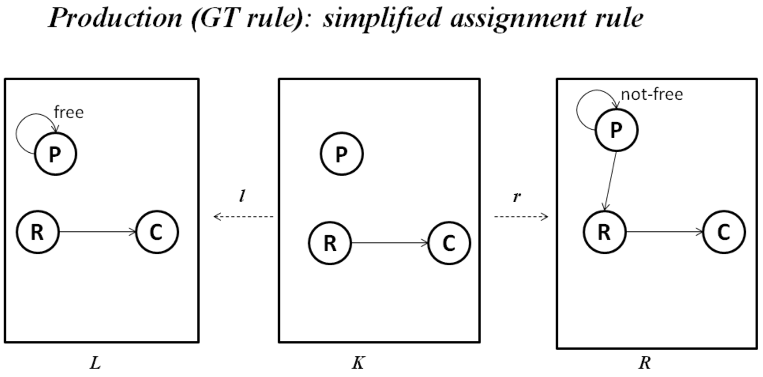


Figure 4.3: Production (GT Rule)

Definition 4.1.4. ***Graph Transformation (GT)***

Assume a (typed) graph rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *and a typed graph* G *with a match* m*, which is a (typed) graph morphism* $m : L \rightarrow G$*. A direct (typed) graph transformation* $G \overset{p,m}{\Rightarrow} H$ *from typed graph* G *to* H *is given by the double pushout diagram in Figure 4.8.*

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "n"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

Figure 4.4: Double Pushout (dpo)

A (typed) graph transformation is a sequence of direct (typed) graph transformations $G_0 \overset{*}{\Rightarrow} G_n$.

Definition 4.1.5. ***Graph Transformation System (GTS)***
A typed graph transformation system $GTS = (TG, P, \pi)$ consists of a type graph TG, a set of rule names P and a function π associated with each name $p \in P$ a span of injective TG-typed graph morphisms $\pi(p) = (L \xleftarrow{l} K \xrightarrow{r} R)$ [68].

Advanced concepts such as attributes, constraints and application conditions are introduced in the next section.

4.2 Typed Attributed Graph Transformation (TAGT)

In order to use graph transformation systems to specify and implement visual modeling techniques such as UML and StADy notation (Section 6.4), typed attributed graph transformation systems are used. The concepts and definitions are extended from graph transformation (Section 4.1), by allowing node and edge attribution. The following formal definitions appear in [21].

Attributed graphs use a different kind of graph called *E-graph* which extend the graph defined in Definition 4.1.1. An *E-graph* is a graph equipped with an additional set of data nodes (or values) and special sets of edge attributes and node attributes connecting, respectively, edges and nodes to values.

Definition 4.2.1. ***E-Graph (EG)***
An E-Graph $EG = (V_G, V_D, E_G, E_{NA}, E_{EA}(source_j, target_j)_{j \in \{G,NA,EA\}}))$ consists of a set of graph nodes V_G, a set of data nodes (vertices) V_D, a set of

graph edges E_G, a set of node attribute edges E_{NA}, set of edge attribute edges E_{EA}, a source function for graph edges $source_G : E_G \rightarrow V_G$, a target function for graph edges $target_G : E_G \rightarrow V_G$, a source function for node attribute edges $source_{NA} : E_{NA} \rightarrow V_G$, a target function for node attribute edges $target_{NA} : E_{NA} \rightarrow V_D$, a source function for edge attribute edges $source_{EA} : E_{EA} \rightarrow V_G$ and a target function for edge attribute edges $target_{EA} : E_{EA} \rightarrow V_D$.

Consider the E-graphs G^1 and G^2 with $G^k = (V_G^k, V_D^k, E_G^k, E_{NA}^k, E_{EA}^k, (source_j^k, target_j^k)_{j\in\{G,NA,EA\}})$ for k=1,2. An E-graph morphism $f : G^1 \rightarrow G^2$ is a tuple $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}})$ with $f_{V_i} : V_i^1 \rightarrow V_i^2$ and $f_{E_j} : E_j^1 \rightarrow E_j^2$ for $i \in \{G, D\}, j \in \{G, NA, EA\}$ such that f commutes with all source and target functions.

Definition 4.2.2. ***Attributed Graph (AG)***

An attributed graph is a tuple $AG = (EG, D)$ where EG is an E-graph and D is an algebra with data signature $DSIG = (S, OP)$ such that $\biguplus_{s\in S} D_s = V_D$. Intuitively, an attributed graph is an E-graph where V_D is the set of all data values available for attribution.

An attributed graph morphisms $f : (EG, D) \rightarrow (EG', D')$ of attributed graphs is a pair of an E-graph morphism $f_{EG} : EG \rightarrow EG'$ and a compatible algebra homomorphism $f_D : D \rightarrow D'$.

Definition 4.2.3. ***Attributed Type (ATG) and Instance Graph (EG)***

An attributed type graph is a an attributed graph $ATG = (TG, Z)$, where Z is the final DSIG-algebra. A typed attributed graph is a tuple (AG, t) with attributed graph AG with a graph morphism $t : AG \rightarrow ATG$.

Typed attributed graphs over an attributed type graph ATG form the category $\mathbf{AGraph_{ATG}}$ *of ATG-typed attributed graphs.*

A type graph models the conceptual structure and provides types for the instance graphs; hence well-formedness of an instance graph is determined by checking whether it conforms to its type graph. UML class-diagram and object-diagram notation (Section 3.4) are used to represent the type and instance graphs, as illustrated in Figures 4.5 and 4.6, respectively. These figures also present a mapping of *E-graph* elements. The type graph defines a basic organisational structure consisting of people, their corresponding capabilities, and their role assignments. The instance graph is the illustration of a person by the name of Bob with cashier capabilities who is assigned to perform cashier duties on c1 case.

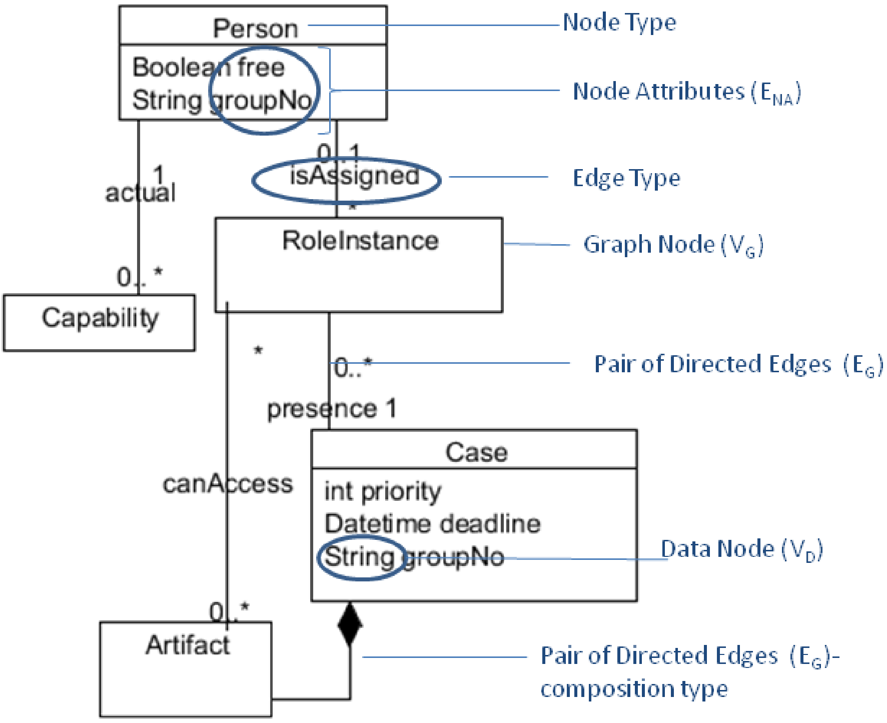


Figure 4.5: Type Graph

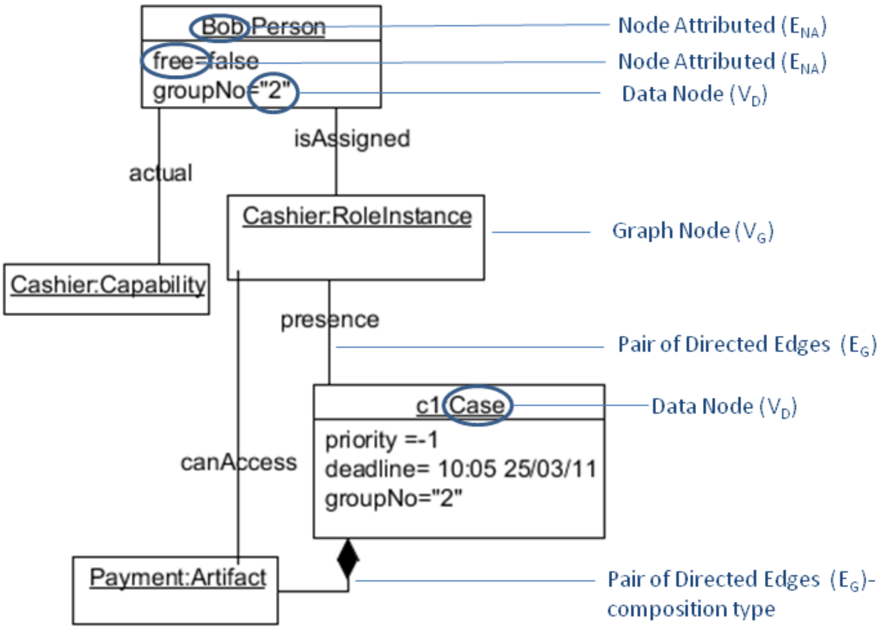


Figure 4.6: Instance Graph

Definition 4.2.4. ***Typed Attributed Graph Transformation Rule***
A typed attributed graph transformation rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ consists of three typed attributed graphs with a common DSIG-algebra $T_{DSIG}(X)$, the left-hand side L, the gluing graph K and the right hand side R, and two injective (typed) attributed graph morphisms l and r that are identities on the data algebra. Rule(ATG,X) is the set of all rules over an attributed type graph with variables in X.

Figure 4.7 provides a basic example of a graph transformation rule for the type graph (TG) defined in Figure 4.5. This example illustrates the assignment of an available person to perform a role on a case. Both the person and the case are located at the same pharmacy (group). Formally, the GT rule ($\pi(p)$) in Figure 4.7 is a span of injective TG-typed morphism

$L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$, but visually only L and R are shown. K is the intersection between L and R.

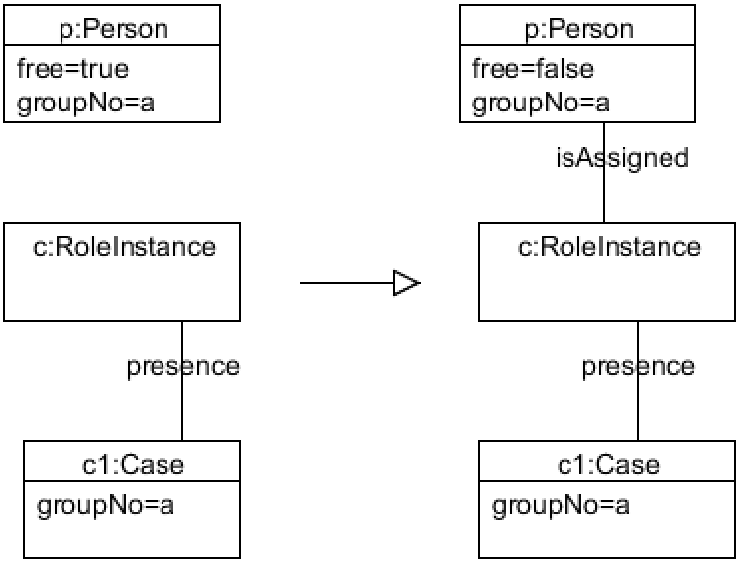


Figure 4.7: GT Rule

Definition 4.2.5. ***Typed Attributed Graph Transformation***
Assume a typed attributed graph rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *and a typed attributed graph* G *with a match* m*, which is a (typed) graph morphism* $m : L \rightarrow G$*. A direct typed attributed graph transformation* $G \overset{p,m}{\Rightarrow} H$ *from typed attributed graph* G *to* H *is given by the double pushout diagram in Figure 4.8 in* $\mathbf{AGraph_{ATG}}$.

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "n"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

Figure 4.8: Double Pushout (dpo)

A (typed) attributed graph transformation is a sequence of direct (typed) attributed graph transformations $G_0 \overset{*}{\Rightarrow} G_n$.

In addition to the basic graph transformation functionality there are various additional advanced concepts such as constraints and negative application conditions that are illustrated in Chapter 7. A *graphical constraint* can be used to define permitted or forbidden subgraphs. They are used to formulate the condition that a graph G must (or must not) contain a certain subgraph G', without being connected to a GT rule. On the other hand, a *negative application condition (NAC)* is defined within GT rules to forbid the occurrence of a graph structure in the pattern matching of a rule, by restricting the application of productions.

Definition 4.2.6. ***Graph Constraint for*** $\mathbf{AGraph_{ATG}}$

An atomic graph constraint is a tuple (a,A) where $a : P \to C$ *is a morphism attributed over* $T_{DSIG}(X)$ *and* $A \subseteq \{t_1 \# t_2 | t_1, t_2 \in T_{DSIG}(X), \# \in \{=, \neq, <, \nless, \leqq, \nleqq\}\}$ *as shown in Figure 4.9.*

A morphism $p : P \to G$ *satisfies (a,A) if there exists q:* $C \to G$ *such that* $q \circ a = p$ *and* $q(t_1) \# q(t_2)$ *for all* $t_1, t_2 \in A$ *where q(t) is the evaluation* *of term t in G's data algebra. G satisfies (a,A) if for all* $P \overset{a}{\rightarrow} G$*, p satisfies (a,A).*

A boolean constraint c is a boolean expression over atomic constraints based on the usual operations $\wedge, \vee, \neq, \rightarrow$*, true and false. Satisfaction of boolean constraints is defined as usual based on the satisfaction of atomic constraints.*

$$\begin{tikzcd}
P \arrow[rr, "a"] \arrow[dr, "p"'] & \arrow[d, phantom, "="] & C \arrow[dl, "q"] \\
& G &
\end{tikzcd}$$

Figure 4.9: Graph Constraint Structure

A positive graphical constraint is denoted with a instance graph as shown in Figure 4.10, whereas a negative graphical constraint is denoted with a diagonal line across the instance graph. This example illustrates the constraint that if a person is assigned to a *RoleInstance* then the person must have the capability to perform the role.

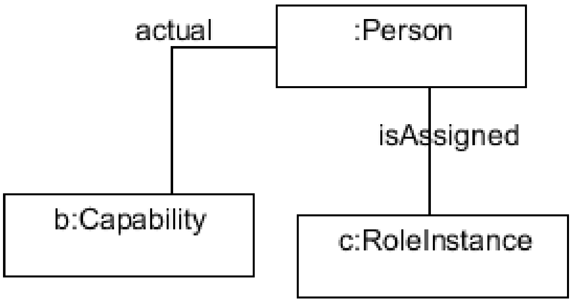


Figure 4.10: Graphical Constraint

Definition 4.2.7. ***Negative Application Condition (NAC)*** *A negative application condition (NAC) for attributed graph rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is a set of boolean constraints N of the form $\neg(x, A)$ where $x : L \rightarrow X$. This is visually represented in Figure 4.11. Satisfaction of boolean constraints $\neg(x, A)$ is defined as in Definition 4.2.6 where p is replaced by the match m of the rule. A match $m : L \rightarrow G$ satisfies N if m satisfies all $\neg(x, A) \in N$. A rule p with application condition N is denoted $\hat{p} = (P, N)$*

$$\begin{tikzcd} L \arrow[rr, "x"] \arrow[dr, "m"'] & {=} & X \arrow[dl, "p", "|" marking] \\ & G & \end{tikzcd}$$

Figure 4.11: NAC Structure

Negative application condition (NAC) is visually denoted with a diagonal line across the structure within the left-hand side of the rule as shown in Figure 4.12. This example illustrates that a new payment artifact can be added to the case if there exists no payment artifact in 'c1:case'.

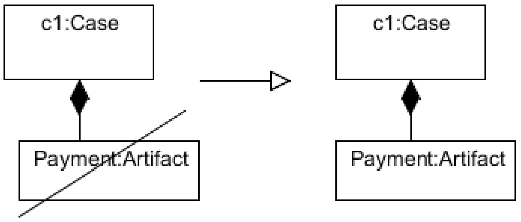


Figure 4.12: Negative Application Condition (NAC)

Figure 4.14 illustrates a condensed way of denoting a GT rule with *NAC* (Definition 4.2.7), which is equivalent to Figure 4.13. The notation used in these two figures will be further discussed in Section 6.4.

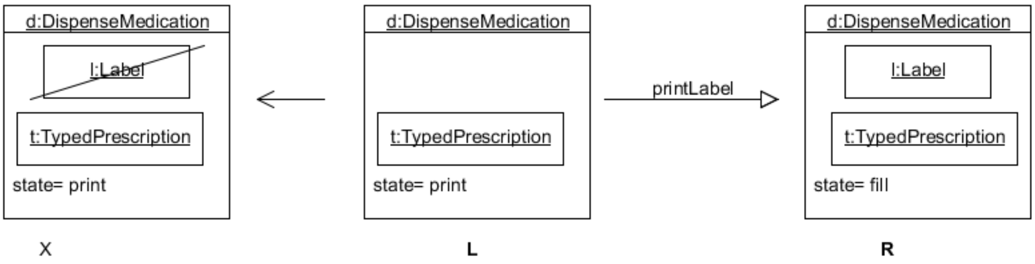


Figure 4.13: GT Rule with NAC format 1

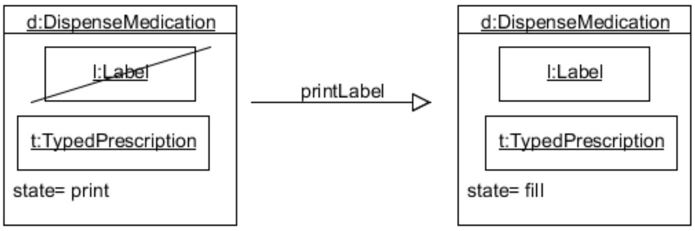


Figure 4.14: GT Rule with NAC format 2

Figure 4.15 illustrates an arithmetic algebraic constraint, which states that if a *person* is assigned to *case* he or she must be in the same *group* as the *case*.

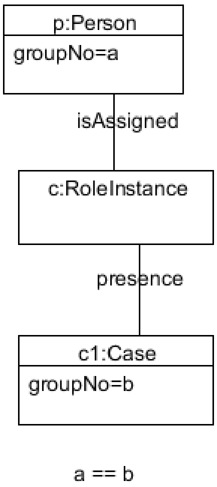


Figure 4.15: Algebraic Arithmetic Constraint

Definition 4.2.8. ***Typed Attributed Graph Transformation System with Negative Application Conditions and Constraints***
A typed attributed graph transformation system (TAGTS) with application conditions and constraints is a tuple (ATG, P, π, C) *where* ATG *is an attributed type graph,* P *is a set of rule names,* π *maps rule names to* ATG*-typed graph transformation rules over (ATG,X) with application conditions, and* C *is a set of boolean constraints.*

Given a rule $\hat{p} = (P, N)$ *a transformation* $G \overset{\hat{p},m}{\Rightarrow} H$ *with application conditions and constraints is a transformation* $G \overset{p,m}{\Rightarrow} H$ *using the underlying rule* p *such that* m *satisfies* N *and* H *satisfies* C*.*

4.3 Stochastic Graph Transformation Systems

Stochastic Graph Transformation Systems (SGTSs) [68] are used to associate estimated (stochastic) time to business process tasks in stage 2 of the methodology as described and illustrated in Section 9.6.7 and visually denoted in Section 7.1. SGTSs are also used in stage 4 of the methodology for stochastic simulation as described in Section 9.8.1. SGTSs are used in the methodology to support modelling of non-functional aspects such as performance and reliability [68].

Stochastic time can be expressed using probability distributions because they describe the range of possible values that a random variable can attain and the probabilities of the occurrences of these values. SGTS (Stochastic Graph transformation System) extends a GTS by associating a continuous distribution function (CDF) with every rule. A continuous distribution function (CDF) identifies the probability of the value falling within a particular interval using a probability density function (PDF) to describe the relative likelihood [22]. The rule distribution is used to compute the time expected to elapse (scheduled delay) prior the application of that particular rule. The behaviour of SGTS can be described as a stochastic process over continuous time where the application of transformation rules is dominated by continuous probability distributions [30]. This is because each active rule-match pair is associated with an independent random variable (timer) which is randomly set according to a continuous probability function. The stochastic graph transformation simulation algorithm will be further discussed in Section 4.4.

Therefore for a GTS to become a SGTS the GT rules defined in the GTS will need to be associated with continuous distribution functions (CDF) such as normal and exponential. First, the GT rules need to be categorized by types of distribution, which can be accomplished by using distributing fitting if sample data are available. Distributing fitting is a procedure of selecting a statistical distribution that best fits the data set [62]. However, if sample data are not available GT rules can be categorized based on distribution properties and theories such as the memorylessness property and the central limit theorem. Exponential distributions are the only memoryless continuous probability distribution, hence memorylessness is a unique categorization for exponential distributions. The memoryless property means that the conditional probability satisfies $P(X > x+y|X > x) = P(X > y), \forall x, y \geq 0$. This formula means that random variable X has the property that "the future is independent of the past", i.e. if a light bulb has survived x units of time then the chance that it survives a further y units of time is the same as that of a fresh bulb surviving y units of time [7]. Exponential distributions are commonly used to represent the time between events that happen at a constant average rate, interarrival times and rate of decay. For instance, if the rate of arrivals is known, such as the time it takes before receiving a telephone call, then the activity can be defined as an exponential distribution. On the other hand, the central limit theorem is a means to classify a distribution as a normal distribution. The theorem states that if the distribution is an average of many independent identically distributed random variables, then the distribution tends towards a normal distribution [76]. The average length of time it takes to complete an action, such as the average time to cut a persons hair, can be categorized as a normal distribution or a lognormal distribution, which defines a positive normal distribution.

After categorizing the distribution, the corresponding scheduled delays (probability values) need to be assigned for each GT rule. For instance, the exponential distribution requires corresponding rate (λ) and average delay (λ^{-1}), whereas the lognormal distributions requires mean (μ) and variance (σ^2) values. If the time it takes to fill a prescription in a pharmacy is considered to be a normal distribution and it takes an average of 1.17 mins with a variance of 0.39583, then normal distribution can be visually denoted in the CDF graph in Figure 4.16. On the other hand, if the arrival rate of new customers to the pharmacy is every 5 minutes, the rate is categorized as an exponential distribution and is denoted in the CDF graph in Figure 4.17. The λ for this example is calculated as follows: λ= (5+10+15)/3= 10, such that average represents the 5 minute interval. The CDF graphs describes the probability that a real-valued random variable X with a given probability distribution will be found at a value less than or equal to x, for instance in the CDF defined in Figure 4.16 the probability that the time it takes to fill a prescription is less than or equal to 2 mins is 0.9 i.e. $P(X \leq 2) = F(x = 2) = 0.9$. The CDF function for exponential is $F(x, \lambda) = 1 - e^{-\lambda(X)}$, therefore probability that X is less or equal to 0.1 in the exponential CDF graph defined in Figure 4.17 is as follows: $P(X \leq 0.1) = F(x = 0.2, \lambda = 10) = 1 - e^{-10(0.2)}$=0.864. Stochastic graph transformation systems is further illustrated in Section 11.7.

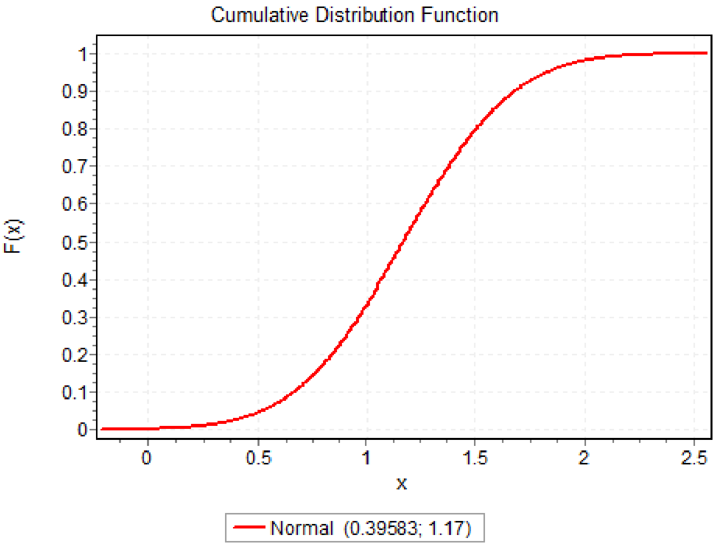


Figure 4.16: Normal CDF: Time it Takes to Fill a Prescription
μ=1.17 σ^2=0.39583

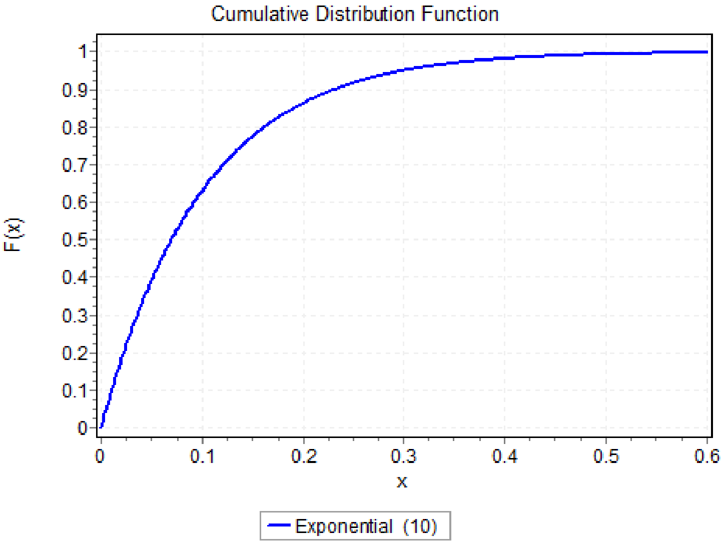


Figure 4.17: Exponential CDF: Arrival Rate of New Pharmacy Customers
(λ^{-1})=1/10=0.1

Formally, a stochastic graph transformation [68] is defined as $SGTS = (TG, P, \pi, F)$, which consists of a graph transformation system $G = (TG, P, \pi)$ and a function $F : P \rightarrow \mathbb{R}_+ \rightarrow [0, 1]$ which associates a probability distribution function (PDF) $F(p) : \mathbb{R}_+ \rightarrow [0, 1]$ to every rule.

4.4 Stochastic Simulation

Simulations are a flexible analysis technique, which can aid in answering analytical questions (Section 1.2). They provide a good insight into the system being modelled through repeated execution of a system with the aid of a computer. A simulation model is a particular type of mathematical model of a system. Models can be classified as static or dynamic, deterministic or stochastic, and discrete or continuous. Dynamic simulation models represent systems as they change over time; whereas static simulation models represent systems at a particular point in time. Stochastic simulation implies experimenting with the model over time including sampling stochastic variants from probability distributions [55]. A stochastic simulation model has one or more random variables (delays) as inputs; deterministic models do not contain random variables. In a stochastic simulation, the output measures must be treated as statistical estimates of the true characteristics of the system, because random inputs lead to random outputs [8]. Knowledge of statistics is helpful for interpreting the simulation results.

Stochastic Simulation can be applied to SGTS with simulation tools such as Graph-based Stochastic Simulation (GraSS) [30, 68]. GraSS is a tool developed in Java-Eclipse that extends VIsual Automated model TRAnsformations (VIATRA2) [72] model transformation plugin with a control based on the SSJ library for Stochastic Simulation in Java. GraSS uses an XML file to load the user-defined rule distributions and probe specification. Events correspond to GT rules with matches, whereas probes are precondition patterns that are used to gather statistics. Probes are similar to queries which identify if a particular structure exists in a graph using identical pre and post conditions in a GT rule. Hence, probes can produce statistical data by counting occurrences of graph structure during a simulation run. The distribution specification in GraSS is currently limited to lognormal and exponential distribution data. The tools results report data about average occurrences of probe matches and rule executions. This simulation tool is used in stages 3 and 4 of the Methodology in Chapter 9. Formally, each simulation run consists of the following steps [30]:

States (G, t, s) of the simulation are given by the current graph G, the simulation time t, and the schedule s, a set of timed events $s : E(G) \to \mathbb{R}_+$ with events given by $E(G) = \{(p, m) | p \in P \wedge \pi(p) = L \leftarrow K \rightarrow R \wedge \exists m : L \to G \text{ satisfying } p\text{'s gluing conditions and application conditions}\}$ maps all rule matches enabled in G to their scheduled time.

1. Initially, the current graph $G = G_0$ is the start graph, the time is set to $t = 0$ and the scheduled time $s(p, m) = RN_{F(p)}$ for each enabled event (p, m) is selected randomly based on $p's$ probability distribution.

2. For each simulation step

   (a) the first event $e = (p, m)$ is identified, i.e., such that for all events e', $s(e) \leq s(e')$. Rule p applied at match m to the current graph

G produces the new current graph H, i.e., $G \overset{p,m}{\Rightarrow} H$.

(b) the simulation time is advanced to $t = s(e)$.

(c) the new schedule s', based on the updated set of enabled events $E(H)$, is defined for all $(p', n') \in E(H)$ with $\pi(p') = L' \leftarrow K' \rightarrow R'$ and the transformation $G \overset{p,m}{\Rightarrow} H$

The result is a *(simulation) run* $r = (G_0 \overset{p_1,m_1,t_1}{\Rightarrow} \cdots \overset{p_n,m_n,t_n}{\Rightarrow} G_n)$, i.e., a transformation sequence where steps are labelled by time stamps $t_1, \cdots, t_n \in \mathbb{R}_+$ with $t_i < t_{i+1}$ for all $i \in \{1, \cdots, n-1\}$. The completion time of the run is called $ct(r) = t_n$.

This simulation algorithm can be illustrated given the following five GT rules and corresponding distribution values in Table 4.1.

Rule Name	**Distribution**	
	Mean	*Variance*
ruleA	1	0.34
ruleB	2	0.50
ruleC	3	0.34
ruleD	4	0.10
ruleE	5	0.24

Table 4.1: GT Rules with Corresponding Normal Distributions

Example Simulation runs:

1. The simulation time t is set to 0.

2. The set of enabled GT rule matches is obtained from the GT engine.

   - 2 matches of ruleB

   - 3 matches of ruleD

- 1 match of ruleA

3. A random number generator (RNG) assigns a scheduling time to each active rule-match pair (event) $e = (p, m)$, based on the probability distribution $s(e)$ associated with the rule.

    - $ruleB_1 = (ruleB, m_1)$ and $s(ruleB_1)$= 1.50
    - $ruleB_2 = (ruleB, m_2)$ and $s(ruleB_2)$= 2.06
    - $ruleD_1 = (ruleD, m_1)$ and $s(ruleD_1)$= 3.10
    - $ruleD_2 = (ruleD, m_2)$ and $s(ruleD_2)$= 2.80
    - $ruleD_3 = (ruleD, m_3)$ and $s(ruleD_3)$= 3.05
    - $ruleA_1 = (ruleA, m_1)$ and $s(ruleA_1)$= 1.25

4. A state list contains the active rule-match pairs, ordered by time as visually represented in the timeline defined in Figure 4.18.

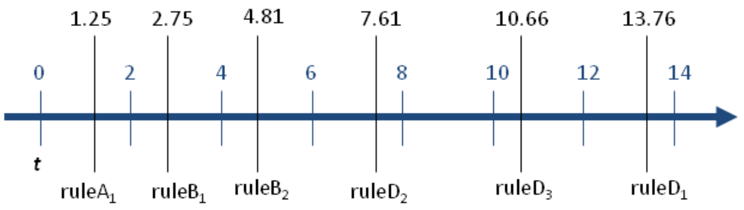


Figure 4.18: Timeline of scheduled rule-match pairs at ***t=0***

5. The first event (e) is removed from the state list.

    - $ruleA_1$

6. The simulation time is increased to the time of event e ($s(ruleA_1)$).

- t= 1.25

7. The event e is executed by the GT engine.

   - $ruleA_1$

8. The new set of enabled GT rule matches is obtained from the GT engine. Due to the execution of $ruleA_1$ the matches for ruleB were disabled; however a rule match for ruleE became enabled.

   - 3 matches of ruleD
   - 1 match of ruleE

9. A random number generator (RNG) assigns a scheduling time to each active rule-match pair (event), based on the probability distribution associated with the rule.

   - $ruleE_1 = (ruleE, m_1)$ and $s(ruleE_1)$= 5.15
   - $ruleD_1 = (ruleD, m_1)$ and $s(ruleD_1)$= 3.10
   - $ruleD_2 = (ruleD, m_2)$ and $s(ruleD_2)$= 2.80
   - $ruleD_3 = (ruleD, m_3)$ and $s(ruleD_3)$= 3.05

10. A state list contains the active rule-match pairs, ordered by time as visually represented in the timeline defined in Figure 4.19.

    - $s(ruleD_2)$= 2.80
    - $s(ruleD_3)$= 3.05
    - $s(ruleD_1)$= 3.10

- $s(ruleE_1)$= 5.15

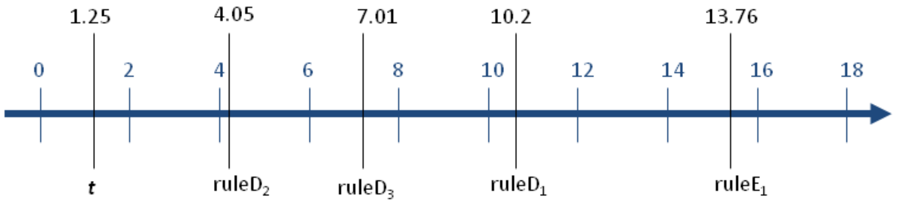


Figure 4.19: Timeline of scheduled rule-match pairs at ***t=1.25***

11. The first event (e) is removed from the state list.
    - $ruleD_2$
12. The simulation time is increased by the time of event e $(s(ruleD_2))$.
    - t= 1.25 + 2.80 = 4.05
13. The simulation continues by following steps 1-7, using the current simulation time which was computed from time associated to the preceding events.

4.5 Summary

In this chapter, definitions for the relevant concepts of Graph Transformation (GT), Typed Attributed Graph Transformation (TAGT), Stochastic Graph Transformation (SGT) and stochastic simulation were introduced.

The GT section defined and illustrated basic GT concepts, whereas the TAGT section presented extension to GT with attributes for the nodes and vertices. GTs are seen throughout the thesis including: stages 2–4 of the methodology (Chapter 11-13) and in Chapter 7.

Lastly, the stochastic graph transformation system and stochastic simulation sections describe the necessary components for stochastic simulation. SGTS is revisited in the methodology (Section 9.6.7) and in the rule-based specification (Section 7.1). This section also introduced GraSS [30, 68], which is the stochastic simulation tool that is used for stages 3 and 4 of the StADy methodology (Sections 9.7 and 9.8). It is an extension of VIATRA2 [72], hence the metamodel, models and GT rules would need to be translated into VIATRA2 syntax, which is further discussed in Chapter 8.

Chapter 5

Pharmacy Case Study

In order to validate the concept requirements defined in Section 2.1, a pharmacy case study is considered. The following is a recap of the defined concept requirements: dynamic (re)-assignment (R1.1); role promotion, demotion and temporary promotion (R1.2); access control (R1.3); process scheduling influenced by deadlines and priorities (R2.1); non-deterministic duration of tasks (R2.2); temporal escalation handling (R2.3); load balancing (R2.4); human error and unpredictability (R2.5).

The people involved in a pharmacy dispensary have different levels of access rights depending on their qualification or role (R1.3). This process is about dispensing medication at a pharmacy, which occurs at a dispensary. Positions in the dispensary include registered pharmacists, pharmacy students, pharmacy technicians and cashiers as illustrated in Figures 5.1 and 5.2. The dispensary also requires external involvement of customers in the role of patients with basic access rights (R1.3). A person's position in the hierarchy of actors reflects their allowable role assignments (R1.1,1.2). For instance if a filling technician role is required to be assigned and both a registered pharmacist and a pharmacy technician are available, the technician should take on that role (R1.1). However in a situation where no pharmacy technicians are available, the pharmacist is flexible enough to take on the filling technician role (R1.1).

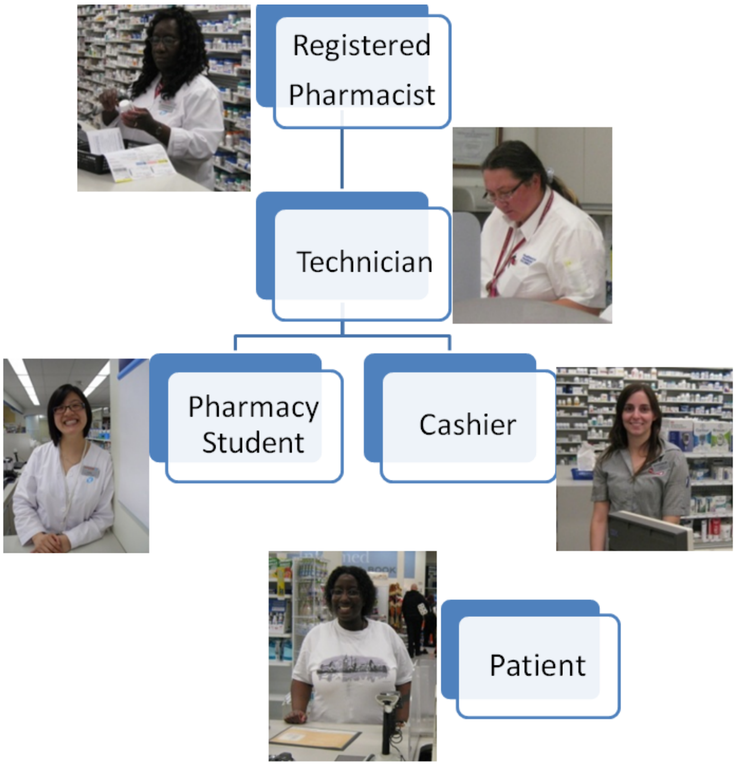


Figure 5.1: Actor Illustration

Figure 5.2: Role Illustration

5.1 Business Environment

Workers are likely to react to emergencies by stopping their predetermined routine schedule of work (R2.1). An example is an urgent prescription that needs to be dispensed by a pharmacist while the current pharmacist on duty is busy. In such a case, the urgent prescription is given priority over other tasks (R2.1).

In a chain of pharmacy stores, the work load may vary across the different stores because it is dependent on the number of prescription requests at particular locations. The customer has a choice to pick up the filled prescription from the store or have it delivered to his or her home. If the pharmacy is extremely busy and the requested prescription has been ordered for delivery, then it can be transferred and filled by any pharmacy that is geographically nearby. This is an example of load balancing (R2.4). However, there is a restriction that prescriptions ordered to be picked up at a store can only be filled at that store.

The pharmacy business process consists of the following sequence of states: *type*, *print*, *fill*, *check*, *payment* and *counsel*, as illustrated in Figure 5.3. Each state represents the stage of that particular case in the process. Pharmaceutical processes are safety critical and must be checked for correctness and accuracy. Hence, each task includes error checking to ensure that it is performed correctly. For instance, if a state is skipped, the process will backtrack to the previous state (R2.5).

A typical process is initiated by a patient requesting a prescription to be filled. The patient is informed of an expected finishing time, resulting in a deadline for the case. The prescription is typed into the pharmacy database by an entry technician and the corresponding bottle-label is printed. The earliest due date (EDD) and priority-based scheduling protocols assist in determining the order prescriptions get filled (R2.1). The prescription is filled by a filling technician and checked by the dispensing pharmacist. The pharmacy cashier receives payment from the patient and then the patient is counselled by the dispensing pharmacist and given the filled prescription.

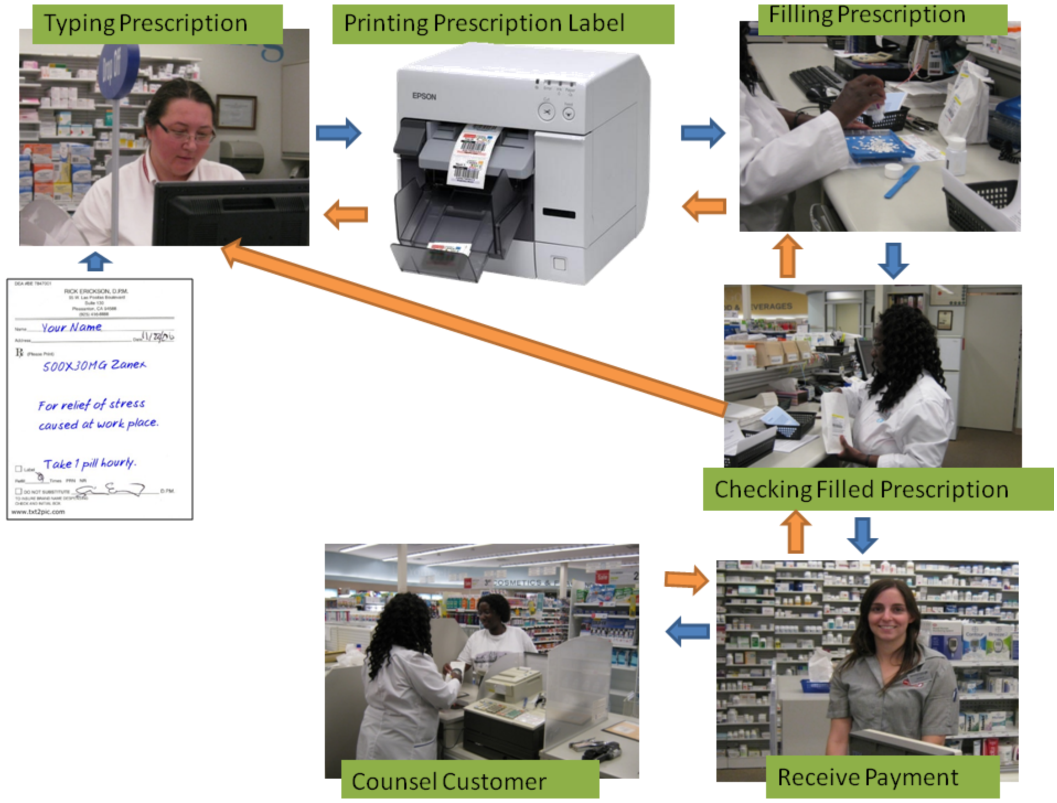


Figure 5.3: Pictorial Version of States

The deadlines are approximations based on the method of placing an order and the degree of the customer's urgency (R2.1). For instance, if a customer decides to wait in the store, then the prescription is expected to be finished within 15 minutes. If the customer decides to return to pick up the prescription, then the prescription is expected to be finished within 1 hour. If the prescription was requested online, then it is expected to be completed within 24 hours.

Temporal exception handling (R2.3) can be triggered when the case is approaching a deadline, has reached the deadline, or has exceeded the deadline. Escalations may also result in people overstepping their permissions and being temporarily assigned to a role required to deal with the exception (R1.2,2.3). An example is a prescription (case) which is 5 minutes past that deadline and has an escalation defined that permits a pharmacy cashier to temporarily take on the role as filling technician.

5.2 UML Models

This section models the case study with UML use-case, class and activity diagrams. The illustrations show the usefulness of each of the diagrams; however each section also points out diagram deficiencies and concludes with possible methods of improvements.

5.2.1 Use-Case Diagram

Figure 5.4 illustrates the pharmacy business process in a UML use-case diagram. The diagram specifies the pharmacy *dispensary* system, which contains various use-cases such as *receive prescription* and *change escalation.* The corresponding use-case descriptions are shown in Tables 5.1 to 5.4. The diagram also specifies which of the seven specified actors can perform the various use-cases. Some of the actors are related through hierarchy, hence can inherit each other's cases, i.e. a *RegPharmacist* can inherit type prescription use-case from the *Technician.*

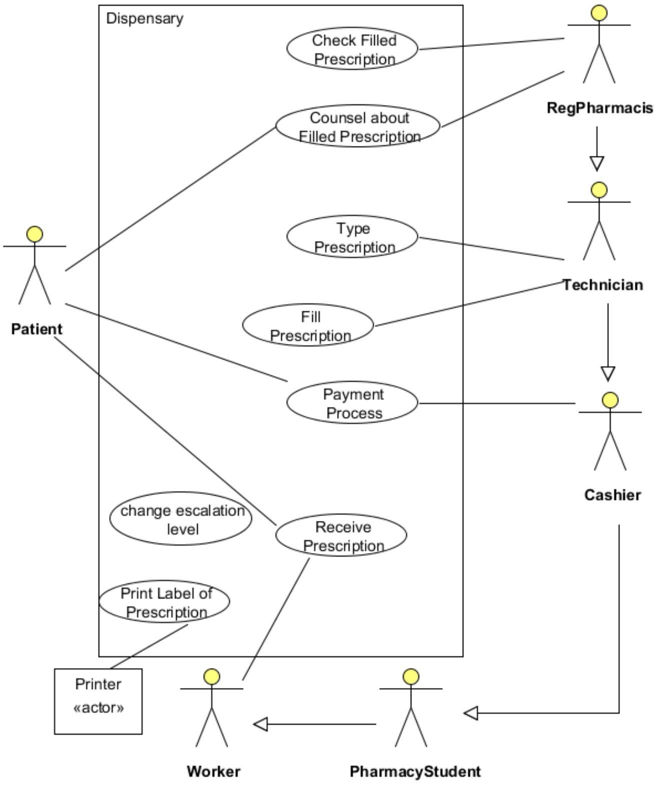


Figure 5.4: UML Use-Case Diagram of the Pharmacy Business Process

Name:	Receive Prescription
Primary Actor:	Worker
Goal (of the User):	receive prescription and set deadline
Precondition:	prescription request to be filled
Postcondition (successful execution):	new prescription case
Trigger Event:	new prescription fill request
System (implementing it):	Dispensary
Participating Actor:	Worker, Patient

Table 5.1: Receive Prescription Case Details

Step	User	Action
1	Patient	brings Prescription to store
2	Worker	receives prescription
3	Worker	informs patient that prescription will be filled within 15 minutes and sets completion time for 15 minutes from the current time

Table 5.2: Case Scenario: Receive Prescription

Step	Condition for alternative	Alternative Action
1	refill prescription ordered online	Prescription is set for online refill
3	Patient plans to return later to pickup prescription	Prescription is set for future pickup and completion time is set to 1 hour from the current time
3	Patient asks for prescription to be delivered	Prescription is set for delivery and completion time is set to 15 mins from the current time
3	if the prescription is set as online refill	The Worker sets the deadline to 24 hour from the current time

Table 5.3: Case Alternative Scenario: Receive Prescription

Name:	Change Escalation Level
Primary Actor:	
Goal (of the User):	Increment Escalation Level
Precondition:	if (Prescription case deadline is less than or equal to 5 minutes away from the deadline) else if (Prescription case has reached or passed the deadline) else if (Prescription case is 5 mins or more passed the deadline)
Postcondition (successful execution):	if (Prescription case deadline is less than or equal to 5 minutes away from the deadline) then set Escalation level to 1 else if (Prescription case has reached or passed the deadline) then set Escalation level to 2 else if (Prescription case is 5 mins or more passed the deadline) then set Escalation level to 3
Trigger Event:	case deadline approaching time
System (implementing it):	Dispensary
Participating Actor:	

Table 5.4: Use-Case Details: Change Escalation level

Use-case diagrams tend to focus on the production-level aspects of a business process such as *receive prescription* and *fill prescription.* Support and managerial-level business aspects such as *change escalation level* and assignment selection are difficult to represent in standard use-case diagrams. This is because use-case's in use-case diagrams are intended to describe sequences of actions, yet managerial and support-level actions do not occur in a sequence. The *receive prescription* case description specifies the overall case details (Table 5.1), case scenario (Table 5.2), and alternatives (Table 5.3). On the other hand the *change escalation level* can only be presented using a basic case description table (Table 5.4) because it is does not represent a goal since it occurs as a support-level function as opposed to a production-level aspect. In order to capture the functionality correctly, the pre and post conditions in the use-case description in Table 5.4 requires a combination of if and else conditions.

This issue can be resolved by extending use-case diagram to encode the alternative and conditional details visually into the diagram using annotations. The extension can define constraints on actors specifying conditions that will permit them to take on particular roles such as constraints corresponding to gained capabilities or required escalation levels. These annotations can be used to refine *actor–case* relation into an *actor–role–case* relation as done in RBAC [57] (Section 3.2). The extended use-case diagram is illustrated in Section 6.3.

The StADy approach proposes to capture the written use-case descriptions visually by using graph transformation rules. This approach is illustrated in Chapter 7: graph-transformation rules are used for defining pre and post conditions in a visual way, by simply decomposing the description and replacing it with three distinct graph transformation rules. Each of these rules can be used to specify the trigger to increment the escalation level. Graph transformation rules can provide a facility to represent both low level management functions and business level operations.

5.2.2 Class Diagram

This section defines the pharmacy business process artifacts using an UML class diagram. The class diagram in Figure 5.5 specifies the relations between pharmacy artifacts, actors and products (cases). Each *DispenseMedication* case can contains attributes such as enumerated *type*, which can be used to distinguish different prescription types, i.e., *delivery* or *walkin.* The actors defined in the use-case diagram (Figure 5.4) are specified using the ≪actor≫ stereotype, such as *Cashier* and *Technician.* The *can access* relation is used to specify which artifacts the actor is permitted to access, such as the *Cashier* can access *payment* and *bag* artifacts. The hierarchy relationship defined in Figure 5.4 corresponds to the access right inheritance relationship in the class diagram in Figure 5.5; for example the pharmacist is permitted to access the payment due to inheritance which was specified in the use-case diagram.

This direct actor–case relation is weak in terms of access control because the actor has direct permission to access the case, whereas if a role is in between actor and case the actor would need permission to perform the role and the role would need permission to access the case, hence strengthening access control. Class diagrams can be refined to increase the access control by containing an *actor–role–object* relation, as done in RBAC (Section 3.2). Hence, StADy defines an extended class diagram which lists the permitted roles inside the corresponding artifact under the heading ≪*role*≫. This is illustrated in Section 6.3.

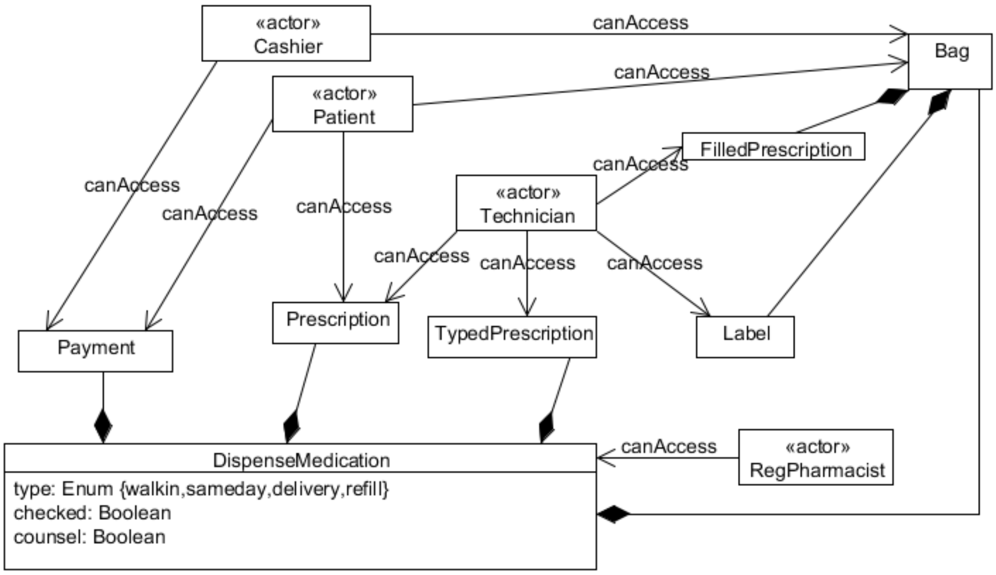


Figure 5.5: UML Class Diagram of the Pharmacy Business Process

5.2.3 Activity Diagram

This section defines a process centric view of the pharmacy business using a standard UML activity diagram. The activity diagram in Figure 5.6 presents the procedural logic of the pharmacy business process where the flow arrows are used to specify the order in which actions occur. The sub-activity diagrams are later defined in Section 10.2. Swimlanes are used to specify the actors required to perform the activities, such as *Worker* and *RegPharmacist*. Coherent object flows are also defined using pins on the end points of the activity nodes such as *Prescription* and *Case* types.

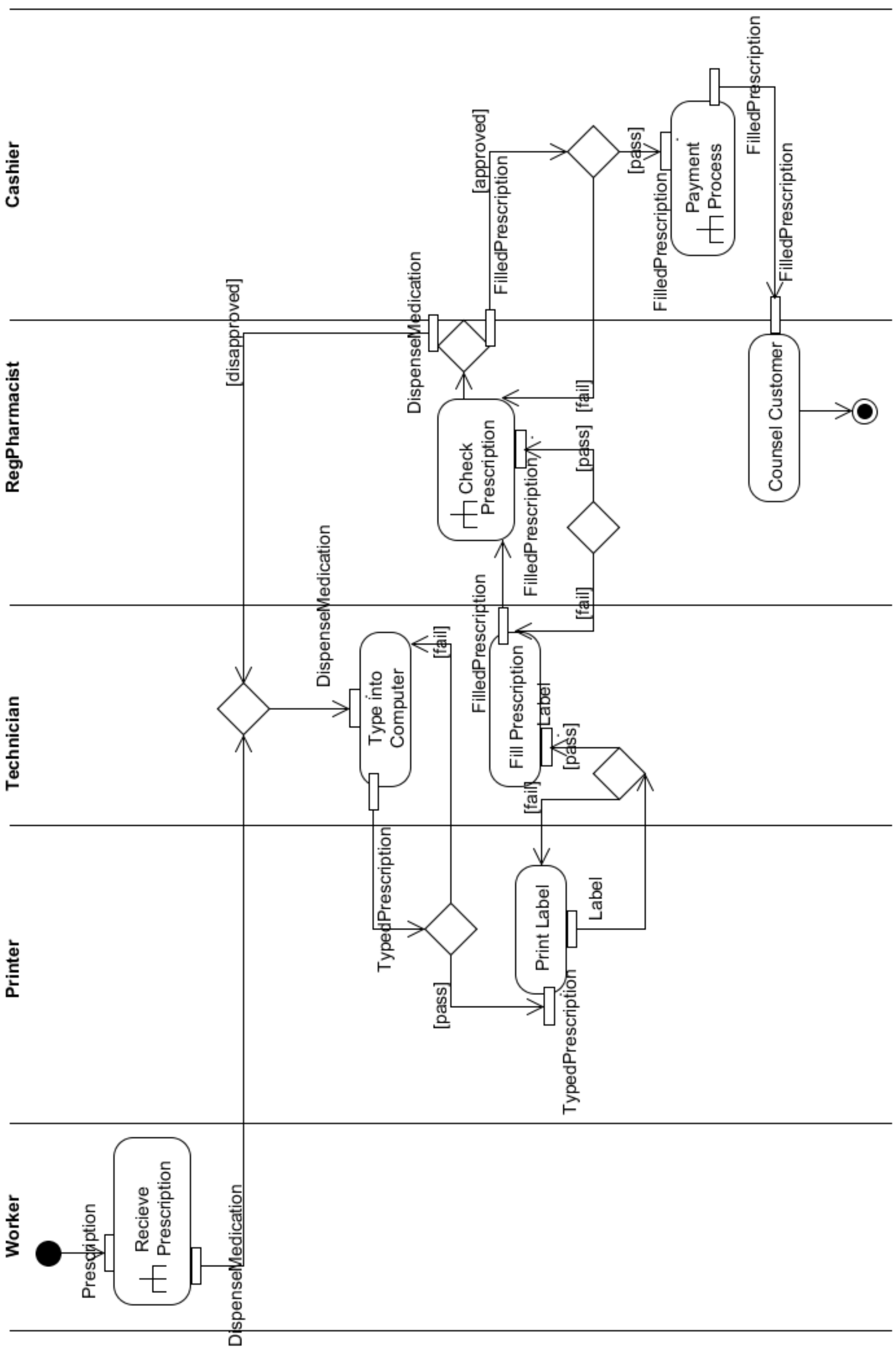


Figure 5.6: UML Activity Diagram of the Pharmacy Business Process

This diagram captures the basic procedural logic of the dispensary control flow; however it is missing managerial aspects such as role assignment. This is due to the fact that activity diagrams were not intended for this level of detail. Swim lanes are used to specify which actor performs which task; however swim lanes lack detail on the actual person that is assigned to take on that task. The activity diagram is defined from the business perspective which focuses on the domain specific activities and omits the majority of the managerial details such as role assignment, scheduling, and load balancing.

Non-deterministic managerial aspects are generically defined in the StADy approach in graph transformation rules, whereas the existing control-flow behaviour of the domain-specific process-centric definitions can be preserved by using states. Hence, the rule-based approach can be used to define the non-deterministic and control-based processes. This is illustrated in Chapter 7.

5.3 Summary

This chapter introduced a pharmacy case study which will be used for illustration of the StADy language (Chapters 6-7) and StADy methodology (Chapters 10-14). The next chapter will introduce the StADy Configuration Modelling Language.

Chapter 6

Configuration Modelling Language

This chapter introduces the StADy configuration modelling language, which is a sub language of the StADy language. It is required for the process execution design stage of the methodology which will be discussed in Chapter 9. Section 6.1 introduces the configuration modelling language's linguistic metamodel (M2-level) [28]. The M2 level of the configuration modelling language generically defines business processes in terms of various elements including actor, role, case and artifact elements. These elements can be used to instantiate the metamodel into various complicated business processes at the instance (M1) level which is domain-specifically defined. For illustration purposes the small pharmacy business domain defined in Chapter 5 is used to define an instance of the metamodel in Section 6.2. Section 6.3 defines the concrete representation of the class-level (M1-O1) of the abstract model. On the other hand, Section 6.4 defines notation for representing object-level

(M1-O0) elements and defines the representation of M1-O0 representation of the abstract model. This chapter focuses on the static aspects of the StADy language, whereas the dynamic aspects of the language will be discussed in Chapter 7.

6.1 Metamodel

A linguistic metamodel [28] (Section 3.3) was defined for modelling the orchestration between people and technical components. Figure 6.1 presents a detailed linguistic metamodel view. This metamodel combines type and instance-level concepts, located on opposite sides of the diagram for clarity, and related by (ontological [28]) "instance-of" relations. Thus, instances of this metamodel (M2) can represent both class-level (O1) and object-level (O0) features.

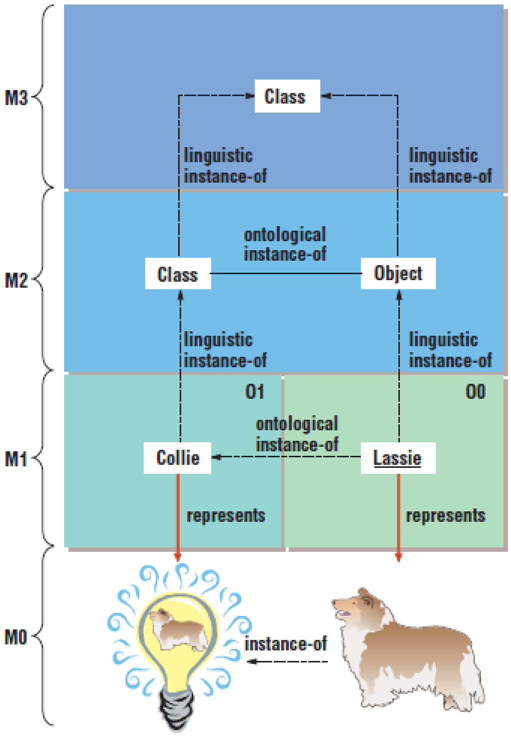


Figure 6.1: Linguistic Metamodeling View [5]

The linguistic [28] metamodel shown in Figure 6.2 captures the structure of the configuration modelling language. Being a *linguistic* metamodel, it combines type and instance-level concepts, located on opposite sides of the diagram for clarity and related by the “instance-of” relationship.

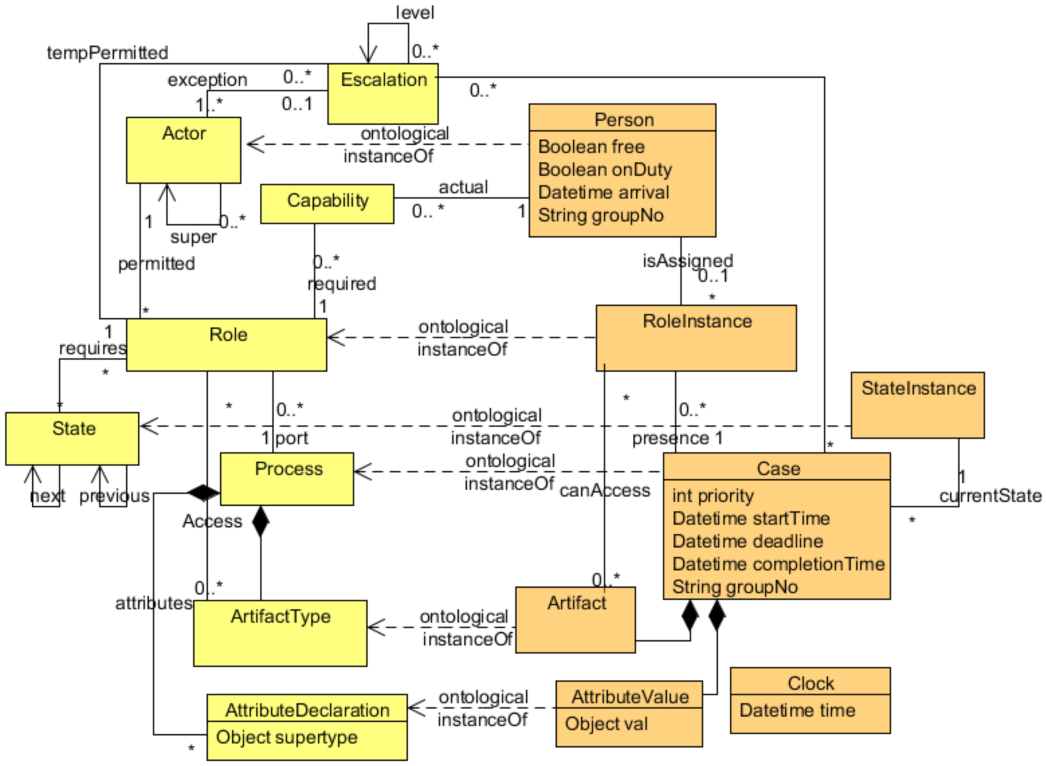


Figure 6.2: Metamodel (M2)

The metamodel includes elements of: *Actor*, *Role*, *Process*, *ArtifactType*, *AttributeDeclaration*, *State* and their corresponding *ontological instances of* [28] (Section 3.3) *Person*, *RoleInstance*, *Case*, *Artifact*, *AttributeValue* and *StateInstance*. It also contains *Capability* and *Escalation* elements. *Escalation* is related to exceptional *Case* to enable an *Actor* to be *tempPermitted* (temporarily) to be assigned to a given Role. Their corresponding levels are represented through hierarchy, whereas *Capability* is represented in the *actual* association between *Person* and *Capability*. If the *Actor* is not directly permitted to a *Role* he/she can obtain the *Capability* required. The metamodel also contains a *Clock* object-level feature which is used to capture the current local date and time of the process.

The StADy Transformation Modelling language (Chapter 7) uses typed attributed graph transformation without inheritance (Section 4.2) because it is uses the StADy metamodel as the type graph for the GTS and it does not have an actual inheritance relation instead it encodes an edge called *super*. Inheritance is only visually represented at the concrete M1-O0 object level (Section 6.4). Figure 6.3 shows the type graph's (StADy metamodel's) mapping to *E-graph* elements (Definition 4.2.1).

The central notion in the metamodel is the association of individuals to permissions via roles to permissions, which is influenced by Sandhu's $RBAC_3$ [57] (Section 3.2). $RBAC_3$'s *user*, *role*, *permission* elements maps to the StADy metamodel's *person*, *actor* and *role* elements respectively. A $RBAC_3$ *user* element is equivalent to StADy's *person* element because it is defined as a model of a human being. A $RBAC_3$ *role* element is equivalent to StADy's *actor* element because it is defined as a job function or job title. The $RBAC_3$ *role* also has a *role hierarchy* relation which corresponds to StADy's *Actor–Actor super* relation. A $RBAC_3$ *permission* element is equivalent to StADy's *role* element because it is defined as an approval of particular mode access to one or more objects, such that the objects correspond to the StADy metamodel's *ArtifactType*. The $RBAC_3$ permission assignment relation is equivalent to the *permitted* actor–role relation in StADy's metamodel. The StADy metamodel uses a specialised version of RBAC *constraints* by defining them using *escalation* and *capability* elements. The StADy metamodel thus includes concepts of $RBAC_3$ and extends it to include a visual representation of the interactions between people and their corresponding involvement in the tasks of a business process.

The structure of the metamodel was also influenced by Axenath et al.'s [6] metamodels developed for business process models: organisational structure, functional structure and resources. Similar to O1 and O0 in this metamodel, their metamodels are also divided into two distinct sides of static and dynamic. The static side is the model itself, whereas the dynamic side is the instantiation of the model. Also similar names such as *Case* and *Process* are used in both models, such that a case is defined as an instance of a business process.

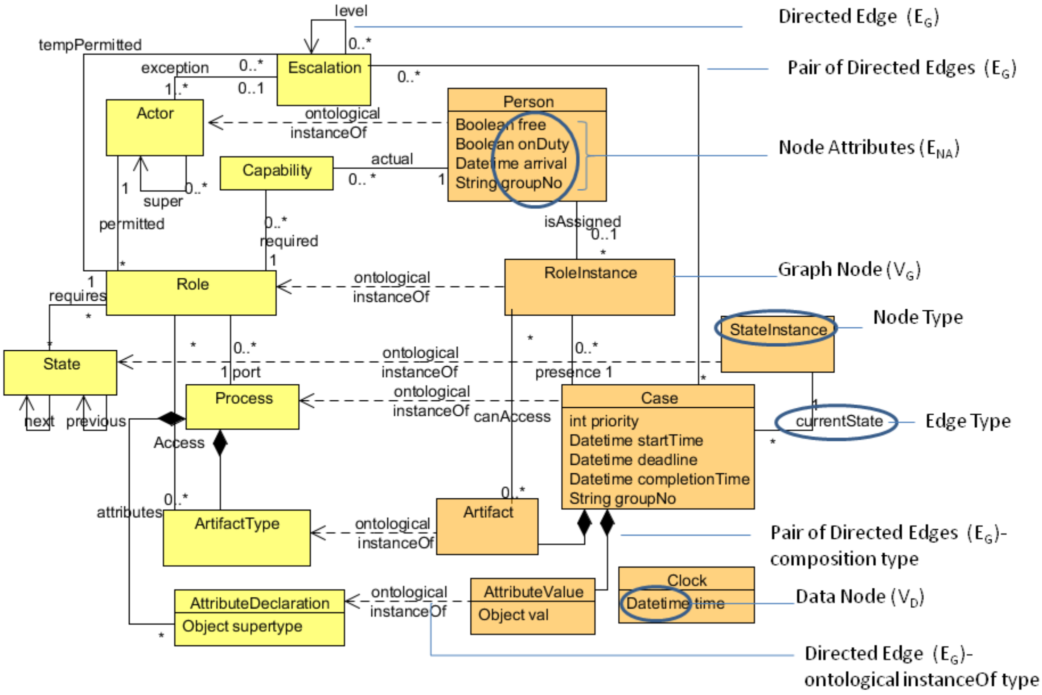


Figure 6.3: Metamodel with mapping to E-graph

6.2 Abstract Syntax M1-O1 and M1-O0

In this section, a M1 instance model is illustrated in abstract syntax using the pharmacy domain described in Chapter 5 as an example. The M1 instance model for the pharmacy domain is defined in Figure 6.4. This section illustrates that the instantiation metamodel defined in Figure 6.2 is an example of a pharmacy model in abstract syntax representation. The StADy language itself is free of references to pharmacy concepts. Since the model is an instance of a *linguistic* metamodel, it combines type and instance-level concepts, located on opposite sides of the diagram for clarity and related by the "instance-of" relationship.

The type-level elements define a dispensary as a *Process* containing various *ArtifactType*s such as prescription and payment. The dispensary *Process* also contains various domain-specific *AttributeDeclaration*s such as an enumerated case type attributes which could be enumerated to represent walkin, delivery, refill and same day prescriptions. Each *ArtifactType* is associated with various *Role*s that can *access* such as the filling technician–filled prescription relation. Each *Role* is associated with *permitted Actor*s such as the entry technician–technician relation. Each *Actor* can inherit permissions from other Actors with the *super* association such as a registered pharmacist inheriting the permissions from a technician. Each *Role* is also associated with the *State* they are involved in such as in payment state the process *requires* a customer and a pharmacy cashier. Each of the *State*s are associated with their corresponding preceding and succeeding *State*s. *Capability* elements are used to connect a *Roles* required *Capability* to a *Person*'s actual obtained *Capability*.

The instance-level elements are ontological instances-of the type level M1 elements and the instance-of the instance level M2 metamodel elements. The model contains the following ontological instance relations: *Actor–Person*, *Process–Case*, *State–StateInstance*, *Role–RoleInstance* and *AttributeDeclaration–AttributeValue*. The model also contains the following instance level metamodel instances: *Escalation*, *Clock* and additional attributes *Person.free*, *Person.name*, *Case.startTime* and *Case.deadline*.

The *Actor–Person* ontological instance relation is used to specify the type of job position a *Person* acquires; for instance in Figure 6.4 Cindy is a registered pharmacist. Each *Person*'s *name* and availability is specified; for instance Bob is not available, whereas Cindy is available to be assigned to a *RoleInstance*. Each *RoleInstance* has an ontological instance relation to a *Role* to specify the type of *Role* that is associated with a *Case*. Each *Case* represents a particular prescription case with corresponding attributes such as start and deadline times and *AttributeValue*s such as domain specific attribute such as the boolean "counsel" attribute shown in Figure 6.4. *Case*s also contain *Artifact*s such as the prescription. Each *Case* is associated with a *StateInstance* which is an ontological relation to corresponding *State*, hence represents the current *State*. *Escalation* levels are also specified and relate the temporary permitted *Role* to an *Actor*. Figure 6.5 illustrates an instance graph mapping to *E-graph* elements (Definition 4.2.1).

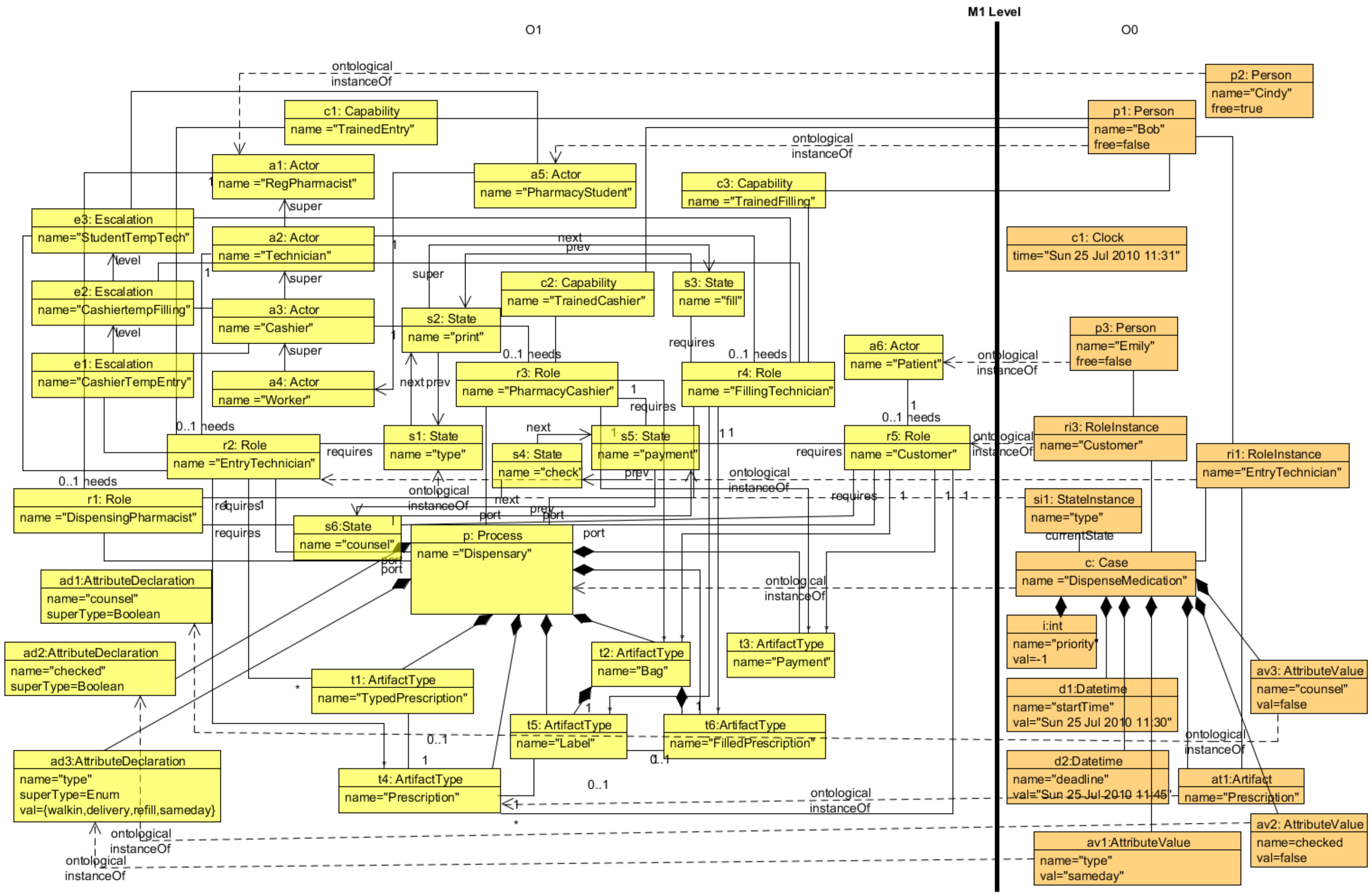


Figure 6.4: M1 level

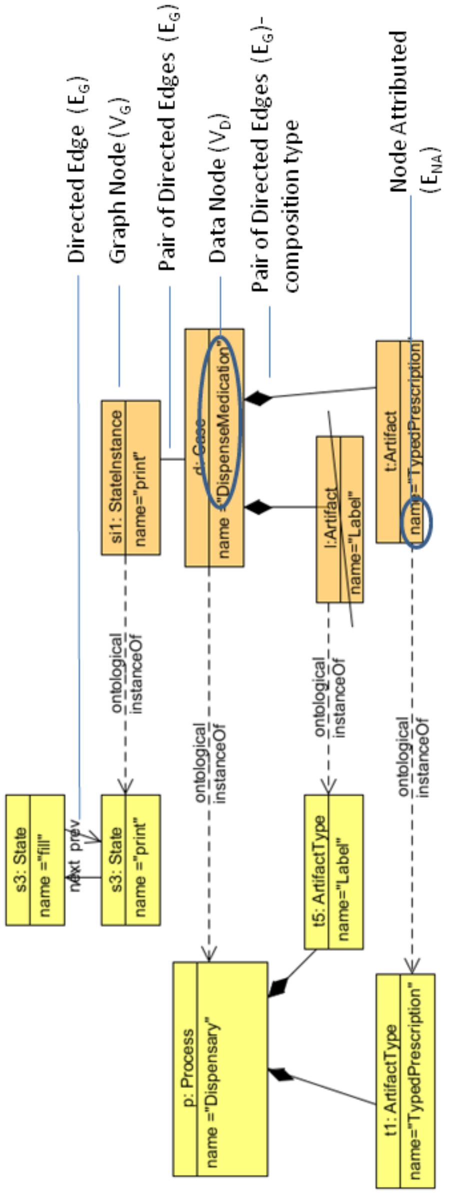


Figure 6.5: M1 abstract instance graph with mapping to E-graph

6.3 Concrete Syntax M1-O1 (Class-Level)

The M1-O1 concrete syntax instance model represents the abstract syntax of the type-level elements of the M1 abstract syntax instance model (Section 6.2) graphically.

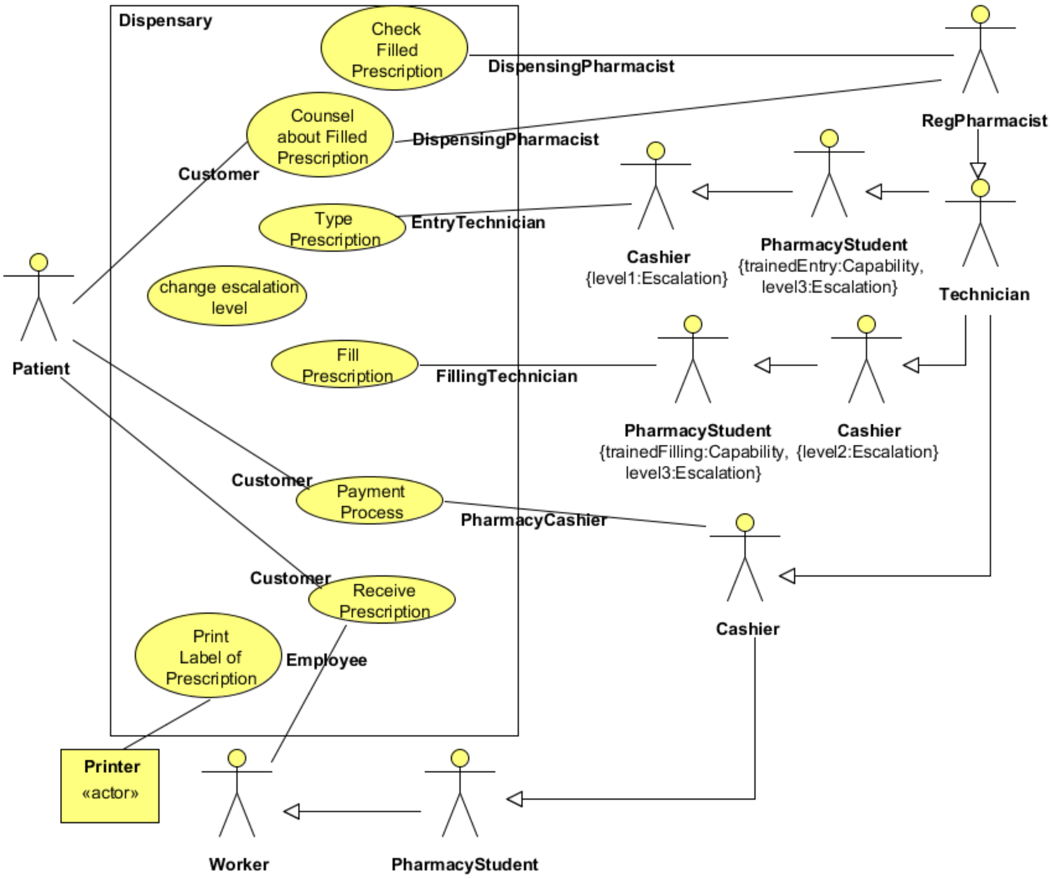


Figure 6.6: Extended Use-Case Diagram (StADy Hierarchy Model)

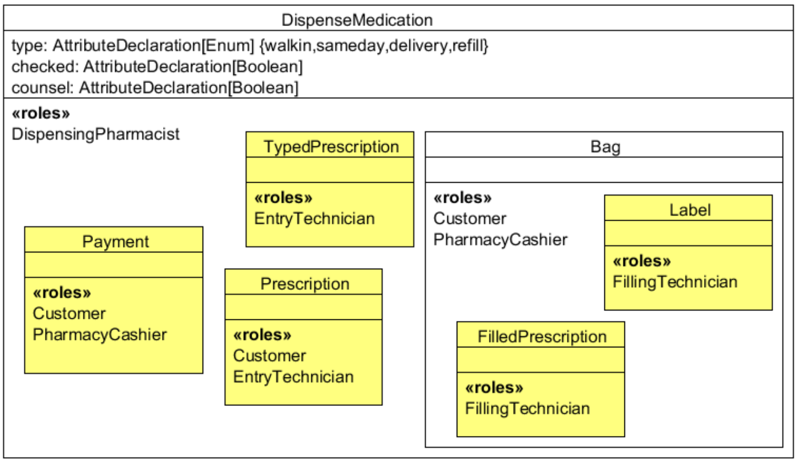


Figure 6.7: Extended Class Diagram (StADy Artifact Model)

Figures 6.6 and 6.7 illustrate the role hierarchies and artifact concepts for the actor hierarchy (super), role association (permitted *Actors*, required *Capabilities*, temporarily Permitted *Escalation*-Actor), and *Process* containment relations. The concrete syntax extends UML use-case diagram and class diagram notation. Figure 6.6 illustrates how to visualise capability and escalation constraints on *Process*, *Actor* and *Role* elements.

Use-cases are also used to model processes, i.e., classes of top-level business objects in hierarchical class models such as the one in Figure 6.7. Attribute declarations appear in the first section of the classes, such as 'counsel:Boolean'. UML composition of classes is visually represented by the containment of classes within a process, i.e., a process is seen as a container for lower-level business objects representing the data relevant to that process. Other attributes such as states, start time and completion time were defined in the metamodel level of this language.

The access control feature (R1.3) is defined in both of these models. In the role hierarchy model, UML's actor hierarchies and allowable roles (actor — use-case relations) are used here, such that the roles are defined at the endpoints of the lines connected to the use-case. Actors allow multiple inheritance and are labelled by constraints corresponding to the capabilities or escalation levels required for promotion. The constraints are defined in set notation underneath the actor symbol. If at least one holds true, the actor can take on a given role.

In the artifact model, actors in the hierarchy model were referred to under the stereotyped keyword ≪role≫ used to define access rights of actors to artifacts. If an actor is assigned to a particular role for a particular *DispenseMedication* case instance, then the actor can access the corresponding artifacts.

6.4 Concrete Syntax M1-O0 (Object-Level)

This section presents the StADy notation and illustrates it in an instance model. The notation for process configurations is pictorial and represents constraints on escalation levels, priorities, time and deadlines in states (instance graphs) or state patterns (graph patterns to be matched by instance graphs). Table 6.1 illustrates the connection between concrete syntax to the abstract syntax at M1-O0 level. The syntax is introduced in Tables 6.2 and 6.3. The values corresponding to escalation, priority and timeline are represented with a positive integer. If the values equal zero, then their corresponding icons will not appear in the Case. The person free and person not free notation define a short-hand notation for the Boolean attribute free in metaclass Person. The terms “internal” and “external” access right notation (Table 6.3) are used to define if an actor can access the artifact contents of the case. If a person’s access rights are internal, then he or she can access some of the artifacts contained in the corresponding case, whereas external access represents lack of access to artifacts contained in the case. Figure 6.8 is the concrete syntax representation of the M1-O0 abstract model in Figure 6.4.

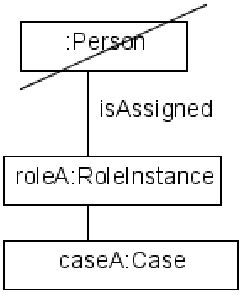
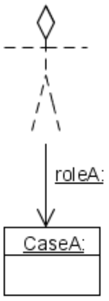
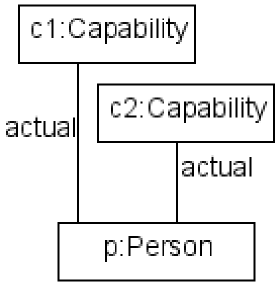
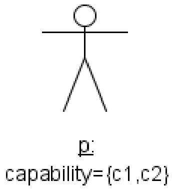
Name	Abstract	Concrete
Request Assignment		
Person Capability	 c1:Capability c2:Capability actual actual p:Person	

Table 6.1: Abstract Syntax at M1-O0 vs Concrete Notation

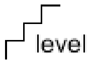
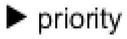
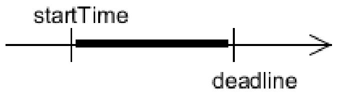
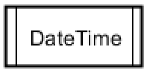
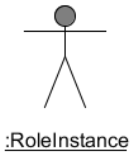
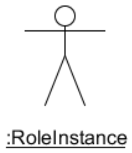
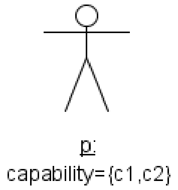
Name	**Description**	**Notation**
Escalation	a Case's escalation level	
Priority	a priority attribute	
Timeline	the startTime and deadline attributes	
Clock	The current time	
Person not free	Person.free=false	
Person free	Person.free=true	
Capability	Person's actual Capability	
State	State Metadata tagged value of current state	{state=value}

Table 6.2: Notation Part 1 of 2

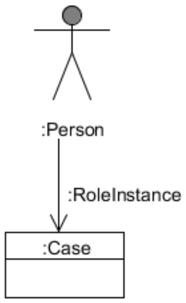
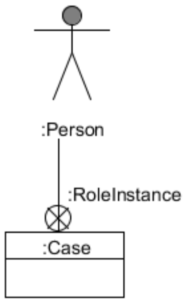
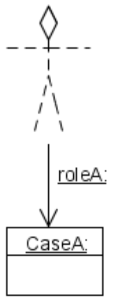
Internal Access	Access to some of the artifacts within case	
External Access	Lack of access to artifacts within case	
Request Assignment	Request for an assignment to be made	

Table 6.3: Notation Part 2 of 2

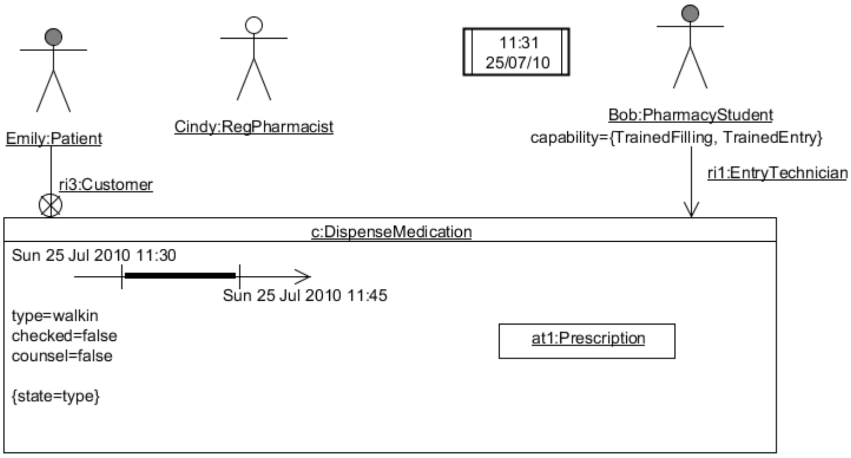


Figure 6.8: Model at M1-O0 level

6.5 Summary

This chapter introduced the first sub-language of the StADy language called StADy configuration modelling language. The configuration modelling language is composed of a generic metamodel which can be instantiated to various business processes, as illustrated in Sections 6.2-6.3. This chapter also introduced the StADy notation (Section 6.4), which is used to visually represent the business process in the StADy transformation modelling language (Chapter 7) and stage 2 of the methodology (Chapter 11) The next chapter will introduce StADy transformation modelling language, which is the second sub-language of the StADy language.

Chapter 7

Transformation Modelling Language

This chapter defines the StADy transformation modelling language, which is a sub language of the StADy language that focuses on the dynamic aspects of a graph transformation (Chapter 4) rule-based process specification to permit dynamic non-deterministic decisions. The graph transformation system (GTS) uses the StADy metamodel defined in Section 6.1 as the type graph of the GTS. The StADy transformation modelling language uses typed attributed graph transformation without inheritance (Section 4.2) because the type graph does not contain inheritance relations as discussed in Section 6.1. The state of a business process can be represented using a graph, whereas the configuration of the changing operations can be defined using graph transformation rules. The following sections discuss the graph transformation rules for the GTS. Sections 7.2 and 7.3 define predefined managerial and support GT rules, which are required for task 2.3 (Section 9.6.3) and task 2.4

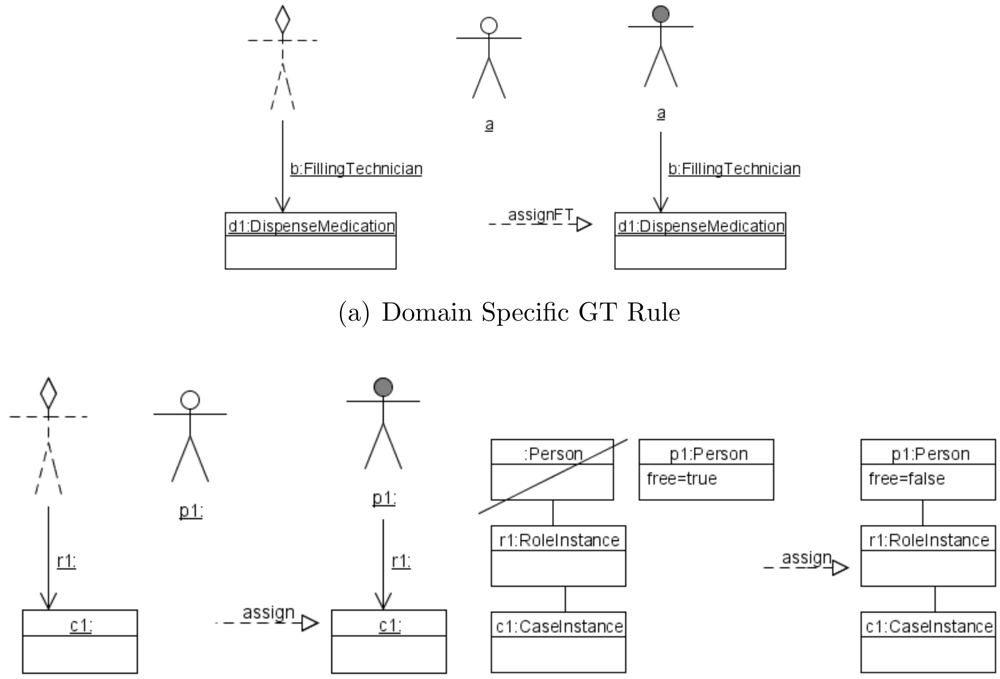
(Section 9.6.4) of the methodology. The GT rules and graphical constraints defined in Sections 7.2 and 7.3 directly correspond with the concept requirements (features) defined in Section 2.1. The selection of which features are required to be modelled in the simulation will be further discussed in task 2.1 (Section 9.6.1). All of these rules of this chapter will be equipped with probability distributions in task 2.7 (Section 9.6.7) which will control the application of the rules during stochastic simulations.

7.1 Overview

Since a business process can be subdivided into three categories: primary (production), secondary (support) and tertiary (managerial) [70] (Section 3.1), the graph transformation rules are also subdivided into these three levels. The primary rules are customer-oriented by focusing on the case, such as the *fill prescription* and *receive payment* rules. The secondary rules support the primary process by maintaining the production, such as the *escalation trigger* and *clock tick* rules, which are further discussed in Section 7.3. The managerial rules direct and coordinate the primary and secondary processes by maintaining the organisational structure in terms of required resources and process allocation, such as the *role assignment* and *role request* rule, which are further discussed in Section 7.2. Each of these rules can be defined in domain-independent or domain-specific syntax. Domain-specific actions capture domain knowledge which is usually unique to the business process, whereas domain-independent actions can be generically used by other business processes.

The rule-based form of modelling is in contrast to the static *control flow* approach, which represents precision in the expected ordering of events. The control flow approach is suitable for defining static aspects in the production level; however it is not suitable for low level dynamic managerial and support aspects such as role assignment and scheduling, because in order to capture these non-deterministic decisions the specifications would have to break down into various complex exceptions. Hence, the StADy language uses a rule-based approach in order to capture the non-deterministic nature of the managerial and support levels and encodes control flow in the graphs to capture the production-level aspects. As a result, the language contains an aspect of control to define predefined routines, yet is flexible enough to represent non-deterministic managerial decisions.

To illustrate the difference between domain-specific and domain-independent rules, consider Figure 7.1. Sub-figure 7.1(a) illustrates a domain-specific rule for assigning a free actor to a vacant role on a *DispenseMedication* case. In most cases, this assignment is independent of the type of the actor or case, as long as the former is qualified to fill the role on the latter. Therefore, abstracting from the domain-specific type, a generic (domain-independent) rule is obtained in Sub-figures 7.1(b) and 7.1(c). Sub-figure 7.1(b) defines the domain-independent rule using the StADy concrete syntax, whereas Sub Figure 7.1(c) denotes the domain-independent rule using abstract syntax.



(a) Domain Specific GT Rule

(b) Generic GT Rule with Concrete Syntax

(c) Generic GT Rule with Abstract Syntax

Figure 7.1: GT Rule with Graphical Consistency Constraints Defined in Figures 7.10, 7.11 and 7.12

Figure 7.2 illustrates the *fill prescription* domain specific production-level rule which denotes that if a prescription is not filled and there is a filling technician assigned to the case instance, then the fill prescription action can be performed. Figure 7.3 illustrates the mapping to *E-graph* elements (Definition 4.2.1) in graphs L R in a graph transformation rule (Definition 4.1.3).

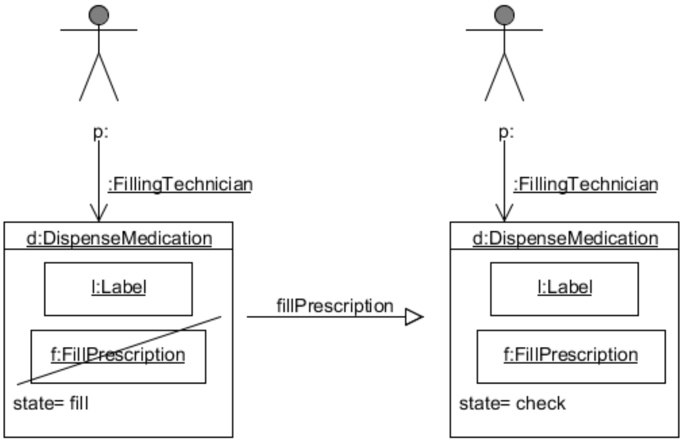


Figure 7.2: Domain-specific Business Level GT Rule

GT Rule: Fill prescription
Pre-Condition: A prescription is not filled and there is a filling technician assigned to the case instance.
Post-Condition: The prescription is filled.

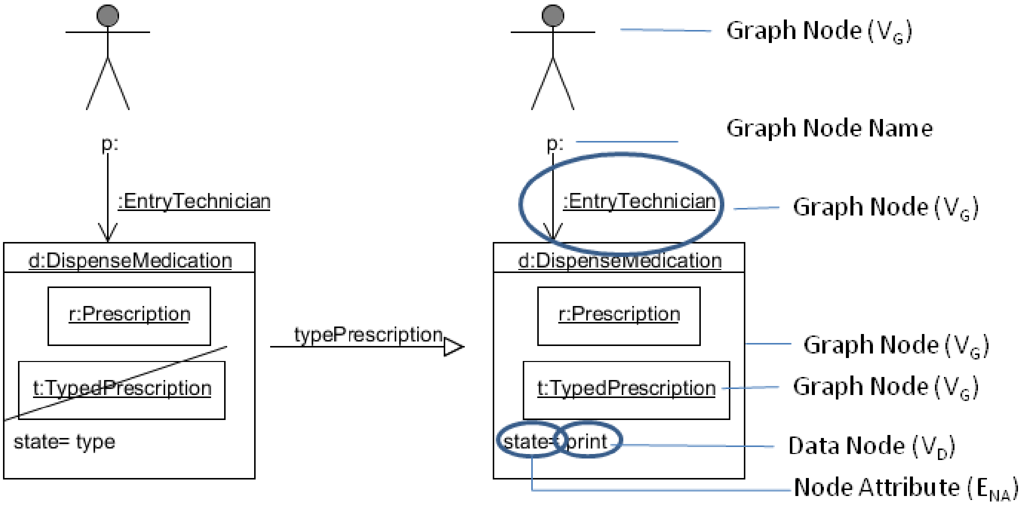


Figure 7.3: GT Rule mapped to E-graph

Other domain specific rules which correspond to the pharmacy case study in Chapter 5 include: *type prescription*, *print label*, and *receive payment.* The state diagram [24] of a pharmacy business process is shown in Figure 7.4. Each state represents the stage of that particular case in the process. Each prescription *case* contains a state label, which corresponds to the task that needs to be performed, with the assumption that upon completion of each task the state label should correctly represent the state transitions. State labels are also used to provide an external view of the case without immediately checking internal contents. However, this assumption can sometimes result in errors if the case is labelled incorrectly. Therefore, validation is required and sometimes results in backtracking transitions, since people do not always perform prescribed actions, *skip* rules are used to represent when an employee skipped a task. For example, an employee might decide not to fill a prescription and to pass it on to the next task, this can be represented in a case by changing the state label of a case without changing the case's contents. With the assumption that the state label is correct, request rule would be used to request a pharmacist to be assigned to the case without checking the internal contents. After the pharmacist is assigned, then the content is verified through pattern matching. If the case is missing a required *artifact* element or contains an error, then a *backtrack* rule would revert the case to its previous state. *Skip* and *backtrack* GT rules are further discussed in Section 7.3.4.

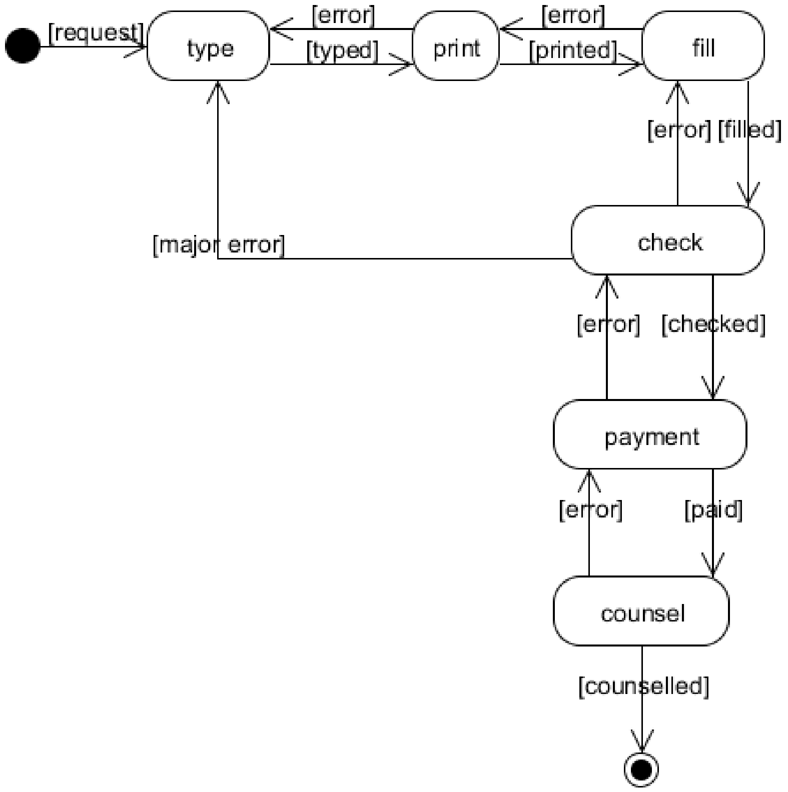


Figure 7.4: State Diagram of Pharmacy Business Process

A stochastic graph transformation system [68] associates probability distributions to rule using general distributions. The general distributions include normal and exponential distributions. Section 4.3 discusses how to assign probability distributions to rule pairs. The distribution parameter can be visually denoted on the transition arrow as shown in Figure 7.5. Attachment of distributions to rules is further discussed and illustrated in Sections 9.6.7 and 11.7, respectively.

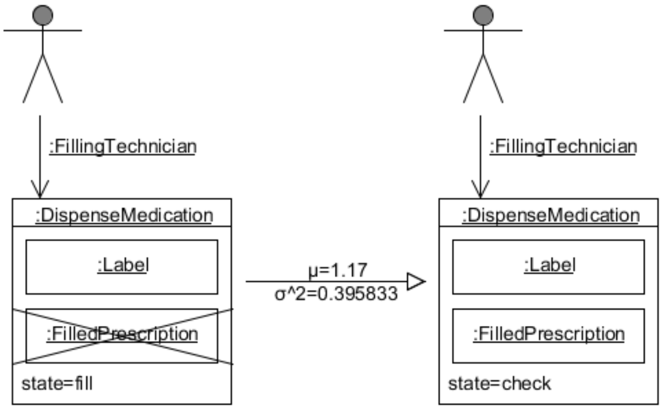


Figure 7.5: Distributed Event for Fill Prescription GT Rule

7.2 Managerial-Level Rules

The following section illustrates and discusses the managerial GT rules in terms of the features defined in Figure 2.2.

7.2.1 Dynamic (Re)-assignment (R1.1)

The dynamic (re)-assignment feature requires request, assignment and unassignment rules. The request rule uses local state labels to determine if an actor is required for a particular case (Figure 7.6). The assignment rule uses the post condition of request rules to assign actors to roles (Figure 7.7). Similarly, the unassignment (Figure 7.8) rule uses the state label to determine if the actor has finished performing their duties.

Request Rule

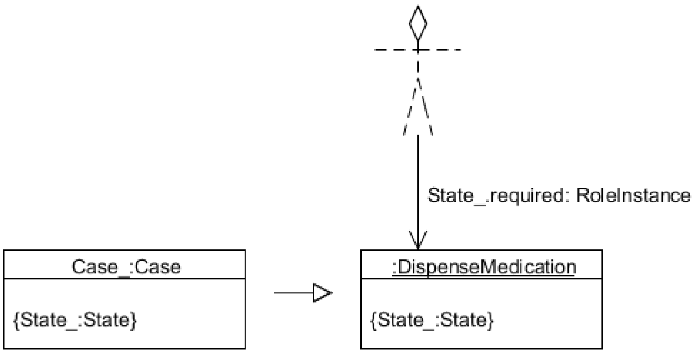


Figure 7.6: Request Rule

Assignment Rule

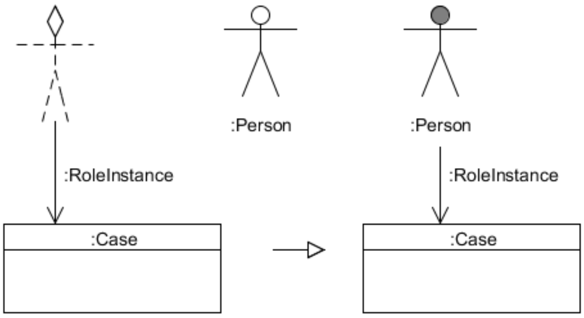


Figure 7.7: Assignment Rule

Unassign Rule

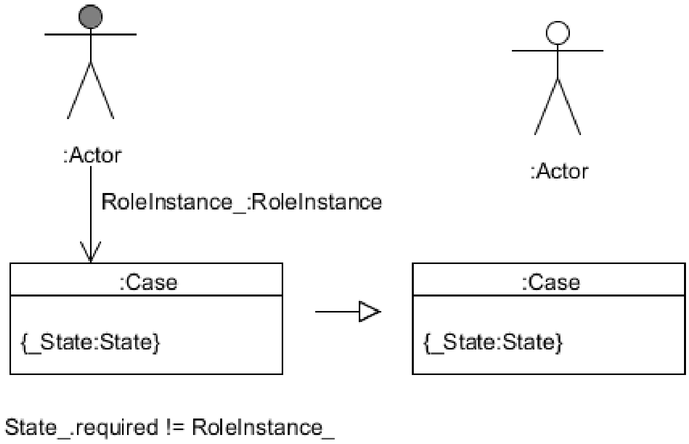


Figure 7.8: Unassignment Rule

Assignment Policy

The assignment policy feature is incorporated into assignment transformation rules. The assignment policy specifies that the least qualified available worker able to do the job, should be assigned. Figure 7.9, states that a registered pharmacist should only be assigned to the filling technician role, if all technicians and trained pharmacy students are not available. Figure 7.10 defines a generic graphical constraint (Section 4.2) stating that the least qualified available actor who is capable of performing the role, is assigned to it.

Strategies for resource usage are usually domain-independent, although they may vary with the type of organisation (small business, public service, large company). Therefore role assignment and scheduling aspects are chosen to be defined independent of the domain concepts, i.e., as direct instances of the metamodel but not instantiating the M1 level diagrams by using a combination of rules and graphical constraints. Operationally, rule application is prohibited if the constraints are not satisfied by the derived graph, so constraints act as global right-sided application conditions. It greatly simplifies the specification of rules, such as the assignment rules shown in Figure 7.1.

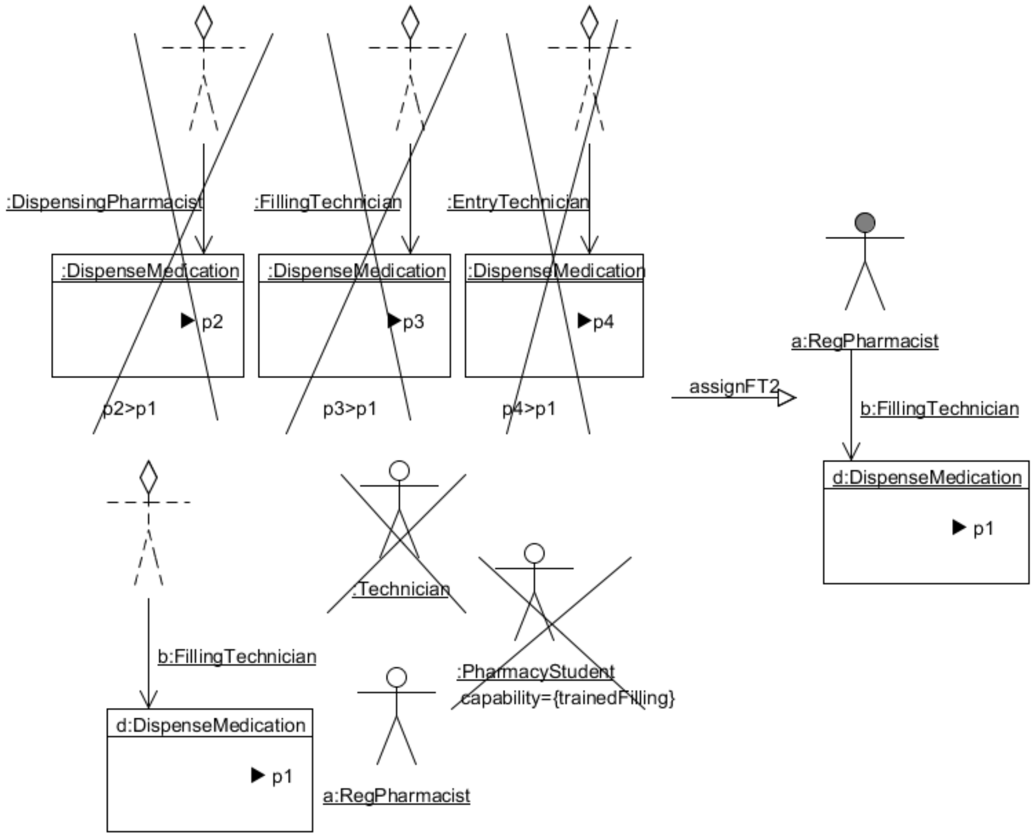


Figure 7.9: Defining an Assignment Policy using Negative Application Conditions

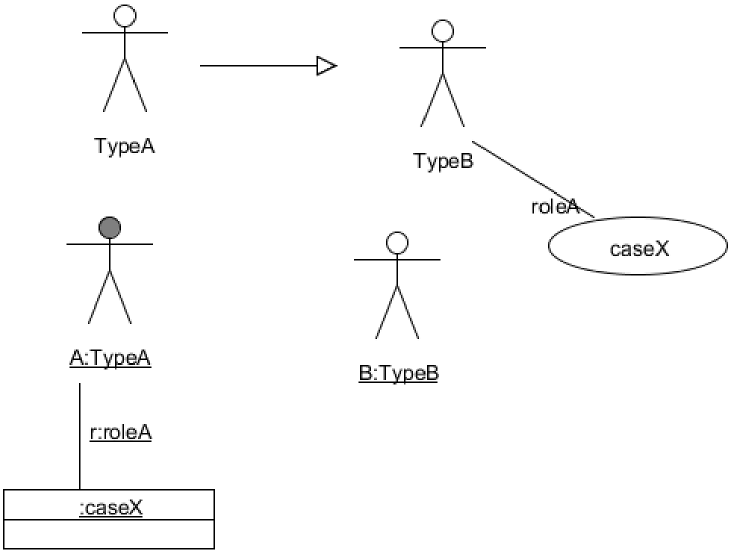


Figure 7.10: Defining Assignment Policy using Graphical Constraint

7.2.2 Scheduling (R2.1)

Scheduling policies that were introduced in Section 3.1 provide the option to schedule according various parameters such as deadlines or priorities. Hence, order of assignment to the case can be based on an earlier deadline or a higher priority constraint. Figure 7.9 uses negative application conditions to define a transformation rule for assigning a filling technician to a case with the highest priority with the assignment policy described below. Figures 7.11 and 7.12 define graphical constraint that state that there should never be a role assigned to a case if there is a required assignment with an earlier deadline or higher priority. Both of these graphical constraints use arithmetic inequalities (Definition ??) to compare cases deadline or priority attribute values, in order to ensure that the case with the closer deadline or higher priority will be scheduled earlier. The scheduling policy is in no relation to the scheduling in the stochastic graph transformation system simulation that was defined in Section 4.3.

Traditionally, application conditions are used to define policies by restricting the applicability of individual rules with complex negative conditions. Figure 7.9 demonstrates this traditional approach of defining an assignment GT rule without defining graphical consistency constraints. This construction of application conditions from constraints has been developed in [31].

Scheduling by Deadline

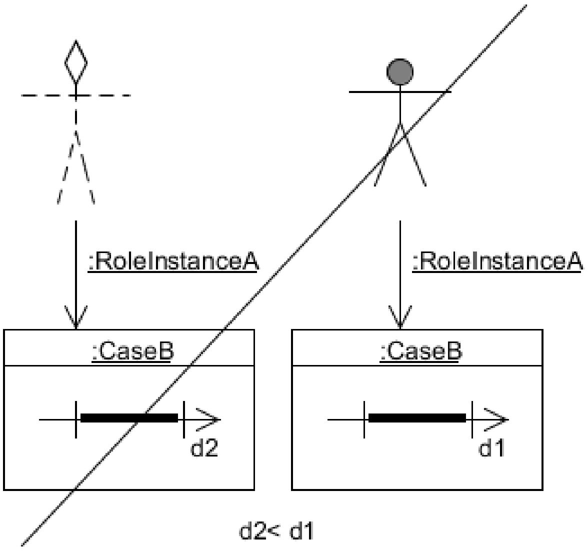


Figure 7.11: Scheduling by Deadline Graphical Constraint

Scheduling by Priority

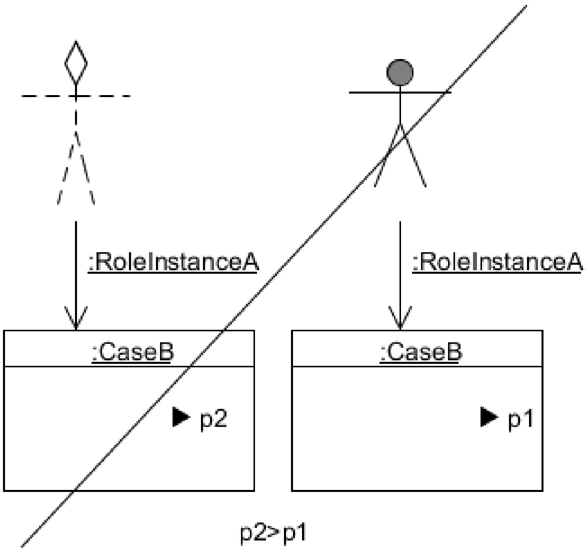


Figure 7.12: Scheduling by Priority Graphical Constraint

Clock

The model's local clock is incremented with a *clock tick* rule used to provide a model of time passing as required, for example, for checking deadlines. The rules can access the current time through the time attribute in the Clock node to compute deadlines or validate conditions to trigger an escalation; hence if a case deadline has passed, the escalation level of the case should be increased. The attachment of a normal probability distribution value will be used to control the application of this rule, which will be further discussed in Section 9.6.7.

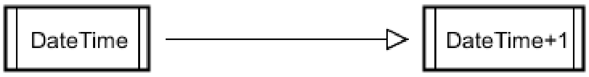


Figure 7.13: Clock Tick Rule

7.3 Support-Level Rules

The following support level rules are defined using domain specific notation.

7.3.1 Escalation Handling (R2.3)

The escalation feature requires trigger and temp assignment transformation rules. The trigger rules are used when a particular case has gone into a escalated state, such as approaching a deadline (Figure 7.14). Temp assignment rules occur only at escalated states by temporarily permitting a person to take on a role that they are not normally allowed, as shown in Figure 7.15.

Trigger Rule

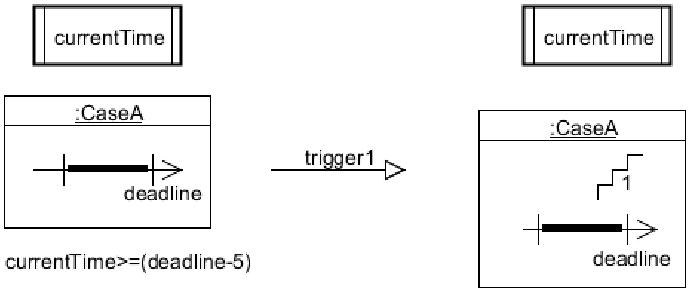


Figure 7.14: Trigger to Escalation Level 1

7.3.2 Role Promotion/Demotion and Temporary Promotion (R1.2)

Temp assignment rules occur only at escalated states by temporarily permitting a person to take on a role that he or she is not normally allowed to perform.

Temp Assign Rule

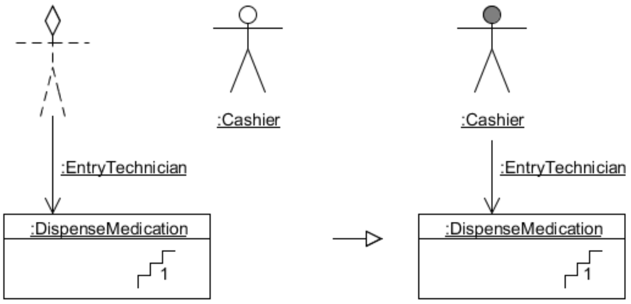


Figure 7.15: Temp Assign Entry Technician

7.3.3 Load Balancing (R2.4)

The load balancing feature uses a transfer transformation rule to distribute a prescription to another store. The transfer rule only transfers a prescription if it is for delivery, as shown in Figure 7.16.

Transfer

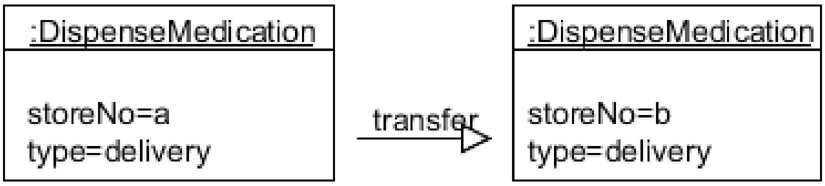


Figure 7.16: Load Balancing Rule: Transfer Prescription to Another Store

7.3.4 Human Error (R2.5)

Skip and backtrack rules are used to represent the human error feature. Since people do not always perform actions, *skip* rules are used to represent when an employee skipped a task. The skip rule changes the case state label without changing the case artifacts whereas the backtrack rule checks that the state label is consistent with the case artifacts. An example of a skip rule is when an employee decides not to fill a prescription and passes it on to the next task, as shown in Figure 7.17. If the case is missing an element, then it is backtracked into the previous state, as shown in Figure 7.18.

Skip Rule

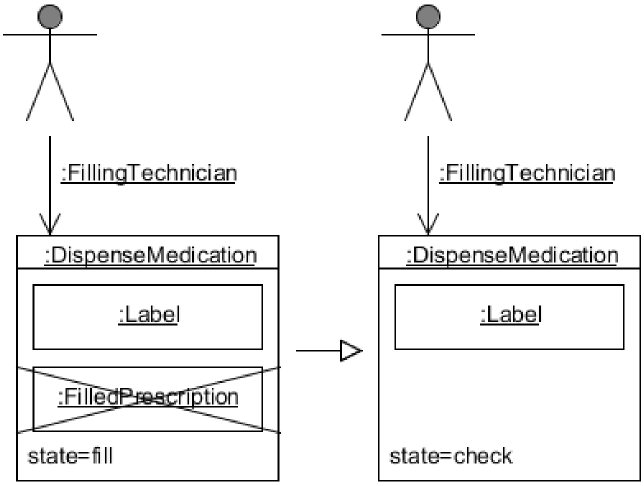


Figure 7.17: Skip Fill Prescription

Backtrack Rule

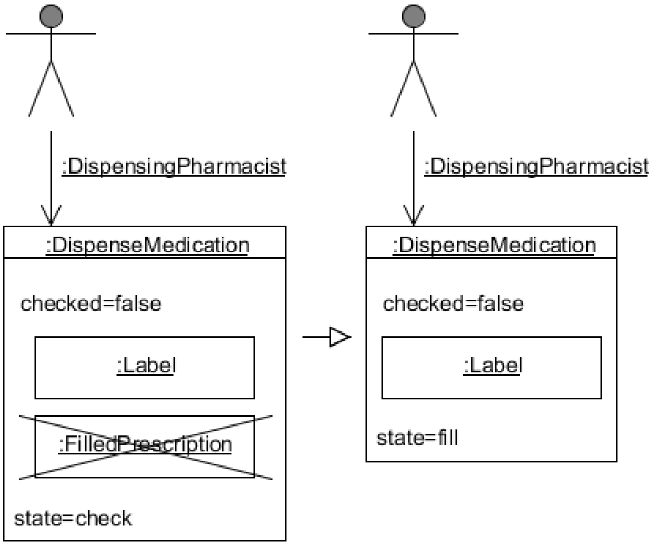


Figure 7.18: Backtrack Check State

7.4 Summary

This chapter introduced the StADy transformation modelling language, which is a sub language of the StADy language. The transformation modelling language uses the configuration modelling language (Chapter 6) to define its graph transformation system. The GT rules and graphical constraints defined in this chapter are required for tasks 2.3-2.4 (Sections 9.6.3-9.6.4) of the methodology and correspond to the features defined in the concept requirements (Section 2.1). The next chapter will define a mapping from the StADy language syntax to VIATRA2 syntax in order to perform simulation using Graph-based Stochastic Simulation (GrASS) [30, 68] in task 4.1 (Section 9.8.1).

Chapter 8

Translation of Design into Simulation

This chapter describes how to translate StADy designs into VIATRA2 [72] syntax for Graph-based Stochastic Simulation (GraSS) [30, 68] implementation. The encoding of StADy into VIATRA2 syntax is required in stage 3 (Chapter 12) of the methodology. Stage 3 encodes the existing StADy metamodel (Section 6.1), models and GT rules into the VIATRA2 tool, in order to perform stochastic simulation on the GraSS tool [30, 68] which extends VIATRA2.

The metamodel and models are defined in VIATRA2 Model Space [72] using the tools model space tree editor and the Visual and Precise Metamodelling (VPM) [71] metamodel in Figure 8.1 which are saved in a VPML file. The VPM metamodel specifies how model element entities and relations are defined in VPM.

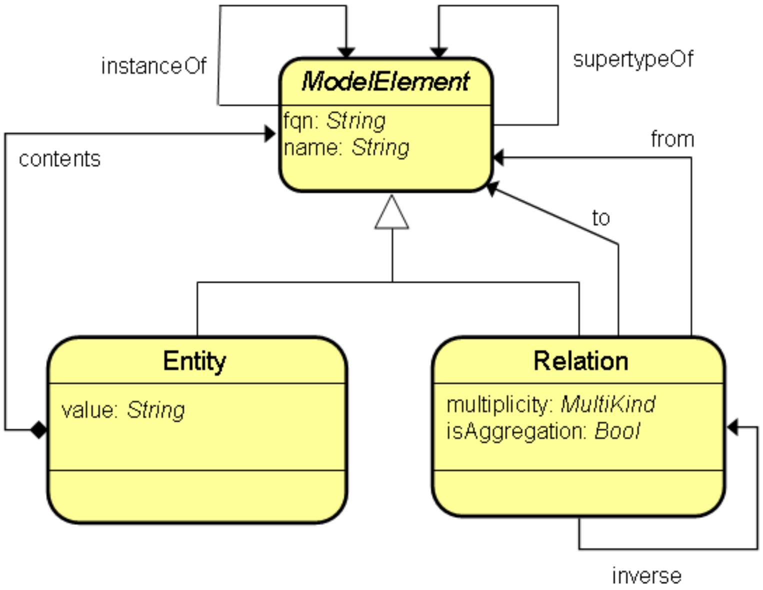
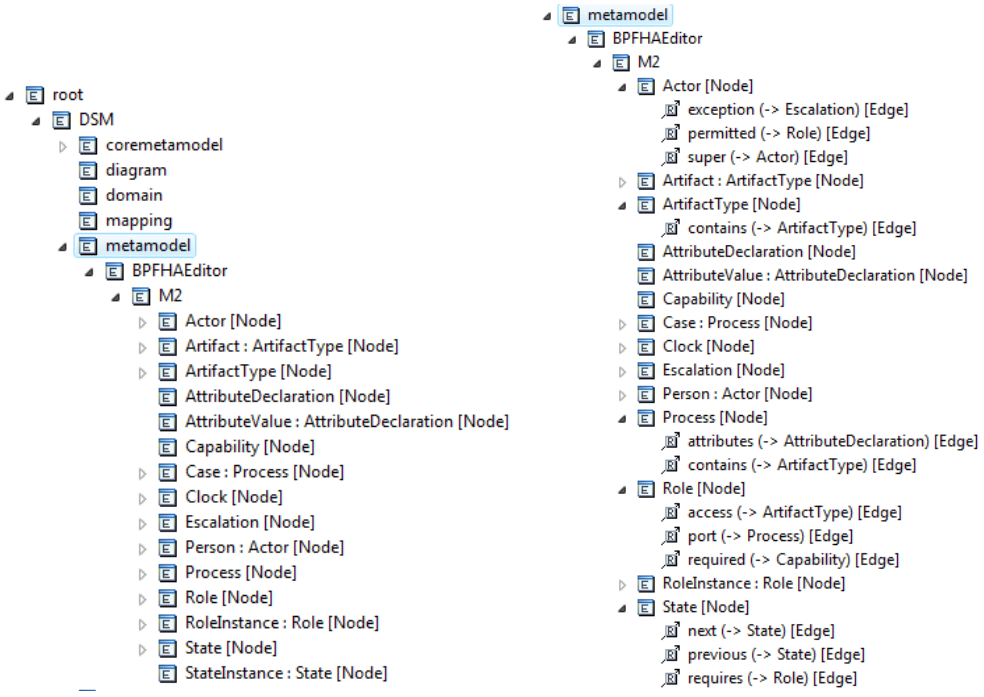


Figure 8.1: VPM metamodel [72]

Section 8.1 defines the StADy metamodel in the VIATRA2 model space, whereas Section 8.2 illustrates the model defined in Section 6.2. VIATRA2 textual syntax is required for defining GT rules and probes, which are saved in a VTML file. The mappings of StADy graphical notation and GT rules to VIATRA2 textual syntax are illustrated in Section 8.3.

8.1 Metamodel in VPM

This section illustrates how the StADy metamodel (Figure 6.2) is represented in the VIATRA2 (VPM) Model Space [72]. Figure 8.2(a) presents an overview of the type and instance level features. The relational and sub entity details are presented in Figures 8.2(b) and 8.3.



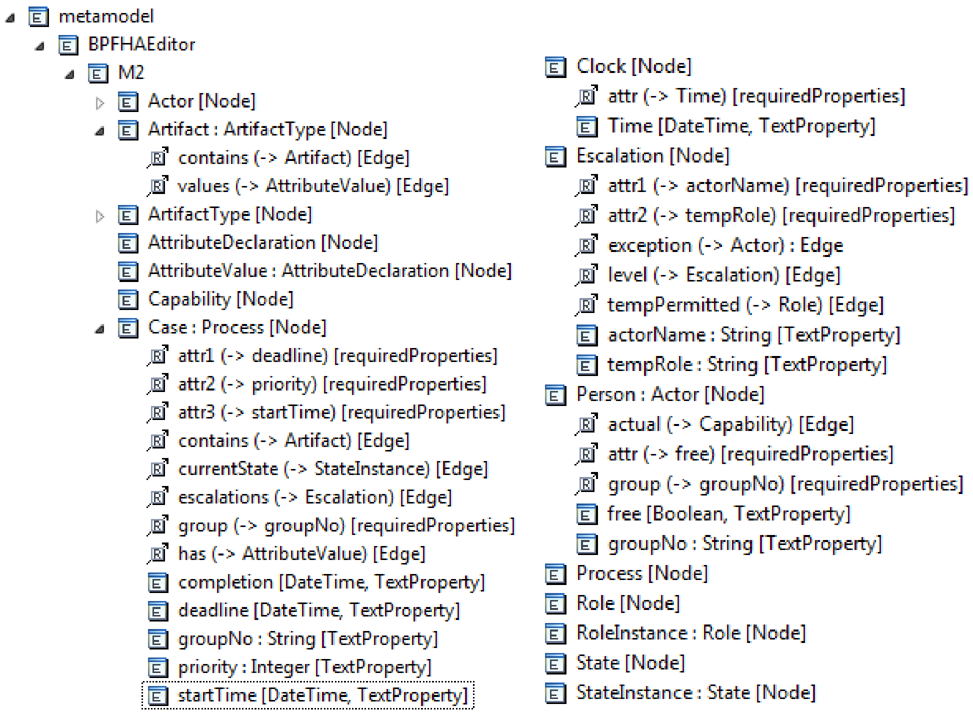
```
root
  DSM
    coremetamodel
    diagram
    domain
    mapping
    metamodel
      BPFHAEditor
        M2
          Actor [Node]
          Artifact : ArtifactType [Node]
          ArtifactType [Node]
          AttributeDeclaration [Node]
          AttributeValue : AttributeDeclaration [Node]
          Capability [Node]
          Case : Process [Node]
          Clock [Node]
          Escalation [Node]
          Person : Actor [Node]
          Process [Node]
          Role [Node]
          RoleInstance : Role [Node]
          State [Node]
          StateInstance : State [Node]
```

(a) Overview

```
metamodel
  BPFHAEditor
    M2
      Actor [Node]
        exception (-> Escalation) [Edge]
        permitted (-> Role) [Edge]
        super (-> Actor) [Edge]
      Artifact : ArtifactType [Node]
      ArtifactType [Node]
        contains (-> ArtifactType) [Edge]
      AttributeDeclaration [Node]
      AttributeValue : AttributeDeclaration [Node]
      Capability [Node]
      Case : Process [Node]
      Clock [Node]
      Escalation [Node]
      Person : Actor [Node]
      Process [Node]
        attributes (-> AttributeDeclaration) [Edge]
        contains (-> ArtifactType) [Edge]
      Role [Node]
        access (-> ArtifactType) [Edge]
        port (-> Process) [Edge]
        required (-> Capability) [Edge]
      RoleInstance : Role [Node]
      State [Node]
        next (-> State) [Edge]
        previous (-> State) [Edge]
        requires (-> Role) [Edge]
```

(b) O1 (Type-Level) Details

Figure 8.2: Metamodel



```
metamodel
  BPFHAEditor
    M2
      Actor [Node]
      Artifact : ArtifactType [Node]
        contains (-> Artifact) [Edge]
        values (-> AttributeValue) [Edge]
      ArtifactType [Node]
      AttributeDeclaration [Node]
      AttributeValue : AttributeDeclaration [Node]
      Capability [Node]
      Case : Process [Node]
        attr1 (-> deadline) [requiredProperties]
        attr2 (-> priority) [requiredProperties]
        attr3 (-> startTime) [requiredProperties]
        contains (-> Artifact) [Edge]
        currentState (-> StateInstance) [Edge]
        escalations (-> Escalation) [Edge]
        group (-> groupNo) [requiredProperties]
        has (-> AttributeValue) [Edge]
        completion [DateTime, TextProperty]
        deadline [DateTime, TextProperty]
        groupNo : String [TextProperty]
        priority : Integer [TextProperty]
        startTime [DateTime, TextProperty]
```

(a) O0 part1

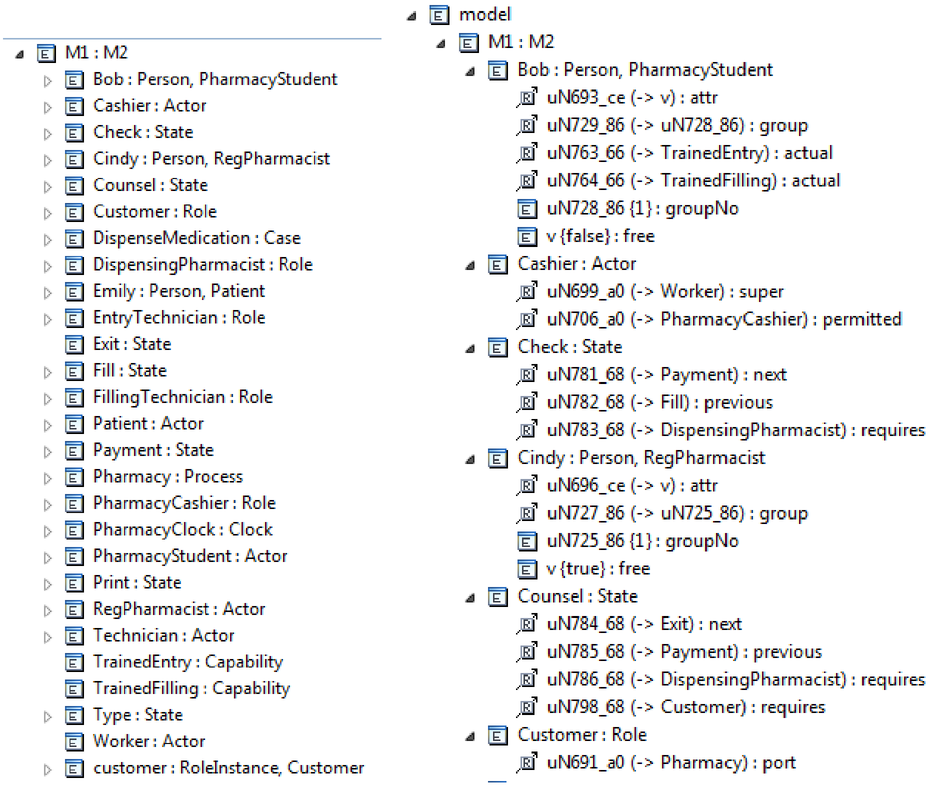
```
Clock [Node]
  attr (-> Time) [requiredProperties]
  Time [DateTime, TextProperty]
Escalation [Node]
  attr1 (-> actorName) [requiredProperties]
  attr2 (-> tempRole) [requiredProperties]
  exception (-> Actor) : Edge
  level (-> Escalation) [Edge]
  tempPermitted (-> Role) [Edge]
  actorName : String [TextProperty]
  tempRole : String [TextProperty]
Person : Actor [Node]
  actual (-> Capability) [Edge]
  attr (-> free) [requiredProperties]
  group (-> groupNo) [requiredProperties]
  free [Boolean, TextProperty]
  groupNo : String [TextProperty]
Process [Node]
Role [Node]
RoleInstance : Role [Node]
State [Node]
StateInstance : State [Node]
```

(b) O0 part2

Figure 8.3: Metamodel O0 (Object-Level) Details

8.2 Model in VPM

This section illustrates the mapping of the M1 abstract model in Figure 6.4 represented in VIATRA2 model space. Sub-figure 8.4(a) represents an overview of the model, whereas Sub-figures 8.4(b)-8.5(b) presents the relation and entity details of the model.



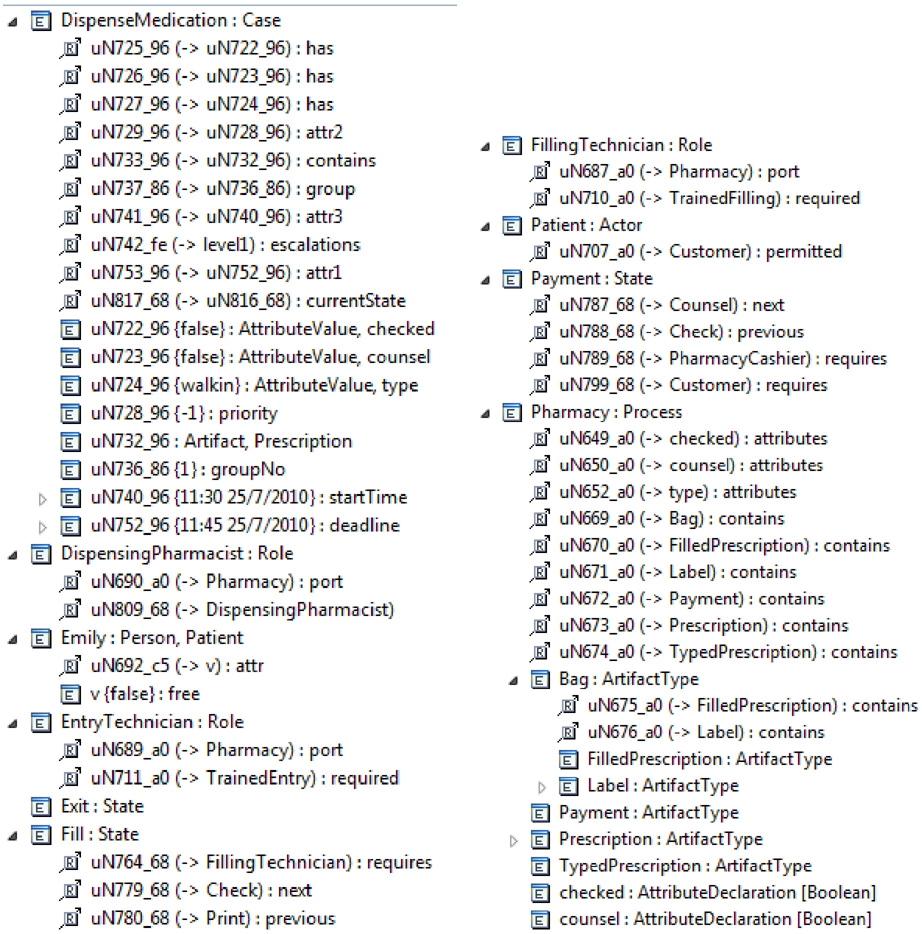
```
◢ M1 : M2
    ▷ Bob : Person, PharmacyStudent
    ▷ Cashier : Actor
    ▷ Check : State
    ▷ Cindy : Person, RegPharmacist
    ▷ Counsel : State
    ▷ Customer : Role
    ▷ DispenseMedication : Case
    ▷ DispensingPharmacist : Role
    ▷ Emily : Person, Patient
    ▷ EntryTechnician : Role
      Exit : State
    ▷ Fill : State
    ▷ FillingTechnician : Role
    ▷ Patient : Actor
    ▷ Payment : State
    ▷ Pharmacy : Process
    ▷ PharmacyCashier : Role
    ▷ PharmacyClock : Clock
    ▷ PharmacyStudent : Actor
    ▷ Print : State
    ▷ RegPharmacist : Actor
    ▷ Technician : Actor
      TrainedEntry : Capability
      TrainedFilling : Capability
    ▷ Type : State
      Worker : Actor
    ▷ customer : RoleInstance, Customer
```

(a) Overview

```
◢ model
  ◢ M1 : M2
    ◢ Bob : Person, PharmacyStudent
        uN693_ce (-> v) : attr
        uN729_86 (-> uN728_86) : group
        uN763_66 (-> TrainedEntry) : actual
        uN764_66 (-> TrainedFilling) : actual
        uN728_86 {1} : groupNo
        v {false} : free
    ◢ Cashier : Actor
        uN699_a0 (-> Worker) : super
        uN706_a0 (-> PharmacyCashier) : permitted
    ◢ Check : State
        uN781_68 (-> Payment) : next
        uN782_68 (-> Fill) : previous
        uN783_68 (-> DispensingPharmacist) : requires
    ◢ Cindy : Person, RegPharmacist
        uN696_ce (-> v) : attr
        uN727_86 (-> uN725_86) : group
        uN725_86 {1} : groupNo
        v {true} : free
    ◢ Counsel : State
        uN784_68 (-> Exit) : next
        uN785_68 (-> Payment) : previous
        uN786_68 (-> DispensingPharmacist) : requires
        uN798_68 (-> Customer) : requires
    ◢ Customer : Role
        uN691_a0 (-> Pharmacy) : port
```

(b) Part 1

Figure 8.4: Model in VIATRA2 Model Space 1-3



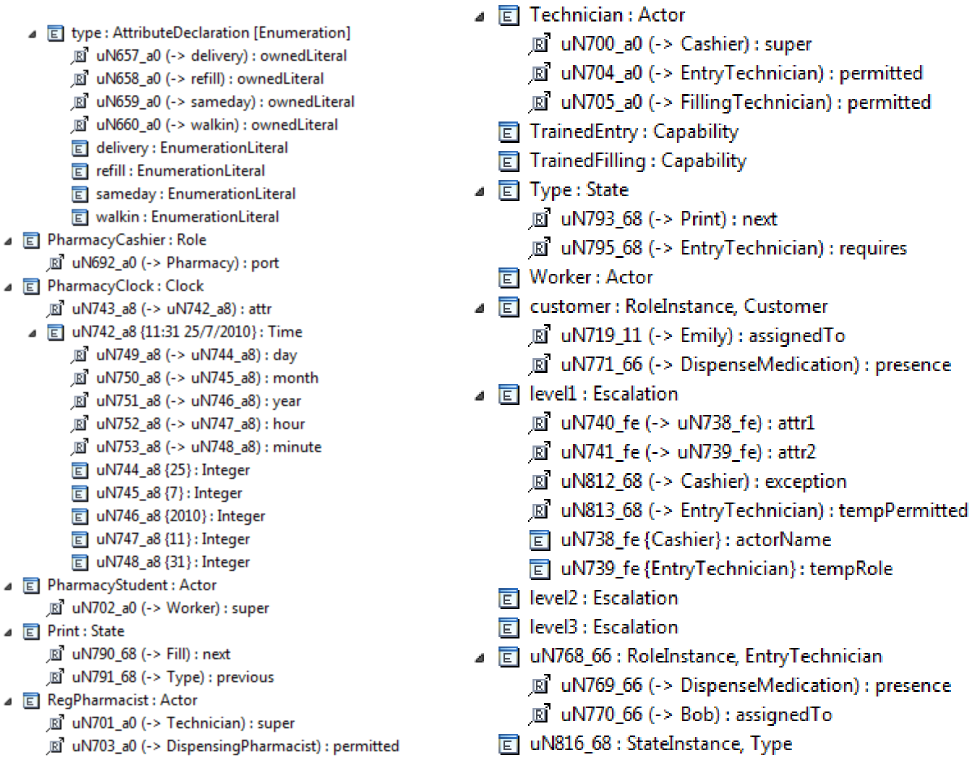
```
DispenseMedication : Case
    uN725_96 (-> uN722_96) : has
    uN726_96 (-> uN723_96) : has
    uN727_96 (-> uN724_96) : has
    uN729_96 (-> uN728_96) : attr2
    uN733_96 (-> uN732_96) : contains
    uN737_86 (-> uN736_86) : group
    uN741_96 (-> uN740_96) : attr3
    uN742_fe (-> level1) : escalations
    uN753_96 (-> uN752_96) : attr1
    uN817_68 (-> uN816_68) : currentState
    uN722_96 {false} : AttributeValue, checked
    uN723_96 {false} : AttributeValue, counsel
    uN724_96 {walkin} : AttributeValue, type
    uN728_96 {-1} : priority
    uN732_96 : Artifact, Prescription
    uN736_86 {1} : groupNo
    uN740_96 {11:30 25/7/2010} : startTime
    uN752_96 {11:45 25/7/2010} : deadline
DispensingPharmacist : Role
    uN690_a0 (-> Pharmacy) : port
    uN809_68 (-> DispensingPharmacist)
Emily : Person, Patient
    uN692_c5 (-> v) : attr
    v {false} : free
EntryTechnician : Role
    uN689_a0 (-> Pharmacy) : port
    uN711_a0 (-> TrainedEntry) : required
Exit : State
Fill : State
    uN764_68 (-> FillingTechnician) : requires
    uN779_68 (-> Check) : next
    uN780_68 (-> Print) : previous
```

(a) Part 2

```
FillingTechnician : Role
    uN687_a0 (-> Pharmacy) : port
    uN710_a0 (-> TrainedFilling) : required
Patient : Actor
    uN707_a0 (-> Customer) : permitted
Payment : State
    uN787_68 (-> Counsel) : next
    uN788_68 (-> Check) : previous
    uN789_68 (-> PharmacyCashier) : requires
    uN799_68 (-> Customer) : requires
Pharmacy : Process
    uN649_a0 (-> checked) : attributes
    uN650_a0 (-> counsel) : attributes
    uN652_a0 (-> type) : attributes
    uN669_a0 (-> Bag) : contains
    uN670_a0 (-> FilledPrescription) : contains
    uN671_a0 (-> Label) : contains
    uN672_a0 (-> Payment) : contains
    uN673_a0 (-> Prescription) : contains
    uN674_a0 (-> TypedPrescription) : contains
    Bag : ArtifactType
        uN675_a0 (-> FilledPrescription) : contains
        uN676_a0 (-> Label) : contains
        FilledPrescription : ArtifactType
        Label : ArtifactType
    Payment : ArtifactType
    Prescription : ArtifactType
    TypedPrescription : ArtifactType
    checked : AttributeDeclaration [Boolean]
    counsel : AttributeDeclaration [Boolean]
```

(b) Part 3

Figure 8.5: Model in VIATRA2 Model Space 2-3



```
      type : AttributeDeclaration [Enumeration]
          uN657_a0 (-> delivery) : ownedLiteral
          uN658_a0 (-> refill) : ownedLiteral
          uN659_a0 (-> sameday) : ownedLiteral
          uN660_a0 (-> walkin) : ownedLiteral
          delivery : EnumerationLiteral
          refill : EnumerationLiteral
          sameday : EnumerationLiteral
          walkin : EnumerationLiteral
  PharmacyCashier : Role
      uN692_a0 (-> Pharmacy) : port
  PharmacyClock : Clock
      uN743_a8 (-> uN742_a8) : attr
      uN742_a8 {11:31 25/7/2010} : Time
          uN749_a8 (-> uN744_a8) : day
          uN750_a8 (-> uN745_a8) : month
          uN751_a8 (-> uN746_a8) : year
          uN752_a8 (-> uN747_a8) : hour
          uN753_a8 (-> uN748_a8) : minute
          uN744_a8 {25} : Integer
          uN745_a8 {7} : Integer
          uN746_a8 {2010} : Integer
          uN747_a8 {11} : Integer
          uN748_a8 {31} : Integer
  PharmacyStudent : Actor
      uN702_a0 (-> Worker) : super
  Print : State
      uN790_68 (-> Fill) : next
      uN791_68 (-> Type) : previous
  RegPharmacist : Actor
      uN701_a0 (-> Technician) : super
      uN703_a0 (-> DispensingPharmacist) : permitted
```

(a) Part 4

```
  Technician : Actor
      uN700_a0 (-> Cashier) : super
      uN704_a0 (-> EntryTechnician) : permitted
      uN705_a0 (-> FillingTechnician) : permitted
  TrainedEntry : Capability
  TrainedFilling : Capability
  Type : State
      uN793_68 (-> Print) : next
      uN795_68 (-> EntryTechnician) : requires
  Worker : Actor
  customer : RoleInstance, Customer
      uN719_11 (-> Emily) : assignedTo
      uN771_66 (-> DispenseMedication) : presence
  level1 : Escalation
      uN740_fe (-> uN738_fe) : attr1
      uN741_fe (-> uN739_fe) : attr2
      uN812_68 (-> Cashier) : exception
      uN813_68 (-> EntryTechnician) : tempPermitted
      uN738_fe {Cashier} : actorName
      uN739_fe {EntryTechnician} : tempRole
  level2 : Escalation
  level3 : Escalation
  uN768_66 : RoleInstance, EntryTechnician
      uN769_66 (-> DispenseMedication) : presence
      uN770_66 (-> Bob) : assignedTo
  uN816_68 : StateInstance, Type
```

(b) Part 5

Figure 8.6: Model in VIATRA2 Model Space 3-3

8.3 VIATRA2 Textual Syntax

This section presents the mappings of StADy modelling notation to VIATRA2 textual syntax and illustrates it in production-level, managerial-level and support-level GT Rules. Details of the VIATRA2 Textual Command Language (VTCL) can be found at [43, 72, 44]. This manual process is required in task 3.3 (Section 9.7.3) of the methodology for encoding the business processes domain-specific rules from StADy graphical notation into VIATRA2 textual syntax. The VIATRA2 textual syntax representation of the GT rules is required in order to perform stochastic simulation on GraSS tool [30, 68] which extends VIATRA2. The VIATRA2 representation of the generically defined managerial-level GT rules (Section 7.2) are available in Appendix A.1. VIATRA2 textual syntax illustrations of partial domain-specific support-level GT rules (Section 7.3) are available in Appendix A.2, whereas domain-specific production-level GT rules are available in Appendix A.3. The managerial and support-level translation design choices are presented in this section, whereas the production-level translation design choices are only illustrated in this section. Task 3.3 (Section 9.7.3) of the StADy methodology provides the developer with translation design choices for their production-level GT rules that were defined in task 2.4 (Section 9.6.4). Each of the GT rules that were translated from StADy designs into VIATRA2 syntax was individually tested, by applying the rule on the VIATRA2 transformation engine and manually verifying that the resulting model transformed as expected.

8.3.1 Modelling Notation

This section presents a mapping of the defined graphical StADy graphical notation (Section 6.4) to VIATRA2 textual syntax, in Tables 8.1 and 8.2. This syntax is used to define the GT rules in Section 8.3.2 to 8.3.4.

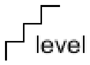
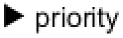
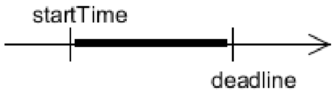
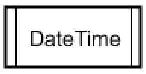
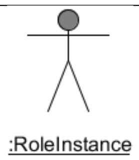
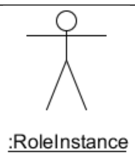
StADy	VIATRA2
	$Escalation(Level\_);$ $Case.escalations(R1\_, Case\_, Level\_);$
	$Case.priority(Priority\_);$ $Case.attr2(R1, Case\_, Priority\_);$
	$Case.deadline(Deadline\_);$ $Case.attr1(R1, Case\_, Deadline\_);$ $Case.startTime(StartTime\_);$ $Case.attr3(R1, Case\_, StartTime\_);$
	$Clock(Clock\_); Clock.Time(Time\_);$ $Clock.attr(R1, Clock\_, Time\_);$
	$Person(Person\_); Person.attr(R2\_, Person\_, Free\_);$ $Person.free(Free\_);$ $check(toBoolean(value(Free\_)) == false)$
	$Person(Person_{)}; Person.attr(R2_{,}Person_{,}Free_{)};$ $Person.free(Free\_);$ $check(toBoolean(value(Free\_)) == true)$

Table 8.1: Notation-VIATRA2 Part 1 of 2

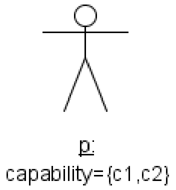
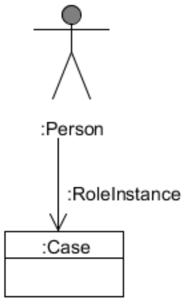
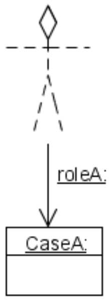
	$Capability(Capability\_); Person(Person\_);$ $Person.actual(R1\_, Person\_, Capability\_)$
{state=value}	$State(State\_); StateInstance(StateInstance\_);$ $Case.currentState(R6, Case, StateInstance\_);$ $typeOf(State\_, StateInstance\_);$
	$RoleInstance(RoleInstance\_); Role(Role\_);$ $typeOf(Role\_, RoleInstance\_);$ $Role.access(R1, Role\_, ArtifactType\_);$
	$RoleInstance(RoleInstance\_)$ $neg\ find\ isAssign(RoleInstance\_);$ $pattern\ isAssign(RoleInstance\_) = \{$ $RoleInstance(RoleInstance\_);$ $Person(Person\_);$ $Case(Case\_);$ $RoleInstance.presence(R2, RoleInstance\_, Case\_);$ $RoleInstance.assignedTo(R1, RoleInstance\_, Person\_);$ $\}$

Table 8.2: Notation-VIATRA2 Part 2 of 2

8.3.2 Production-Level GT Rule in VIATRA2

The production-level GT rules are required to be encoded into VIATRA2 [72] textual syntax in Section 9.7.3 of the methodology. Therefore, this section illustrates the mapping of the *fill prescription rule* (Figure 8.7), which was described in Section 7.1 into VIATRA2's textual syntax.

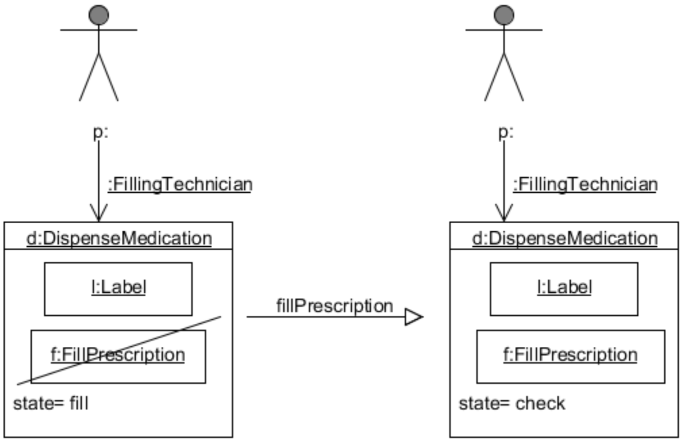


Figure 8.7: Domain-specific Business Level GT Rule
GT Rule: Fill prescription
Pre-Condition: A prescription is not filled and there is a filling technician assigned to the case instance.
Post-Condition: The prescription is filled.

The *Rule_FillPrescription* GT rule, uses various patterns including: $FTassigned(Case\_, RoleInstance\_, Role\_, Person\_)$, $FilledPrescriptionExist(Case\_)$, and $LabelExist(Case\_)$. These pattern are called from the *gtRules*, like method calls in Java. The comments within the code describe the various rule features.

```
gtrule Rule_FillPrescription() =
    {
        precondition pattern lhs(Case_,RoleInstance_,Role_,Person_,FilledPrescription_,
                                StateInstance_, NextState_,R6_) =
            {
                //Declarations i.e. Type(Instance)
                Case(Case_);
                ArtifactType(FilledPrescription_);
                State(State_);
                State(NextState_);
                StateInstance(StateInstance_);

                //Verify that ArtifactType that was matched is of FilledPrescription
                check(name(FilledPrescription_)=="FilledPrescription");

                //Verify that a Filling Technician is assigned to the Case
                find FTassigned (Case_,RoleInstance_,Role_,Person_);

                //Negative application condition
                //Ensure that the prescription is not already filled
                //i.e FilledPrescription Artifact is not contained in the Case
                neg find FilledPrescriptionExist(Case_);

                //Verify that a Label exists in the Case
                find LabelExist(Case_);

                //Retrieve Case's current State
                Case.currentState(R6_,Case_,StateInstance_);

                //Find the next state from the current state
                State.next(R7,State_,NextState_);

                //Check if current state is the required state
                typeOf(State_,StateInstance_);
                check(name(State_)=="Fill");

            }
            action {
                let
                  NewArtifact_=undef,
                  R1_=undef,
                  NewStateInstance_=undef,
                  R2_=undef

            in seq {
                //Remove Case's old state
                delete(StateInstance_);
                delete(R6_);

                //Change Case state to the next state
                // create Class inside the Case
                new(StateInstance(NewStateInstance_)in DSM.model.M1);
                //create instanceOf relation with the StateInstance:State
                new(instanceOf(NewStateInstance_,NextState_));
                new (Case.currentState(R2_,Case_,NewStateInstance_));

                //Add Filled Prescription Artifact to the Case
                // create Class inside the Case
                new(Artifact(NewArtifact_)in Case_);
                //create instanceOf relation with the Artifact:ArtifactType
                new(instanceOf(NewArtifact_,FilledPrescription_));
                //create relation between Case and Artifact
                new (Case.contains(R1_,Case_,NewArtifact_));

                println("Filled Prescription added ");
            }
                println ("Fill Prescription for "+name(Case_)+" Case");
          }
        }

    //FillingTechnician Assigned Pattern
    pattern FTassigned (Case_,RoleInstance_,Role_,Person_) =
    {
            //Declarations
            Person(Person_);
            Case(Case_);
            Role(Role_);
            RoleInstance(RoleInstance_);

            //Verify that role type is FillingTechnician
            check (name(Role_)=="FillingTechnician");

            //Verify that RoleInstance_ is of the same FillingTechnician role type
            typeOf(Role_, RoleInstance_);

            //Verify that this RoleInstance exists on the Case
            RoleInstance.presence(Rel2, RoleInstance_, Case_);

            //Verify that a person is assigned to the RoleInstance_
            //i.e a person is assigned a RoleInstance_ of FillingTechnician_
            //which is associated the Case_
            RoleInstance.assignedTo(Rel,RoleInstance_,Person_);
    }


    //Case contains an instance of FilledPrescription:ArtifactType
    pattern FilledPrescriptionExist(Case_)=
        {
            //declarations
            Artifact(Artifact_);
            ArtifactType(ArtifactType_);
            Case(Case_);

            //Verify that Case_ contains an Artifact_ of FilledPrescription type
            Case.contains(Rel, Case_, Artifact_);
            typeOf(ArtifactType_,Artifact_);
            check (name(ArtifactType_)=="FilledPrescription");
        }



        //Case contains an instance of  Label: ArtifactType
        pattern LabelExist (Case_)=
        {
            //declarations
            Artifact(Artifact_);
            ArtifactType(ArtifactType_);
            Case(Case_);

            //Verify that Case_ contains an Artifact_ of Label type
            Case.contains(Rel, Case_, Artifact_);
            typeOf(ArtifactType_,Artifact_);
            check (name(ArtifactType_)=="Label");
        }
```

8.3.3 Managerial-Level GT Rule in VIATRA2

The following section illustrates the mapping between a managerial-level rule specified in StADy and its VIATRA2 encoding. Appendix A.1 presents a catalogue of all of the managerial rules (Section 7.2) in VIATRA2 syntax. The request rule (Figure 8.8) that was described in Section 7.2.1 is textually illustrated below.

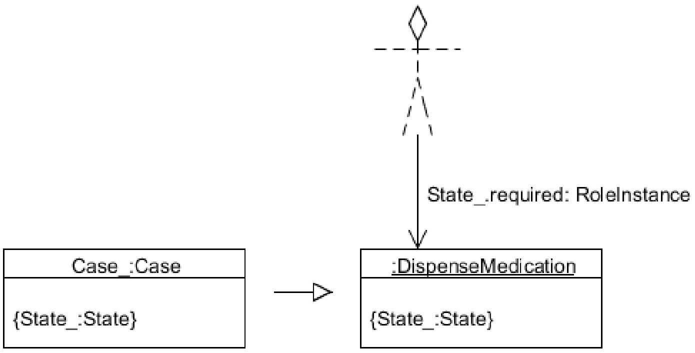


Figure 8.8: Request Person

```
gtrule Rule_Request() =
   {
          precondition pattern lhs(Case_,Role_) =
           {
               //Declaration i.e. Type(Instance)
               Case(Case_);
               Role(Role_);
               State(State_);
               StateInstance(StateInstance_);
               //Retrieve Case's current state
               Case.currentState(R6,Case_,StateInstance_);

               typeOf(State_,StateInstance_);
               //find Role that State requires
               State.requires(R5,State_,Role_);
               //ensure RoleInstance does not exist on the case
               neg find RoleInstance(Case_,Role_);
           }

           action {
               let
                     NewRoleInstance_=undef,
                     R1_=undef

               in seq {
                   //Create new RoleInstance in Model
                   new(RoleInstance(NewRoleInstance_)in DSM.model.M1);
                   //Create instanceOf Relation to Role i.e. RoleInstance:Role
                   new(instanceOf(NewRoleInstance_,Role_));
                   //create relation between RoleInstance and Case
                   new (RoleInstance.presence(R1_,NewRoleInstance_,Case_));
                   println(name(Role_)+" Requested ");
           }
       }
   }
```

8.3.4 Support-Level Rules in VIATRA2

The following section illustrates the mapping between a domain specific support-level rule specified in StADy and its VIATRA2 encoding. Appendix A.2 presents a catalogue of illustrations of the support-level rules using the pharmacy domains (Section 7.3) in VIATRA2 syntax. The skip rule (Figure 8.9), that was described in Section 7.3.4 is textually illustrated below.

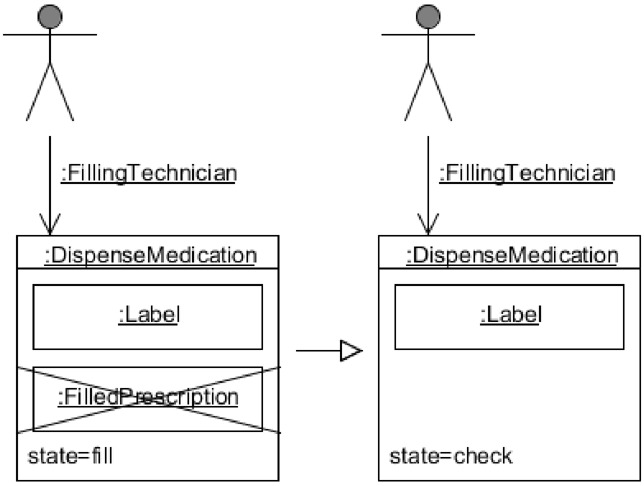


Figure 8.9: Skip Fill Prescription

```
gtrule SkipRule_FillPrescription() =
{

        precondition pattern lhs(Case_, StateInstance_, NextState_,R6_) =
        {
            //Declarations i.e. Type(Instance)
            Case(Case_);
            ArtifactType(FilledPrescription_);
            State(State_);
            State(NextState_);
            StateInstance(StateInstance_);

            //Verify that ArtifactType that was matched is of FilledPrescription
            check(name(FilledPrescription_)=="FilledPrescription");

            //Verify that Filling Technician is assigned to the case
            find FTassigned (Case_,RoleInstance_,Role_,Person_);

            //Negative application condition
            //Ensure that the prescription is not already filled
            //i.e FilledPrescription Artifact is not contained in the Case
            neg find FilledPrescriptionExist(Case_);

            //Verify that a Label exists in the Case
            find LabelExist(Case_);

            //Retrieve Case's current State
            Case.currentState(R6_,Case_,StateInstance_);

            //Find the next state from the current state
            State.next(R7,State_,NextState_);
        }
        action {
            let
                NewStateInstance_=undef,
                R2_=undef
            in seq{
                    //Remove Case's old state
                    delete(StateInstance_);
                    delete(R6_);

                    //Change Case state to the next state
                    //create Class inside the Case
                    new(StateInstance(NewStateInstance_)in DSM.model.M1);
                    //create instanceOf relation with the StateInstance:State
                    new(instanceOf(NewStateInstance_,NextState_));
                    new (Case.currentState(R2_,Case_,NewStateInstance_));

                    println("no Filled Prescription added ");
              }
       }
}
```

8.4 Summary

This chapter described how to translate StADy language syntax into VIATRA2 [72] syntax for Graph-based Stochastic Simulation (GraSS) [30, 68] implementation. The encodement of StADy into VIATRA2 syntax is relevant because it is required in stage 3 (Chapter 12) of the methodology. The next chapter will define the StADy Methodology.

Chapter 9

Methodology

This chapter discusses the methodology needed by developers to employ the thesis' new stochastic modelling and analysis of dynamic human-resource allocation (StADy) approach for testing scheduling protocols, policies and regulations. The methodology is composed of the following four stages:

1. Business modelling
2. Process execution design
3. Process encoding
4. Performance evaluation

These stages are illustrated using the pharmacy case study (Chapter 5) in Chapters 10 to 14.

9.1 Scope

It is worthwhile to use the approach if the workflow consists of human involvement with the additional properties of role hierarchies, role allocations, access rights and scheduling. The optional features specified in the feature diagram (Figure 2.2) do not have to be currently employed by the business process; however the StADy approach can be used to determine if they are beneficial to the business process prior to employing it.

There are various businesses in the scope of the StADy approach, even though it is currently limited to analysing the effect of assignment policies, load balancing, escalation handling and scheduling protocols of business processes involving humans. These business processes have common organisational characteristics of dynamic (re)-assignment, RBAC, human error, scheduling and non-deterministic duration of tasks. The StADy methodology can be applied to the pharmacy business process discussed in Chapter 5 and various other businesses such as mining, banks, retailers, home renovators and real estate businesses. The StADy methodology can be applied to the dynamic resource allocation management aspect of mortgage processing at a bank, because this process involves various human-involvement and is influenced by human decisions. For instance, if a customer urgently needs money for his or her new house and the mortgage broker that was handling the application is ill another mortgage broker can temporary work on the application. The StADy methodology can assist in simulating different protocols to handle such cases. On the other, the manufacturing business is out of scope for the StADy methodology for some domains which heavily depend on machines such as the auto industry.

In conclusion businesses with tasks that require human intelligence are in scope, where as businesses that have majority of tasks performed by computers without human interface are out of scope. With the assumption that a workflow problem has been selected within the scope, a four-stage methodology needs to be performed, which consist of business modelling, process designing, process encoding and performance evaluation stage as outlined in Section 9.3 and discussed in detail in Sections 9.5-9.8.

9.2 Preliminary Stage

Prior to employing the four stage StADy methodology, the following two preliminary tasks are required:

Task 0.1 Selection of performance questions to answer.

Task 0.2 Detailed specification of business process in case study format.

The first preliminary task is the selection of a performance question, either from the previously defined ones in Section 1.2 or from formulating a new one based on a different combination of features from the feature diagram (Figure 2.2). The mandatory features (M) include the following: dynamic (re)-assignment, RBAC, human error, scheduling, duration of tasks. These mandatory features can either be scheduled by deadline (M_D) or by priority (M_P). On the other hand, the optional features (O) include the following: assignment policy (A), load balancing (L), escalation handling (E). These optional features have 15 unique combinations=$(\{\emptyset\}, \{A\}, \{L\}, \{E\}, \{AL\}, \{AE\}, \{LE\}, \{ALE\})$ and since the mandatory feature is either scheduled by deadline or priority there are 30 possible combinations of features that can be compared in performance questions. For instance comparison between for M_D with $\{ALE\}$ features versus M_P with $\{ALE\}$ features, can be used to formulate the following performance question: Which scheduling method is more favourable with the assignment policy, load balancing and escalation handling combination?

The second preliminary task is for specifying the workflow in mind by case study as illustrated in Chapter 5. It should clearly identify various actors (job positions) and roles that can be assumed by them and specifications of users' access rights to artifacts. This documentation should also describe existing and potential future policies, protocols, and procedures, which should be based on the current available model features of assignment policies, load balancing, escalation handling and scheduling protocols.

9.3 Outline

Stage 1: Business Modelling

Task 1.1 Define business domain in terms of UML class and use-case diagrams

Task 1.2 Create a process centric view using activity diagrams

Stage 2: Process Execution Design

Task 2.1 Simulation Model Design

Task 2.2 Extend the UML class and use-case diagrams

Task 2.3 Select domain-independent GT rules in language concrete syntax

Task 2.4 Define domain-specific GT rules in language concrete syntax

Task 2.5 Define probes and selection of rules to observe in VIATRA2 model space

Task 2.6 Define instance graph in StADy concrete syntax

Task 2.7 Select distributions for each rule

Stage 3: Process Encoding

Task 3.1 Define StADy metamodel and model in VIATRA2

Task 3.2 Select predefined managerial GT rules in VIATRA2 textual syntax

Task 3.3 Translate domain specific GT rules in VIATRA2 textual syntax

Task 3.4 Translate probes in VIATRA2

Task 3.5 Encode simulation parameters

Stage 4: Performance Evaluation

Task 4.1 Perform stochastic simulations

Task 4.2 Analyse the results and formulate conclusions

Stage 1 – Business Modelling

The business modelling stage uses standard UML use-case, class and activity diagrams to visually represent the business process, as illustrated in

Section 5.2. This stage is further discussed in Section 9.5 and illustrated in Chapter 10.

Stage 2 – Process Execution Design

The process execution design stage extends the UML use-case and class diagrams to represent additional information in terms of access control, escalation handling and capabilities. The StADy configuration modelling language (Chapter 6) syntax is used to define the business process using the StADy transformation modelling language (Chapter 7). Graph transformation rules are used to define business process activities, whereas instance graphs are used to represent the business process in a state of time. This stage is further discussed in Section 9.6 and illustrated in Chapter 11.

Stage 3 – Process Encoding

The process encoding stage is required as a preparation step in order to use Graph-based Stochastic Simulation (GraSS) tool [30, 68]. The models and graph transformation rules defined in stage 2 will need to be defined in VIATRA2 [72] because GraSS extends VIATRA2. Chapter 8 presents a detailed translation from StADy graphical syntax to VIATRA2's textual syntax. This stage is further discussed in Section 9.7 and illustrated in Chapter 12.

Stage 4 – Performance Evaluation

The performance evaluation stage is for running simulations, analyzing results and producing conclusions. The simulations should compare various simulation models such as the ones informally defined in Section 9.6.1. The results can be analyzed using a combination of graphs and tables. Conclusions to the initial performance questions (Section 1.2) should be made. This stage is further discussed in Section 9.8 and illustrated in Chapter 13.

9.4 Overview of Tasks and Artifacts

Tasks that have local, global or transitive artifact dependencies must occur sequentially, whereas tasks that are independent can occur in parallel. A megamodel is used to provide a global view of the relations between models and metamodels in the StADy approach. Figure 9.1 denotes the order that tasks must occur in, whereas Figure 9.2 shows the StADy megamodel [9] which defines the artifact dependencies. Both of these diagrams artifact/task elements are either unshaded, lightly shaded or darkly shaded to correspond to UML, StADy and VIATRA2 GraSS components, respectively.

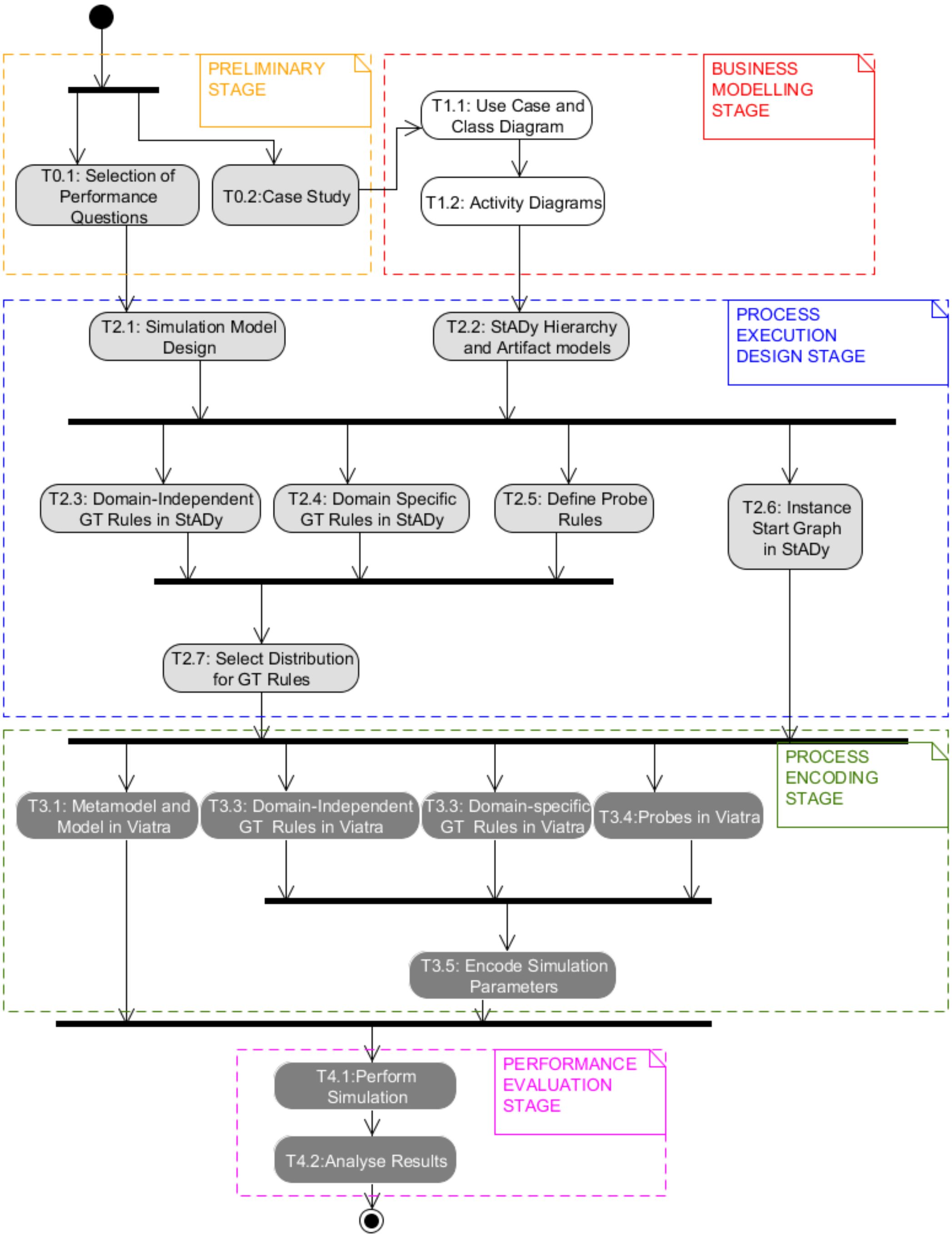

PRELIMINARY STAGE
T0.1: Selection of Performance Questions
T0.2:Case Study
BUSINESS MODELLING STAGE
T1.1: Use Case and Class Diagram
T1.2: Activity Diagrams
PROCESS EXECUTION DESIGN STAGE
T2.1: Simulation Model Design
T2.2: StADy Hierarchy and Artifact models
T2.3: Domain-Independent GT Rules in StADy
T2.4: Domain Specific GT Rules in StADy
T2.5: Define Probe Rules
T2.6: Instance Start Graph in StADy
T2.7: Select Distribution for GT Rules
PROCESS ENCODING STAGE
T3.1: Metamodel and Model in Viatra
T3.3: Domain-Independent GT Rules in Viatra
T3.3: Domain-specific GT Rules in Viatra
T3.4:Probes in Viatra
T3.5: Encode Simulation Parameters
PERFORMANCE EVALUATION STAGE
T4.1:Perform Simulation
T4.2:Analyse Results

Figure 9.1: Order of Tasks

It is important to maintain consistency between the artifacts shown in StADy's megamodel (Figure 9.2). Since the megamodel provides a global view of the relations between the artifacts, each edge implies a required consistency relation between artifacts. The conformance relations and representation is an essential part of this approach, hence consistency checks must be performed at each task. Sections 9.5 to 9.8 describe each task in detail, outline the required consistency checks and Chapters 10 to 13 illustrate how to solve performance questions 1-2 (Section 1.2) for the pharmacy case study.

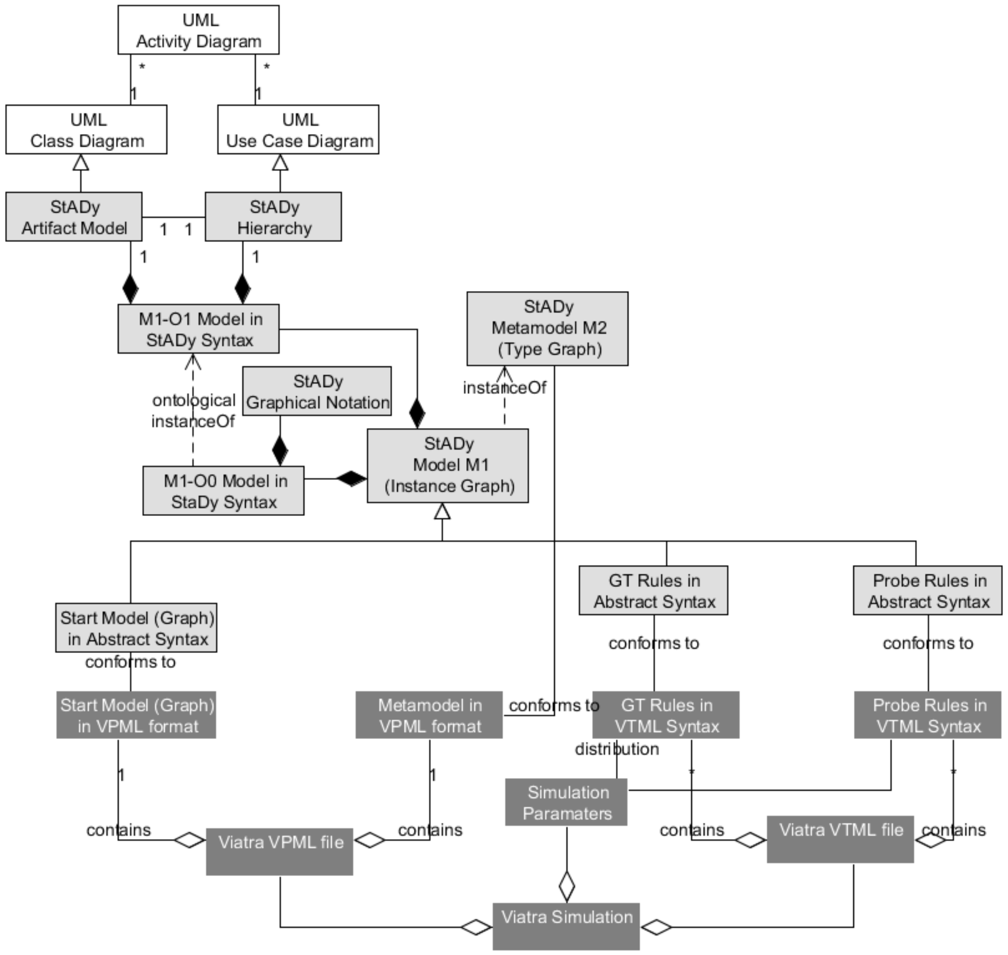


Figure 9.2: Megamodel

9.5 Stage 1 – Business Modelling

This section describes the tasks in stage 1 of the StADy methodology. Each task description generically illustrates portions of the procedure, and defines consistency checks. The business process modelling stage uses existing modelling techniques to graphically define the business process. Task 1.1 defines the business domain using use-case and class diagrams, whereas task 1.2 defines the process centric view using activity diagrams.

9.5.1 Business Domain (Task 1.1)

Task 1 requires standard UML use-case and class diagrams (Section 3.4) to visually represent the business domain. The use-case diagram should include the actor use-case relations in the system. Each use-case in the diagram should have corresponding use-case description. Figure 9.3 shows a generic use-case structure.

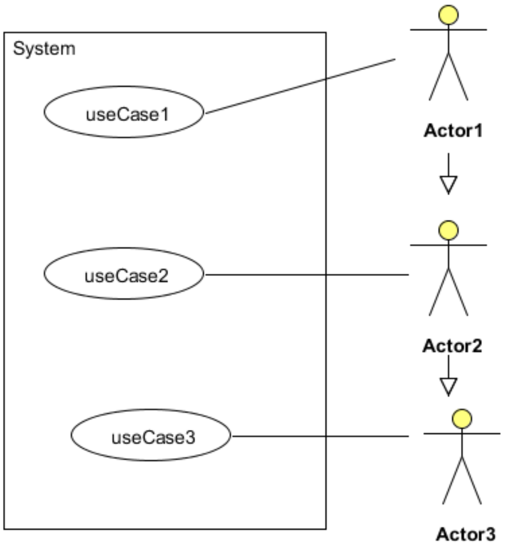


Figure 9.3: Generic UML Use-Case Diagram

Since a use-case is composed of various artifacts, a class diagram can be used to show the types of artifacts that are mentioned in the use-case descriptions. The class diagram should represent the actors defined in the use-case using the ≪actor≫ stereotype. The use of the composition association can help define which artifacts are composed in a particular case, whereas navigability association can be used to identity an actor's access rights to an artifact. Figure 9.4 depicts the generic structure of the required class diagram.

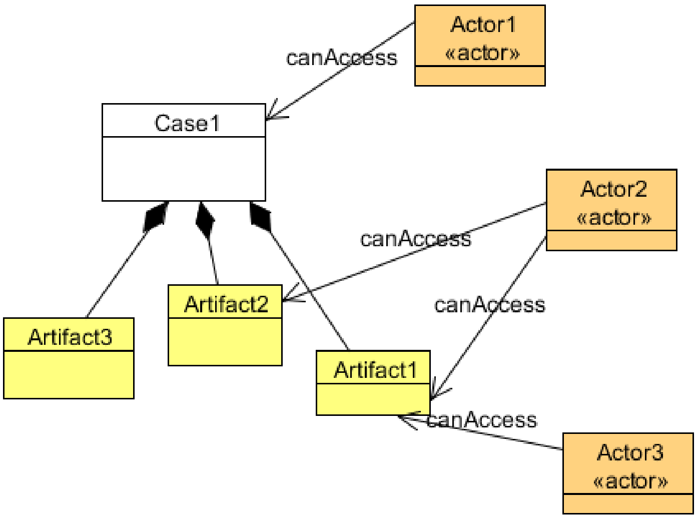


Figure 9.4: Generic UML Class Diagram

This task requires the following consistency checks to ensure consistency between the use-case and class diagram.

Consistency Checks: An approach to checking consistency between UML use-case, and class diagrams is discussed in detail by Chanda et al. [13].

- *UML Class Diagram - UML Use-Case Diagram:* With the assumption that the actor stereotype is used in the class diagram, the actors defined in the class diagram should exist in the use-case diagram.

1. All of the stereotype ≪actors≫ defined in the class diagram must be contained in the use-case diagram.
2. The class diagram should contain the artifacts mentioned in the use-case description.
3. Additional parameters defined in the use-case description should be defined as attributes.

9.5.2 Process Centric View (Task 1.2)

The process centric view task is used to define the control flow of the workflow with standard UML activity diagram notation using enhanced representation for coherent object flow, roles and sub activities (Section 3.4). Artifacts should be specified using coherent object flow and swimlanes should specify which actors can perform particular activities. The activity diagram can also be decomposed using a rake symbol and corresponding sub activities.

This task requires the following consistency checks to ensure consistency between the activity, use-case and class diagram artifacts (Section 9.4).

Consistency Check:

- *UML Activity Diagram - UML Use-Case Diagram:*

1. According to guidelines described by Microsoft authors in [45] the overall workflow goal expressed in the activity diagram should be represented in the use-cases, such that the actions of the workflow are the use-cases.
2. The actor defined in the activity diagram's swim lanes should pertain to the actors defined in the use-case diagrams [13].

- *UML Activity Diagram - UML Class Diagram:* The objects defined in the activity diagram should correspond to types defined in the class diagram [13].

9.6 Stage 2 – Process Execution Design

This section describes the tasks in stage 2 of the methodology. Each task description includes consistency checks. The process execution design stage extends the UML use-case and class diagrams that were defined in stage 1 (Section 9.5) to represent additional information in terms of access control, escalation handling and capabilities. The StADy modelling language (Chapters 6-7) is used to define the business process using a graph transformation rule-based process specification. Graph transformation rules are used to define business process activities, whereas instance graphs are used to represent the business process in a state of time.

9.6.1 Simulation Model Design (Task 2.1)

This task is for designing simulation models based on performance questions such as those presented in Section 1.2. Since the performance questions are based on the features defined in the feature diagram in Figure 2.2, each model can be designed to incorporate particular features for comparison in order to answer the performance questions.

This task requires the following consistency check to ensure consistency between the simulation models and performance questions.

Consistency Check:

- *Simulation Models- Performance Question:* The combination of the simulation model versions should incorporate all of the features mentioned in the performance questions.

9.6.2 StADy Hierarchy and Artifact Models (Task 2.2)

This task extends the UML class and use-case diagrams that were defined in task 1.1 (Section 9.5.1) to contain additional information in terms of access control, escalation handling and capabilities. The extended use-case and class diagrams are renamed to StADy's hierarchy and artifact models respectively, as shown in Figures 9.5 and 9.6, which were discussed and illustrated in Section 6.3.

Access control aspects are added to UML use-case and class diagrams defined in stage 1, by defining allowable roles in the use-case diagram and specify role access rights in the class diagram. The StADy hierarchy model defines allowable roles on the endpoints of an actor — use-case relations. In the StADy artifact model, access rights of roles to artifacts can be specified by listing the hierarchy model role names within the artifacts that they are permitted to access under the stereotyped keyword ≪role≫. If an actor is assigned to a particular role for a particular case instance, then the actor can access the corresponding artifacts.

The hierarchy model permits multiple inheritance amongst actors. It can also be used to define capability and escalation constraints by defining set notation underneath the actor symbol, such that if at least one constraint holds true, then the actor can take on a given role. Therefore, the hierarchy model visually expresses some additional information defined in use-case description with multiple inheritance, constraints and allowable roles on the endpoints of actor-case association.

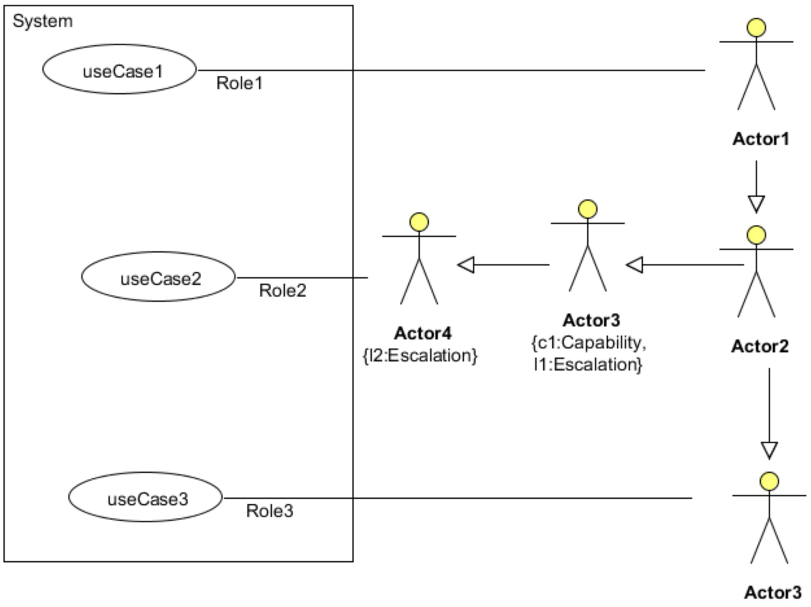


Figure 9.5: Generic StADy Hierarchy Model

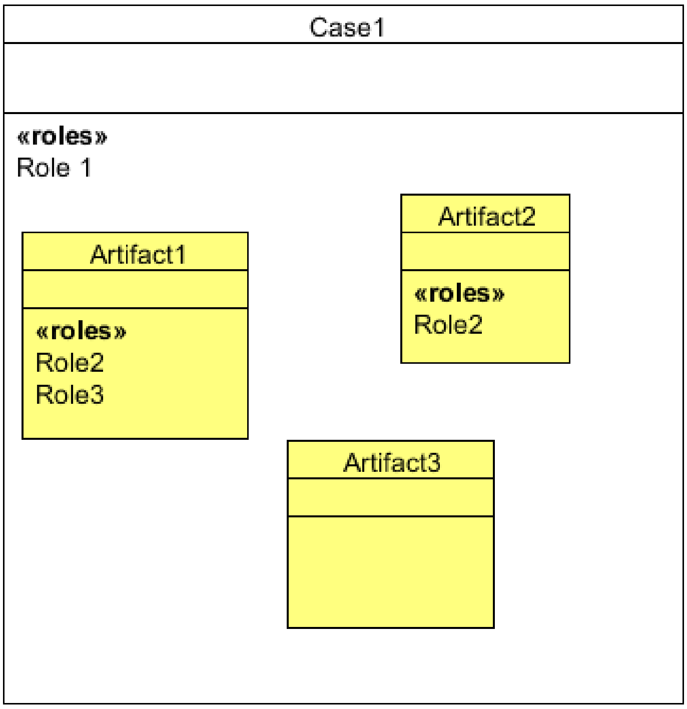


Figure 9.6: Generic StADy Artifact Model

This task requires the following consistency checks to ensure consistency between the StADy hierarchy model, StADy artifact model, use-case diagram and class diagram artifacts.

Consistency Checks:

- *StADy Hierarchy Model - Use-Case Diagram:* The StADy Hierarchy Model must contain the original Use-Case Diagram.

- *UML Class Diagram - StADy Artifact Model:*

  1. Each ≪actor≫ stereotype *artifact* association in the original class diagram should correspond to the refined actor–role association,

by having the associated role name written under the stereotyped keyword ≪role≫ inside *artifacts* in the *StADy artifact model*.

2. Each artifact that has a composition relation to a Case in the class diagram, should be represented as a box inside the Case in the Artifact Model, which is commonly used to represent containment in DSL.

- *StADy Hierarchy Model - StADy Artifact Model:*

  1. The artifact role access that is defined in the Artifact Model should correspond to roles defined in the Hierarchy model.
  2. The combination of both models is an instance of the O1 side of the metamodel.

9.6.3 Domain-Independent GT Rules in StADy notation (Task 2.3)

This task selects predefined managerial GT rules from Section 7.2 in order to help solve the performance questions (Section 1.2) in mind. This task occurs in a two step process by first choosing features from the feature diagram then selecting rules accordingly. The GT rule selection is dependent on the features chosen for the simulation models designed in task 2.1 (Section 9.6.1). The selection can be accomplished because each generically defined managerial GT rule and graphical constraint defined in Section 7.2 directly corresponds to features defined in the feature diagram in Figure 2.2.

An overview of the generic managerial GT rules and graphical constraints is as follows:

- Clock: The current date and time is captured by a Clock node. Syntactically, rules can access the current time through the time attribute in the Clock node to compute deadlines or validate conditions to trigger an escalation; hence if a case deadline has passed, the escalation level of the case should be increased.
    - Clock Tick Rule: This increments the model's local clock, which is used to provide a model of time passing.
- Dynamic Assignment: The dynamic (re)-assignment feature requires request, assignment and unassignment rules.
    - Request Rule: Uses local state labels to determine if an actor is required for a particular case.
    - Assignment Rule: Uses the post condition of request rules to assign actors to roles.
    - Unassignment Rule: Uses the state label to determine if the actor has finished performing his or her duties.
- Assignment Policy: predefined assignment protocols can be used to determine who should be assigned to a particular role
    - Least Qualified Assignment Policy Rule: specifies that the least qualified available worker able to do the job, should be assigned.
- Schedule: The scheduling feature provides the option to schedule according to deadlines or priorities. Hence, order of assignment to the

case can be based on an earlier deadline or a higher priority constraint.

- Request by deadline Rule: The assignment scheduling is influenced by the case's deadline; hence the one with the earlier deadline would have the person requested to be assigned to it first.
- Request by priority Rule: The assignment scheduling is influenced by the case's priority; hence the one with the higher priority would have the person requested to be assigned to it first.

The generic (domain-independent) GT rules and constraints are further discussed in Section 7.2.

9.6.4 Domain Specific GT Rules in StADy notation (Task 2.4)

This task uses the combination of StADy models created in task 2.2 (Section 9.6.2), StADy metamodel (type graph) specified in Section 6.1 and StADy graphical notation (Section 6.4) to define domain specific production-level and support-level GT rules (Chapter 7).

The use-case diagram and its corresponding use-case descriptions produced in task 1.1 (Section 9.5.1) can be used to formulate the production-level domain specific rules, such that each use-case corresponds to at least one rule and the use-case description pre and post conditions informally describe the contents of the rule. Also, each activity in the activity diagram defined in task 1.2 (Section 9.5.2) corresponds to one or many non-deterministic production-level graph transformation rules.

On the other hand, the support-level rules are a modification of the predefined rules defined in Section 7.3 in order to match the business domain that is being modelled.

This task requires the following consistency checks to ensure consistency between activity diagrams, use-case diagram, GT rules and StADy metamodel.

Consistency Checks:

- *UML Activity Diagram - GT Rules:*

  1. Each activity defined in the activity diagram must correspond to one or many GT rules. If the activity corresponds to multiple GT rules, then the preconditions of those rules must be the same.
  2. The objects defined as input and output parameters of the activity must also be present in the pre and post conditions of the corresponding rules.

- *UML Use-Case Diagram - GT Rules:* Each use-case corresponds to at least one rule and the use-case description informally describes the contents of the pre and post conditions of the rule.

- *StADy Metamodel - GT Rules:* The GT Rules must be typed over the StADy metamodel.

9.6.5 Probes and Observation Rules in StADy notation (Task 2.5)

This task is required for analysing the simulation results based on application of rule frequency or pattern matching with probes. The observed rule frequency requires a selection of GT rules that are of particular interest to the business expert. For instance, if the assignment GT rule application frequency was observed then the average number of times assignment occurred in a simulation run can be observed. A probe is a pattern match that captures a particular pre-condition throughout a simulation run. No actions are performed with a probe rule; hence the post condition of the probe rule does not have an effect on the model. Probe rules can be graphically defined using StADy concrete syntax. This task is used to decide on tests that can be used during the simulation to help solve the initial performance questions (Section 1.2) by using observed rule frequency and average probe occurrence.

This task requires the following consistency checks to ensure consistency between probes, StADy metamodel, StADy hierarchy model and artifact model artifacts.

Consistency Check:

- *StADy Metamodel - Probes and Observation Rules:* The probes and observation rules must be typed over the StADy metamodel.

9.6.6 Start-Graph in StADy DSL (Task 2.6)

This task is required for defining the initial state (start graph) of the simulation model. Start graphs can be used as a basis of comparison for simulations to ensure all the simulation runs that are being compared all start from the same initial state.

This task requires the following consistency check to ensure consistency between the start graph and StADy metamodel.

Consistency Check:

- *StADy Metamodel - Start Graph:* The start graph must be typed over the StADy metamodel.

9.6.7 Stochastic Graph Transformation System (Task 2.7)

This task extends the GTS into an SGTS in preparation for stochastic simulation that is performed in stage 4. The Graph-based Stochastic Simulation (GraSS) [30, 68] tool that was selected to be used in the StADy methodology is currently limited to exponential and lognormal distribution specification. Therefore, each GT rule needs to be categorized into exponential or lognormal distribution, and then associated with specified scheduling delays in order to define the stochastic delays for each rule in GraSS [30, 68]. The exponential distribution requires a corresponding rate, whereas the lognormal distribution requires mean and variance values. These scheduled delays should be well selected and properly researched in order to reflect realistic simulation results, as described in Section 4.3. This task is achieved by gathering stochastic data about the activities in the business process and assigning distribution parameters to the domain-independent and domain-specific rules defined in tasks 2.3-2.4 (Sections 9.6.3-9.6.4). The stochastic data comes from quantitative research that has been either published in papers or books or gathered from own observations. The published works should be research projects in the same business process of interest; however if there are minor inconsistencies in the business process activity breakdown then consult a domain expert for advice of possible estimations for some of the values. Since the correctness of data is crucial distributions that are estimated need to be tested in a simulation run to see if they represented meaningful data in relation to the research statistics. The impact of distribution selection is further discussed and illustrated in Section 15.6.4.

This task requires the following consistency checks to ensure consistency between the GT rule and distribution artifacts.

Consistency Checks:

- *GT Rule- Distribution:* Each defined GT Rule must have a distribution.

9.7 Stage 3 – Process Encoding

This section describes the tasks in stage 3 of the methodology. Each task description includes consistency checks. The process-encoding stage is for encoding the start graph from task 2.6 (Section 9.6.6), GT rules from tasks 2.3-2.4 (Section 9.6.3-9.6.4) and probes from task 2.5 (Section 9.6.5) into the VIATRA2 tool, in order to perform stochastic simulation on Graph-based Stochastic Simulation (GraSS) tool [30, 68] which extends VIATRA2. Chapter 8 discussed and illustrated the mapping from StADy concrete syntax to VIATRA2 textual syntax.

9.7.1 Start Graph in VIATRA2 Model Space (Task 3.1)

This task is for defining simulation start graph from task 2.6 (Section 9.6.6) into the VIATRA2 (VPM) Model Space [72], by using the predefined StADy metamodel (Section 6.1). The metamodel and model data should be entered into a VPML file using the VIATRA2 Model Space tree editor. Screen shots of the StADy metamodel VIATRA2 representation is available in Section 8.1, whereas screen shots of different start graphs are available in Section 8.2.

This task requires the following consistency checks to ensure consistency between the start graph in StADy graphical syntax and start graph in VPM Syntax.

Consistency Checks:

- *Start Graph in StADy Syntax - Start Graph in VPM Syntax:* The model element entities and relations in VPM must correspond to the entities and relations found in the abstract syntax representation of the graphical syntax.

9.7.2 Domain-Independent GT Rules in VIATRA2 Syntax (Task 3.2)

This task is for defining the selected predefined domain-independent GT from task 2.2 (Section 9.6.3) into a VIATRA2 Textual Command Language (VTCL) document. The textual representation of all of the managerial-level rules (Section 7.2) is available in Appendix A.1. Section 8.3.3 illustrates the mapping between the GT rule notations.

9.7.3 Domain-Specific GT Rules in VIATRA2 Syntax (Task 3.3)

This task is for translating the domain-specific rules that are created in task 2.4 into the VIATRA2 textual syntax [72] as discussed in Section 8.3. The support-level GT rules are predefined in Appendix A.2 and need to be modified to match the business domain being modelled, as illustrated in Section 8.3.4. The production-level GT rules need to be manually mapped into VIATRA2 textual syntax, as illustrated in Section 8.3.2 and Appendix A.3. Each production-level GT contains state data, skeleton VIATRA2 syntax code for coding states GT rules which is available below:

```
 //With State Type
gtrule DomainSpecificSkelton() =
    {
            precondition pattern lhs(Case_, StateInstance_, NextState_,R6_) =
            {
                Case(Case_);
                State(State_);
                State(NextState_);
                StateInstance(StateInstance_);

                Case.currentState(R6_,Case_,StateInstance_);

                typeOf(State_,StateInstance_);
                check(name(State_)=="<state name>");
                State.next(R1,State_,NextState_);
                .             .
                .
                .
            }
            action {
                  let
                    NewStateInstance_=undef,
                    R2_=undef

            in seq {

                delete(StateInstance_);
                delete(R6_);
                new(StateInstance(NewStateInstance_)in DSM.model.M1);
                new(instanceOf(NewStateInstance_,NextState_));
                new (Case.currentState(R2_,Case_,NewStateInstance_));
                .
                .
                .
            }
        }
    }
```

This task requires the following consistency checks to ensure consistency between the GT rules in StADy graphical syntax and GT rules in VTML Syntax.

Consistency Checks:

- *StADy Graphical syntax- VIATRA2 textual syntax:* The textual and a graphical representation are equivalent if they generate the same abstract representations of a particular model or there exist a mapping from the graphical representation to textual representation based on the discussion in Section 8.3.

9.7.4 Probes StADy in VIATRA2 Syntax (Task 3.4)

This task translates the probes defined in task 2.5 (Section 9.6.5) into VIATRA2 textual syntax, which has similar format as GT rule in VIATRA2, except the action (postcondition) is empty. The VIATRA2 textual syntax was discussed in Section 8.3. The following VIATRA2 code outlines a probe rule structure:

```
gtrule Probe() =
{
    precondition pattern lhs() =
    {
        //Enter pattern to match
    }
    action
    {
        //Do nothing
    }
}
```

This task requires the following consistency checks to ensure consistency between the probes in StADy graphical syntax and probes in VPM Syntax (Section 9.4).

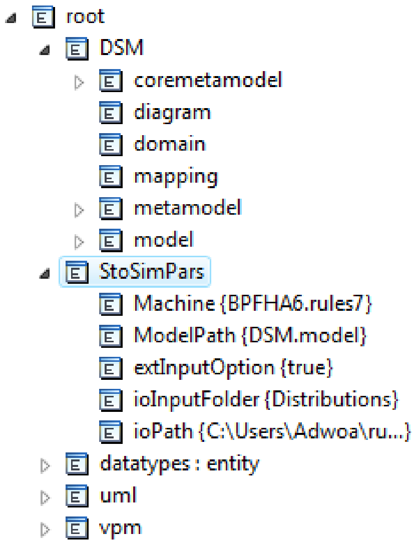
Consistency Checks:

- *StADy Graphical syntax- VIATRA2 textual syntax:* Probe Rules can be represented in two types of concrete syntax, which are instances of the same StADy metamodel; therefore the abstract syntax representation (Section 6.2) of the model can be used to validate consistency. The textual and graphical representation are equivalent if they generate the same abstract representations of a pre-condition model defined in the probe, or there exist a mapping from the graphical representation to textual representation based on the discussion in Section 8.3.

9.7.5 Simulation Parameters (Task 3.5)

The GraSS [30, 68] tool requires initial execution and stochastic setup in addition to the VTCL and VPML files created in task 3.1-3.4 (Sections 9.7.1-9.7.4). This section discusses task 3.5, which is for defining the simulation parameters into the VIATRA2 model space (VPML file) and XML files. The main control parameters are stored in the *StoSimPars* element of the model space (VPML file) as illustrated in Figure 9.7. The values of the entities within *StoSimPars* correspond to external simulation parameters. The entities within *StoSimPars* consist of *Machine*, *ModelPath*, *ioPath*, *ioIn-*

putFolder, and *extInputOption* values which correspond to external simulation parameters. The *Machine* entity defines the VIATRA2 machine i.e. *BPFHA6.rules7*, such that BPFHA6.vpml and rules7.vtcl are being used. The *ModelPath* entity defines the relative path to the model element (defined in task 3.1) i.e. *DSM.model*, such that the model is located as a subset to the DSM entity. The *ioPath* defines the absolute path to the graph transformation system specifications. The *ioInputFolder* sets the name of the input XML folder i.e, the *Distributions* folder is the subfolder of the folder defined in *ioPath*. The *extInputOption* entity is used to determine whether the additional parameters are to be read from the XML file or to be found already in the model space, if the value is set to true then additional parameters can be read.



```
◢ E root
  ◢ E DSM
    ▷ E coremetamodel
      E diagram
      E domain
      E mapping
    ▷ E metamodel
    ▷ E model
  ◢ E StoSimPars
      E Machine {BPFHA6.rules7}
      E ModelPath {DSM.model}
      E extInputOption {true}
      E ioInputFolder {Distributions}
      E ioPath {C:\Users\Adwoa\ru...}
  ▷ E datatypes : entity
  ▷ E uml
  ▷ E vpm
```

Figure 9.7: StoSimPars Entity

The additional parameters are defined in a parameter.xml file, which defines internal parameters such as runtime specifications. A sample parameter.xml file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<allparameters>

    <iopars>
        <parameter name ="CDF_Input" value="rules6.xml" />
        <parameter name ="Output_Folder" value="ioF1" />
        <parameter name ="Output_Files" value="log" />
    </iopars>

    <runpars>
        <parameter name ="Rule_Set" value="random" />
        <parameter name ="Depth_Limit" value="100" />
        <parameter name ="Time_Limit" value="50" />
        <parameter name ="Time_Opt" value="false" />
        <parameter name ="Batch_Size" value="4" />
    </runpars>

    <varpars>
        <parameter name ="Variation_Type" value="S" />
        <parameter name ="Variations" value="3" />
        <parameter name ="Factor_Up_Opt" value="true" />
        <parameter name ="Factor_Up_Base" value="10" />
    </varpars>

    <outpars>
        <parameter name ="Confidence_Level" value="0.90" />
        <parameter name ="Hierarchical_Aggregation" value="short" />
        <parameter name ="2D_Linear_Aggregation" value="full" />
        <parameter name ="Debug_Level" value="0" />
    </outpars>

    <expars>
        <parameter name ="Feedback_Opt" value="false" />
        <parameter name ="Clock_Name" value="r1.clock" />
        <parameter name ="Extra_Attribute" value="" />
        <parameter name ="Extra_Operator" value="" />
    </expars>

</allparameters>
```

The distribution values defined in task 2.7 (Section 9.6.7) need to be entered into an XML file which contains the stochastic input. The input values refer to cumulative distribution functions (CDF) of the GT rules encoding in tasks 3.2–3.3 (Sections 9.7.2–9.7.3). The GT rules that were defined in the previous tasks are referred to as *action rules (A)* in the GraSS tool. Each action rule can be categorised as *observable (O)* and/or *sensitive (S)*. If the rule is *observable*, then the simulator will return statistics about the rule application and delay. If the rule is *sensitive*, then the CDF can vary throughout different batches. *Probe (P)* rules will provide statistics of the number of rule matches at each step. The stochastic input file is composed of the assignment of CDF to the action rules and information to define how to use the rules with respect to transformation and probing. Below is an extract of a stochastic input xml file. The *ruleset* name in the CDF file corresponds to the parameter defined in parameter.xml file. Each action rule defined in the *vtcl* file should be assigned a CDF. If it is an exponential distribution then the structure should be similar to Rule1 and if it is lognormal distribution then the structure should be similar to Rule2. The rule type attribute distinguishes the type of action or probe rule i.e. *AO* sets the rule as applicable observable action rule. Each probe should be defined similar to Probe1 or Probe2. The probe definition is shown in the *probeset*, such that the *op* corresponds to the name of a predefined probe in the vtcl file or an operation. The operation div, sum and divsq (division square) uses probes as arguments as illustrated in *Probe2* probe.

```
<?xml version="1.0" encoding="UTF-8"?>
<allrules>
    <ruleset name="random">
        <rule name="Rule1" type="AO">
            <event name="0" type="exp">
            <rate value="0.01667"/>
        </event>
    </rule>

    <rule name="Rule2" type="AO">
        <event name="0" type="norm">
            <mean value="1.25" />
            <variance value="0.4375" />
        </event>
    </rule>

    <probeset>
        <probe name="Probe1" op="ProbeName1"> </probe>

        <probe name="Probe2" op="div">
            <arg name="ProbeName1" pos="2"></arg>
            <arg name="ProbeName2" pos="1"></arg>
        </probe>

        </probeset>
    </ruleset>
</allrules>
```

This task requires the following consistency checks to ensure consistency between the GT rule/probe distributions and encodings of their distributions.

Consistency Checks:

- *GT Rule Distributions- Encodings of Distributions:* All GT Rule distribution parameters that were selected for a particular simulation model version GT rules must be encoded into the stochastic input file.
- *Probes- Encoding of Probes:* Each probe formulated in task 3.4 should be added to the probe definition list in the stochastic input file.

9.8 Stage 4 – Performance Evaluation

This chapter describes the tasks in stage 4 of the methodology. The performance-evaluation stage is for running simulations, analyzing results and producing conclusions.

9.8.1 Stochastic Simulation (Task 4.1)

This task is for running the stochastic simulation in Graph-based Stochastic Simulation (GraSS) tool [30, 68]. Simulation experiments consist of a set of runs, as specified by the parameters file defined in task 3.5 (Section 9.7.5). All runs within the same batch share the same stochastic parameters, although the user has the option to alter the parameters between different batches.

A good recommendation is to run the simulation with a high max depth or high max simulation time, in order to produce realistic results. During the simulation run information is printed out to the Eclipse console depending on the debug level that was set in the parameter. For instance, level 3 prints out the internal representation, whereas level 1 only prints out the applied rules during execution.

9.8.2 Results and Conclusions (Task 4.2)

This task is for analysing the simulation results produced from the simulation experiment performed in task 4.1 (Section 9.8.1). The statistical data that are obtained from the simulation runs capture information on GT (action) rules applied and defined probe rules. The final results include the number of matches of the probe rules and application and timing of GT rules. It also defines correlations between pairs of rules and can group the results in batches, slices or an experiment.

The simulation results are represented in the format specified in the parameter.xml file. If the hierarchical aggregation option was chosen then the statistics are printed for each run, batch and experiment. The results are presented in comma-separated values, statistics as JSS reports and covariance analysis output. If the 2D-linear aggregation option was selected then the results are aggregated for batches and slices, such that the aggregation is used for runs of the same batch and slice is set of runs in different batches.

The final two stages involve normalizing the data and visually representing the data with bar graphs using a tool such as Microsoft Excel. This can be used to analyse the results in order to come up with final conclusions.

9.9 Summary

This chapter introduced the StADy methodology for modelling and analysing business processes with human involvement, by defining the scope of business problems, discussing preliminary tasks, outlining the methodology with diagrams of task and artifact relations and presented details of each stage. The next five chapters will illustrate how to answer the performance questions defined in Section 1.2 using the StADy methodology.

Chapter 10

Illustration of Stage 1 – Business Modelling

This chapter illustrates the tasks in stage 1 of the methodology (Chapter 9) on the pharmacy case study defined in Chapter 5. The business process modelling stage uses existing modelling techniques to graphically define the business process. Task 1.1 defines the business domain using use-case and class diagrams, whereas task 1.2 defines the process centric view using activity diagrams.

10.1 Business Domain (Task 1.1)

This section illustrates the application of task 1.1 on the pharmacy case study (Chapter 5). The pharmacy business domain is defined in terms of UML use-case and class diagram and the consistency between these artifacts is be tested.

As defined and described in Section 5.2, Figures 10.1 and 10.2 define the use-case and class diagrams of the pharmacy business domain respectively. Tables 10.1 to 10.22 present the use-case descriptions of the use-case diagram. Consistency holds true between the use-case and class diagram because they passed the consistency tests as follows:

1. All of the stereotype ≪actors≫ defined in the class diagram must be contained in the use-case diagram.
   - This is true because *Cashier*, *Patient*, *Technician* and RegPharmacist are actors defined in the use-case diagram.
2. The class diagram should contain the artifacts mentioned in the use-case description.
   - *Prescription* (Table 10.2)
   - *TypePrescription* (Table 10.5)
   - *Label* (Table 10.8)
   - *FillPrescription* (Table 10.11)
   - *Payment* (Table 10.18)
3. Additional parameters defined in the use-case description should be defined as attributes.
   - *type: [Enum]* {*walkin,sameday,delivery,refill*} (Table 10.4)
   - *checked: Boolean* (Table 10.15)
   - *counsel:Boolean* (Table 10.21)

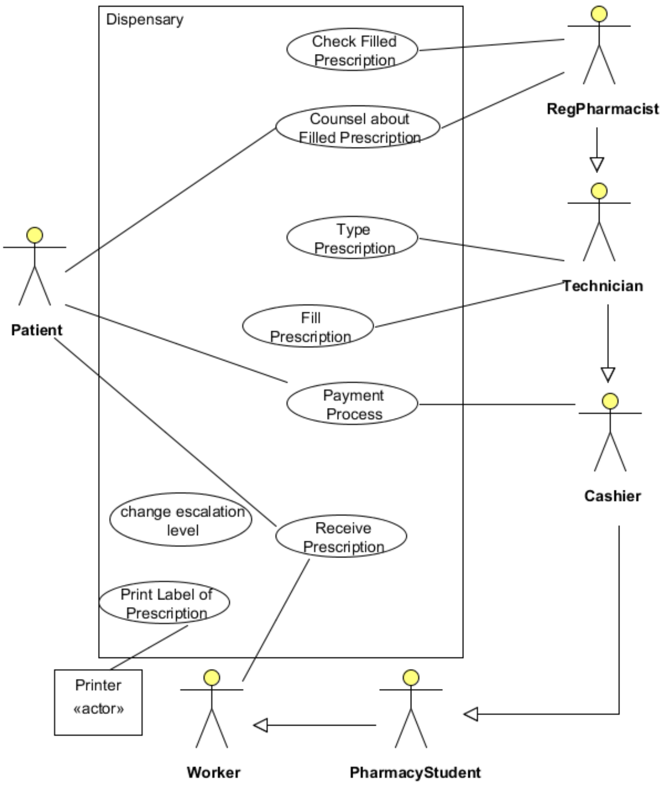


Figure 10.1: UML Use-Case Diagram of the Pharmacy Business Process

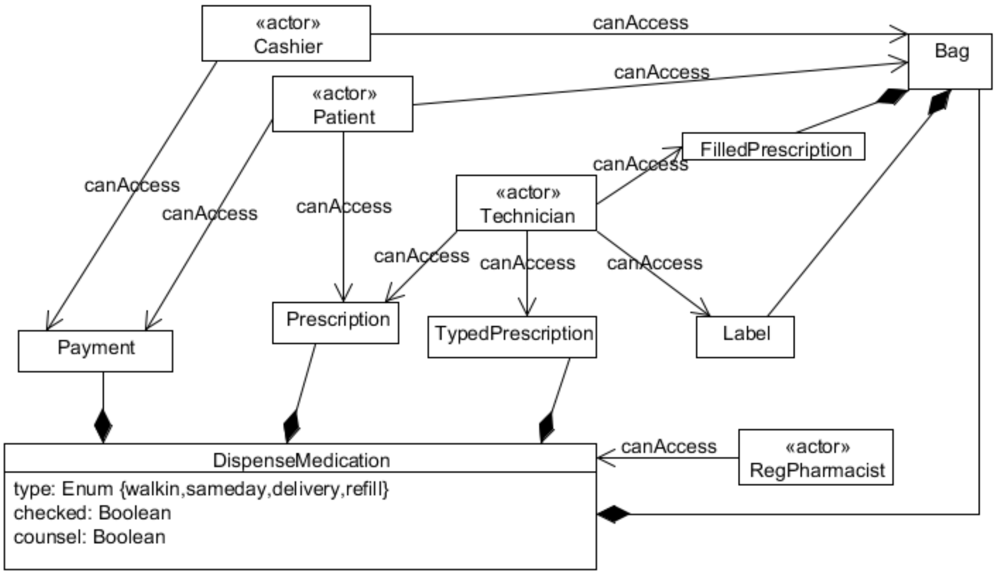


Figure 10.2: UML Class Diagram of the Pharmacy Business Process

Change Escalation Level

Name:	Change Escalation Level
Primary Actor:	
Goal (of the User):	Increment Escalation Level
Precondition:	if (Prescription case deadline is less than or equal to 5 minutes away from the deadline) else if (Prescription case has reached or passed the deadline) else if (Prescription case is 5 mins or more passed the deadline)
Postcondition **(successful execution):**	if (Prescription case deadline is less than or equal to 5 minutes away from the deadline) then set Escalation level to 1 else if (Prescription case has reached or passed the deadline) then set Escalation level to 2 else if (Prescription case is 5 mins or more passed the deadline) then set Escalation level to 3
Trigger Event:	case deadline approaching time
System (implementing it):	Dispensary
Participating Actor:	

Table 10.1: Use-Case Details: Change Escalation level

Receive Prescription Case Description

Name:	Receive Prescription
Primary Actor:	Worker
Goal (of the User):	receive prescription and set deadline
Precondition:	prescription request to be filled
Postcondition (successful execution):	new prescription case
Trigger Event:	new prescription fill request
System (implementing it):	Dispensary
Participating Actor:	Worker, Patient

Table 10.2: Receive Prescription Case Details

Step	**User**	**Action**
1	Patient	brings Prescription to store
2	Worker	receives prescription
3	Worker	informs patient that prescription will be filled within 15 minutes and sets completion time for 15 minutes from the current time

Table 10.3: Case Scenario: Receive Prescription

Step	**Condition for alternative**	**Alternative Action**
1	refill prescription ordered online	Prescription is set for online refill
3	Patient plans to return later to pickup prescription	Prescription is set for future pickup and completion time is set to 1 hour from the current time
3	Patient asks for prescription to be delivered	Prescription is set for delivery and completion time is set to 15 mins from the current time
3	if prescription is set as online refill	The Worker sets the deadline to 24 hour from the current time

Table 10.4: Case Alternative Scenario: Receive Prescription

Type Prescription Case Description

Name:	Type Prescription
Primary Actor:	Technician
Goal (of the User):	type prescription and send to printer
Precondition:	prescription not typed yet
Postcondition (successful execution):	prescription typed
Trigger Event:	written prescription not in computer system
System (implementing it):	Dispensary
Participating Actor:	Technician

Table 10.5: Type Prescription Case Details

Step	**User**	**Action**
1	Technician	read Prescription
2	Technician	type prescription data into computer

Table 10.6: Case Scenario: Type Prescription

Step	**Condition for alternative**	**Alternative Action**
1 & 2	if Escalation level for the particular prescription case is 1 or higher	Cashier can temporarily take on the Entry Technician role
1 & 2	if Pharmacy Student finishes his or her prescription filling training	Pharmacy student can take on filling technician role.

Table 10.7: Case Additional Exceptions: Type Prescription

Print Label Case Description

Name:	Print Label
Primary Actor:	Printer
Goal (of the User):	successful print of prescription label
Precondition:	typed prescription without label
Postcondition (successful execution):	label printed
Trigger Event:	typed prescription sent to printer
System (implementing it):	Dispensary
Participating Actor:	Printer

Table 10.8: Print Label Case Details

Step	**User**	**Action**
1		receive print job
2	Printer	automatically prints label

Table 10.9: Case Scenario: Print Label

Step	**Condition for alternative**	**Alternative Action**
1	error requesting print job	Printer request data to be resent

Table 10.10: Case Alternative Scenario: Print label

Fill Prescription Case Description

Name:	Fill Prescription
Primary Actor:	Technician
Goal (of the User):	successful print of prescription label
Precondition:	prescription label exist without filled prescription
Postcondition (successful execution):	prescription filled
Trigger Event:	label arrived at filling station
System (implementing it):	Dispensary
Participating Actor:	Technician

Table 10.11: Fill Prescription Case Details

Step	User	Action
1	Technician	pick label from printer
2	Technician	read over label
3	Technician	prepare prescription
4	Technician	put prescription contents into bottle
5	Technician	label bottle

Table 10.12: Case Scenario: Fill Prescription

Step	Condition for alternative	Alternative Action
1	label missing from printer	request printer to reprint label

Table 10.13: Case Alternative Scenario: Fill Prescription

Step	Condition for alternative	Alternative Action
1 to 5	if Escalation level for the particular prescription case is 2 or higher	Cashier can temporarily take on the Filling Technician role
1 to 5	if Pharmacy Student finishes his or her prescription filling training	Pharmacy student can take on the filling technician role.

Table 10.14: Case Additional Exceptions: Fill Prescription

Check Filled Prescription Case Description

Name:	Check Prescription
Primary Actor:	Registered Pharmacist
Goal (of the User):	checks correctness of filled prescription and label
Precondition:	prescription is filled
Postcondition (successful execution):	filled prescription has been checked
Trigger Event:	filled prescription arrived at checking station
System (implementing it):	Dispensary
Participating Actor:	Registered Pharmacist

Table 10.15: Check Prescription Case Details

Step	**User**	**Action**
1	Registered Pharmacist	selects prescription case from the queue
2	Registered Pharmacist	checks that label is correctly typed
3	Registered Pharmacist	checks that prescription is correctly filled
4	Registered Pharmacist	signs off the prescription

Table 10.16: Case Scenario: Check Filled Prescription

Step	**Condition for alternative**	**Alternative Action**
1	prescription is not filled	sends prescription case back to filling technician
2	label is incorrectly typed	gives corrections and case back entry technician to retype
3	prescription is incorrectly filled	notes errors and sends case refill to filling technician to refill

Table 10.17: Case Alternative Scenario: Check Filled Prescription

Payment Process Case Description

Name:	Payment Process
Primary Actor:	Cashier
Goal (of the User):	recieves prescription payment
Precondition:	prescription is ready (i.e. successfully checked))
Postcondition (successful execution):	payment received
Trigger Event:	prescription ready and patient present
System (implementing it):	Dispensary
Participating Actor:	Cashier, Patient

Table 10.18: Payment Process Case Details

Step	**User**	**Action**
1	Cashier	requests payment from Customer
2	Patient	gives payment

Table 10.19: Case Scenario: Payment Process

Step	**Condition for alternative**	**Alternative Action**
2	payment not given	leaves filled prescription aside for a grace period

Table 10.20: Case Alternative Scenario: Payment Process

Counsel about Filled Prescription Case Description

Name:	Counsel about Filled Prescription
Primary Actor:	Registered Pharmacist
Goal (of the User):	counsels the patient
Precondition:	filled prescription has been paid for
Postcondition (successful execution):	filled prescription given to patient
Trigger Event:	prescription paid and patient is present
System (implementing it):	Dispensary
Participating Actor:	Registered Pharmacist, Patient

Table 10.21: Counsel about Filled Prescription Case Details

Step	User	Action
1	Registered Pharmacist	counsels patient
2	Registered Pharmacist	gives filled prescription to patient
3	Patient	receives prescription

Table 10.22: Case Scenario: Counsel

10.2 Process Centric View (Task 1.2)

This section illustrates the application of task 1.2 on the pharmacy case study (Chapter 5). A process centric view of the pharmacy business process is created by defining the control flow in terms of UML activity diagrams (Figures 10.3-10.6).

Figure 10.3 illustrates a typical pharmacy business process. The swim lanes are used to partition the diagram actions based on actor and system roles. The pins correspond to input and output object parameters such as *Label* and *FilledPrescription*. Actions can be decomposed into sub activities and be denoted using the rake symbol; hence there are three sub activity diagrams defined: *Receive Prescription* (Figure 10.4), *Check Process* (Figure 10.5) and *Payment Process* (Figure 10.6). Each of these activities has input and output object parameters defined in boxes at the edge of their border.

Below is description of activities in the activity diagram:

Receive Prescription: A customer submits a prescription to be filled by the pharmacy. Since prescriptions can be one of four types (refill, same day, delivery and walk-in), different actions are performed to create a new *DispenseMedication* based on the prescription type, as shown in Figure 10.4.

Type into Computer: The entry technician receives the *Case* containing the written *Prescription* and types it into the pharmacy software.

Print Label: Once the *TypedPrescription* is received by the printer then the *Label* can be printed.

Fill Prescription: When the filling technician receives the *Label* then he or she can fill the prescription.

Check Prescription: The pharmacist checks the prescription for major errors; if it is faulty he or she sends the *DispenseMedication* to be retyped and refilled otherwise the *FilledPrescription* is passed on to the next step, as shown in Figure 10.5.

Payment Process: The cashier requests and receives *Payment* from the customer, as shown in Figure 10.6.

Counsel: The pharmacist counsels the customers and gives them the completed *FilledPrescription.*

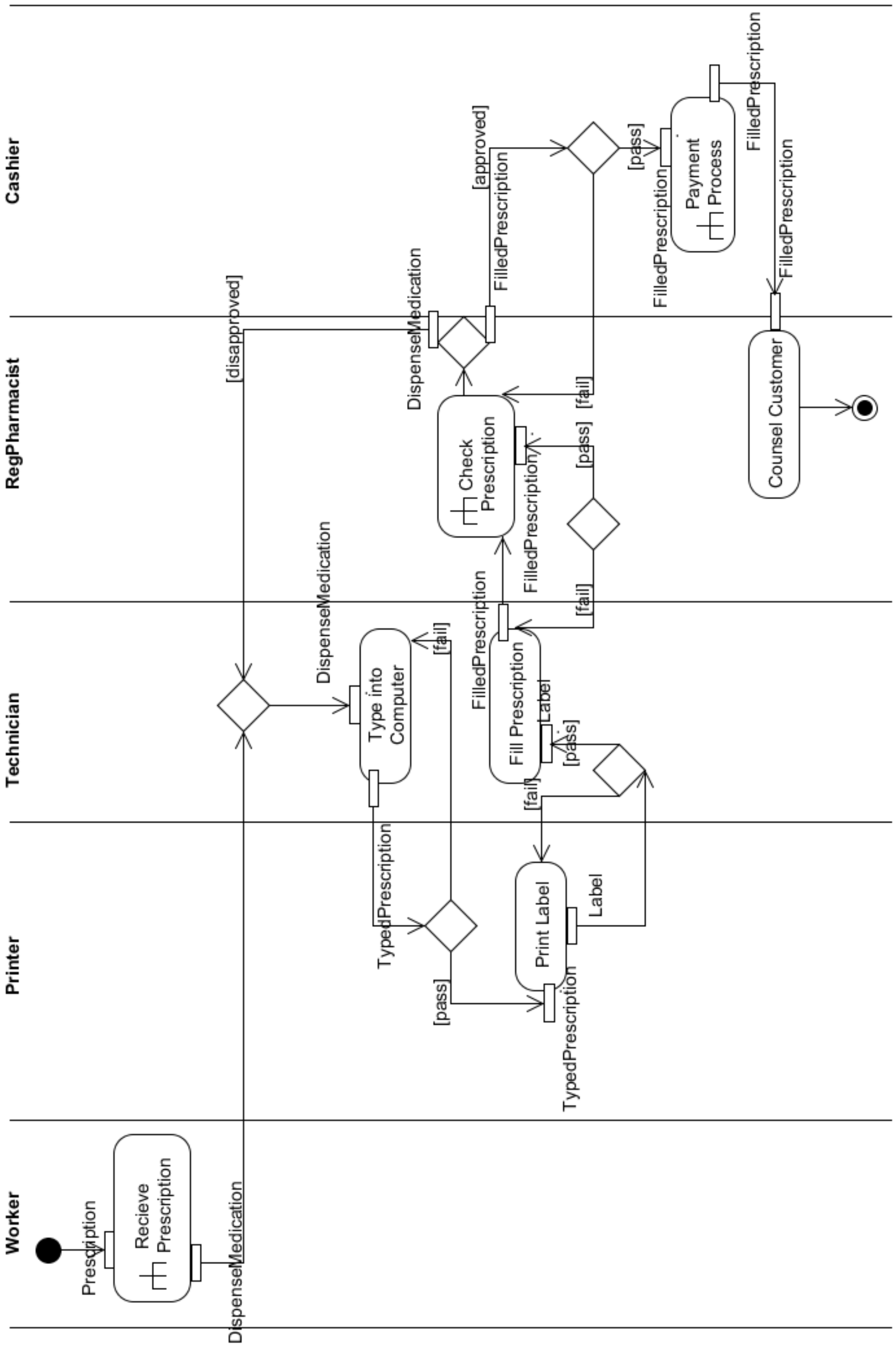


Figure 10.3: Activity Diagram: Dispense Medication

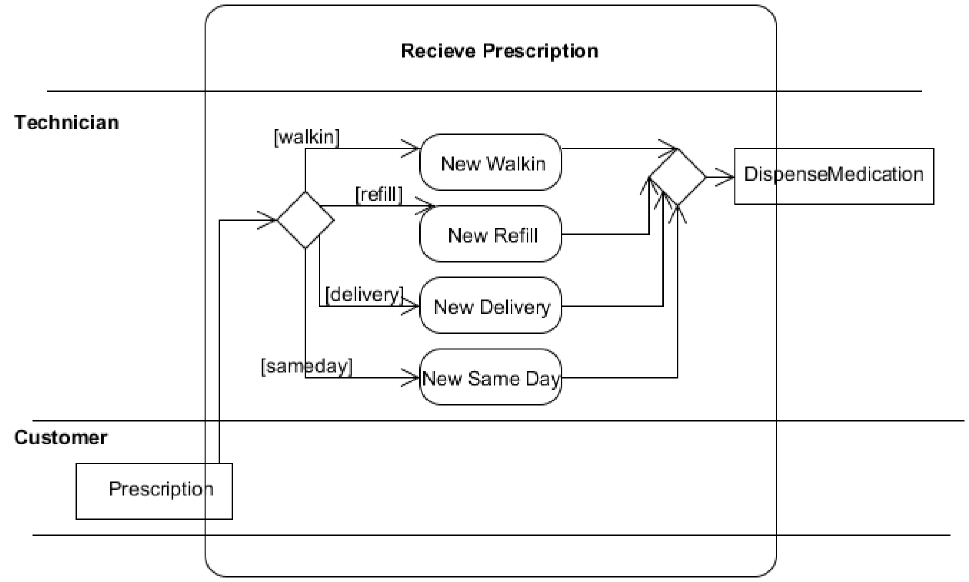


Figure 10.4: Subactivity: Receive Prescription

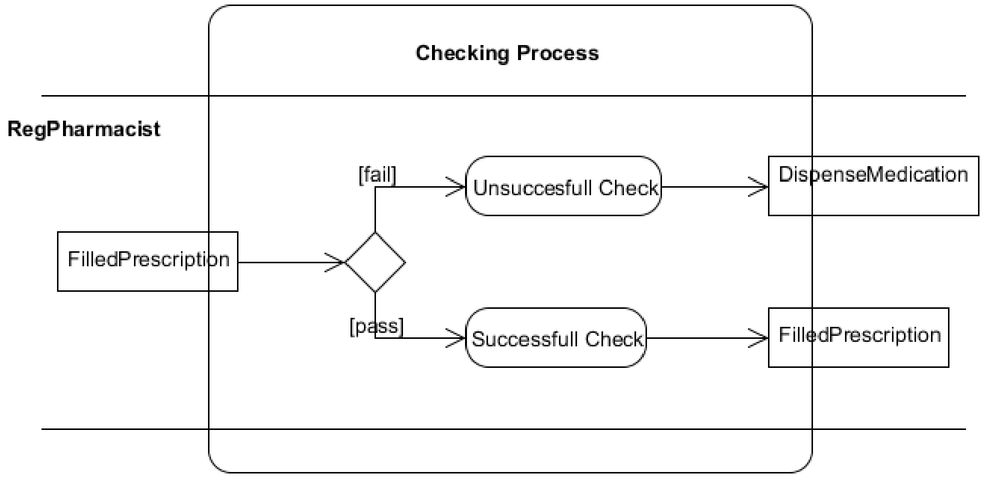


Figure 10.5: Subactivity: Check Process

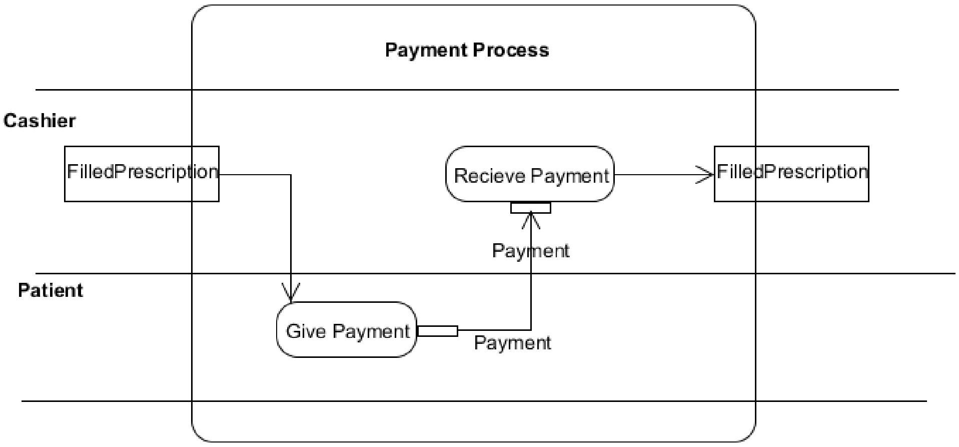


Figure 10.6: Subactivity: Payment process

Consistency holds true between the activity, use-case and class diagrams because they passed the consistency tests as follows:

- *UML Activity Diagram - UML Use-Case Diagram:*

  1. According to guidelines described by Microsoft authors in [45] the overall workflow goal expressed in the activity diagram should be represented in the use-cases, such that the actions of the workflow are the use-cases.

     – This holds true because each activity in the main activity diagram (Figure 10.3) corresponds to a use-case in the use-case diagram defined in task 1.1 (Section 10.1).

  2. The actor defined in the activity diagram's swim lanes should pertain to the actors defined in the use-case diagrams [13].

     – This holds true because the following actors defined in the activity diagram are also defined in the use-case diagram defined in task 1.1 (Section 10.1): *Worker*, *Printer*, *Technician*, *RegPharmacist*, and *Cashier*.

- *UML Activity Diagram - UML Class Diagram:*

  1. The objects defined in the activity diagram should correspond to types defined in the class diagram [13].

     – This holds true because *Prescription*, *DispenseMedication*, *TypedPrescription*, *FilledPrescription*, *Label* and *Payment* types exist in the activity diagram and the class diagram defined in task 1.1 (Section 10.1).

10.3 Summary

This chapter illustrated the business process modelling tasks in stage 1 of the StADy methodology (Chapter 9). This stage defines the business domain using use-case and class diagrams and process using activity diagrams. The next chapter (Chapter 11) will illustrate stage 2, which extends the diagrams to contain additional information and uses them to define domain-specific production level GT rules.

Chapter 11

Illustration of Stage 2 – Process Execution Design

This chapter illustrates the tasks in stage 2 of the methodology (Chapter 9) based on the pharmacy case study defined in Chapter 5. The process execution design stage extends the UML use-case and class diagrams that were defined in stage 1 (Chapter 10) to represent additional information in terms of access control, escalation handling and capabilities. The StADy language (Chapter 6-7) syntax is used to define the business process using a graph transformation rule-based process specification. Graph transformation rules are used to define business process activities, whereas instance graphs are used to represent the business process in a state of time.

11.1 Simulation Model Design (Task 2.1)

This section illustrates how to design simulation models based on initial performance questions presented in Section 1.2. These questions can be reworded to include domain specific details. Below is a pharmacy domain specific version of question number 3.

Question 3 Is it beneficial to temporarily permit cashier and untrained pharmacy students to take on technician roles for prescription cases that are approaching deadline/past deadline (escalation handling)? Is it beneficial to transfer prescriptions to a nearby store (load balancing)? Would these features increase the percentage of prescription cases that are completed within a given deadline, or reduce the time that cases run past their deadline?

Due to escalations dependency relation with the role promotion/demotion, this feature is implicitly tested in Question #3 with escalation handling feature. With these three questions (Section 1.2) in mind this section will explore different simulation models based on the feature model shown in Figure 2.2.

These questions can be answered by performing simulations on the 8 unique model versions outlined in Table 11.1 and 11.2. The models in versions 1-4 compare the use of assignment policies and different types of scheduling (Questions 1-2). On the other hand, the models in versions 5-8 focus on the effect of load balancing and escalation (Question 3) and use scheduling by deadline and assignment policy as default features. Therefore, two distinct simulation designs are required to capture all three of the performance questions. The simulation type 1 should capture comparisons of assignment policies and scheduling routines, whereas simulation type 2 should capture the effectiveness of escalation handling and load balancing. Simulation type 1 is illustrated in the stage descriptions in Chapters 10-13, whereas simulation type 2 is discussed as a second application to the case study in Chapter 14. The scheduling routines will be defined independently from simulation algorithm defined in Section 4.4 by using the StADy transformation modelling language (Chapter 7) to define GT rules and constraints to specify user-defined order which was discussed in detail in Section 7.2.2.

Assignment	**Scheduling**	
	Deadline	*Priority*
No Policy	Version 1	Version 2
Policy	Version 3	Version 4

Table 11.1: Versions for Answering Question 1 and 2 (Simulation Type 1)

Escalation	**Load Balancing**	
	Yes	*No*
Yes	Version 5	Version 6
No	Version 7	Version 8

Table 11.2: Versions for Answering Question 3 (Simulation Type 2)

The results (Sections 13.2 and 14.3.2) of these questions will help to determine the most efficient variant of the model with respect to the chosen performance measures.

11.2 StADy Hierarchy and Artifact Models (Task 2.2)

This section illustrates the application of task 2.2 on the pharmacy case study (Chapter 5). The UML use-case and class diagrams defined in task 1.1 (Section 10.1) are extended into StADy hierarchy model and artifact models, respectively.

The corresponding StADy artifact model and hierarchy models defined in Section 6.3 are extensions of the UML models defined in Section 10.1. Figures 11.1 and 11.2 are a recap of the pharmacy business processes's StADy hierarchy and artifact model.

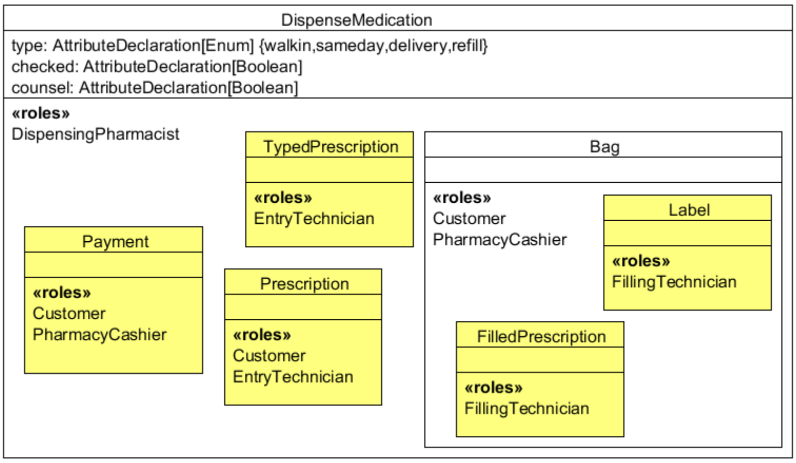


Figure 11.2: StADy Artifact Model

Consistency holds true between the StADy models and the UML models defined in task 1.1 (Section 10.1) because they passed the following consistency tests:

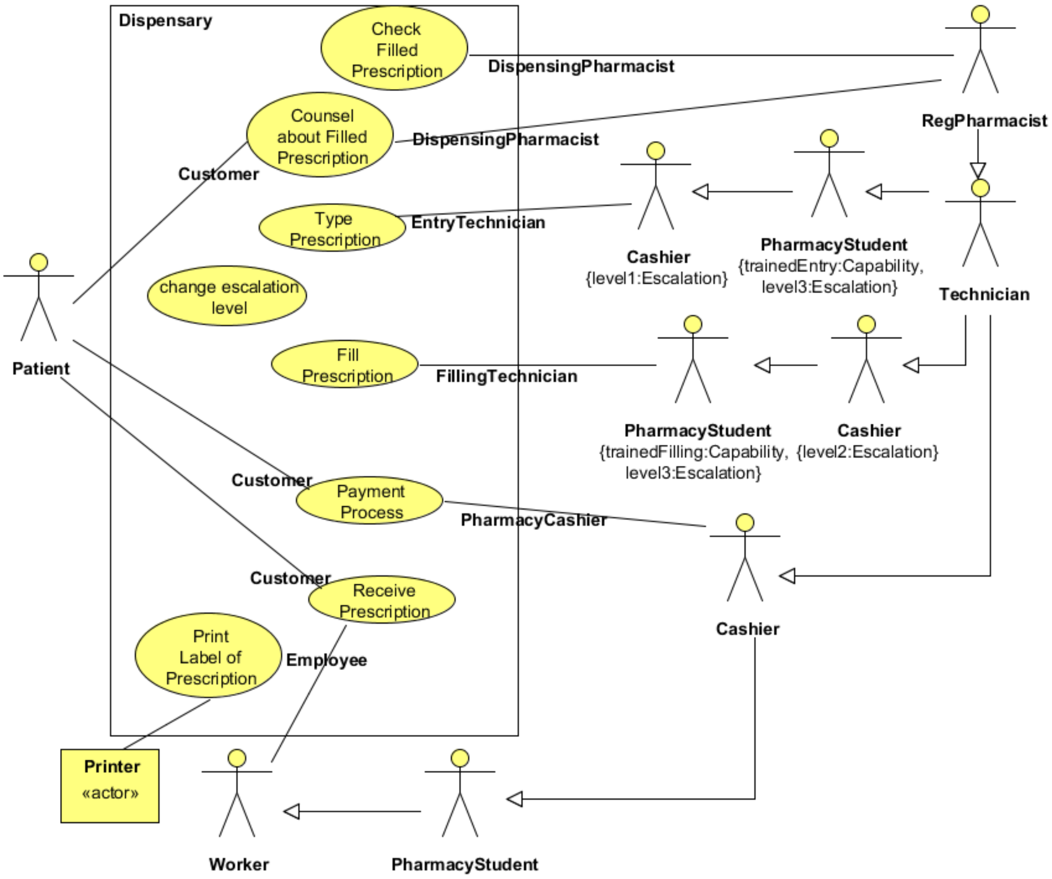


Figure 11.1: StADy Hierarchy Model

1. The StADy Hierarchy Model must contain the original Use-Case Diagram.

   - This holds true because all of the originally specified actor and use-case relations are within the hierarchy model.

2. Each ≪actor≫ stereotype *artifact* association in the original class diagram should correspond to the refined actor–role association, by having the associated role name written under the stereotyped keyword

≪role≫ inside the *artifacts* in the *StADy artifact model.*

- The following relations are consistent in both diagrams as shown in Table 11.3.

≪actor≫	**artifact**	**≪role≫**
Cashier	Payment	PharmacyCashier
Cashier	Bag	PharmacyCashier
Patient	Payment	Customer
Patient	Prescription	Customer
Patient	Bag	Customer
Technician	Prescription	EntryTechnician
Technician	TypedPrescription	EntryTechnician
Technician	Label	FillingTechnician
Technician	FilledPrescription	FillingTechnician
RegPharmacist	DispenseMedication	DispensingPharmacist

Table 11.3: ≪actor≫– artifact – ≪role≫ Consistency Check

3. Each artifact that has composition relation to a Case in the class diagram, should be represented as a box inside the Case in the Artifact Model, which is commonly used to represent containment in DSL.

   - This holds true because all of the six defined artifacts are located within the DispenseMedication case of the Artifact Model.

4. The artifact role access that is defined in the Artifact Model should correspond to Roles defined in the Hierarchy model.

   - This holds true because all of the roles listed in Table 11.3 exist in Figure 11.1.

5. The combination of both models is an instance of the metamodel.

- This relation can be validated by looking at the abstract representation of the combined diagrams (Figure 6.4) in relation to the metamodel (Figure 6.2).

11.3 Domain-Independent GT Rules in StADy notation (Task 2.3)

This section illustrates the application of task 2.3 on the pharmacy case study (Chapter 5). Predefined managerial GT rules are selected based on features captured in the unique simulation models specified in Section 11.1.

Based on Table 11.1 simulation models versions 1-4 (Section 11.1) require the following predefined domain independent managerial global application conditions and GT rules from Section 7.2.

Version 1: Scheduling by deadline (Figure 7.11)

Version 2: Scheduling by priority (Figure 7.12)

Version 3: Assignment policy (Figure 7.10), scheduling by deadline (Figure 7.11)

Version 4: Assignment policy (Figure 7.10), scheduling by priority (Figure 7.12)

All of the 4 versions also require the following rules: assignment (Figure 7.7), unassignment (Figure 7.8), request (Figure 7.6) and clock (Figure 7.13).

11.4 Domain Specific GT Rules in StADy notation (Task 2.4)

This section illustrates the application of task 2.4 on the pharmacy case study (Chapter 5), by defining the domain specific support-level and production-level GT rules. The support-level rules are based on the predefined GT rules in Section 7.3. The human error/unpredictability GT rules are required for all of the simulation model versions. The skip GT rule (Figure 7.17) and backtrack GT rule (Figure 7.18) can be domain specifically defined for each state that is being modelled in the pharmacy business process. On the other hand, the production-level GT rules are defined as follows:

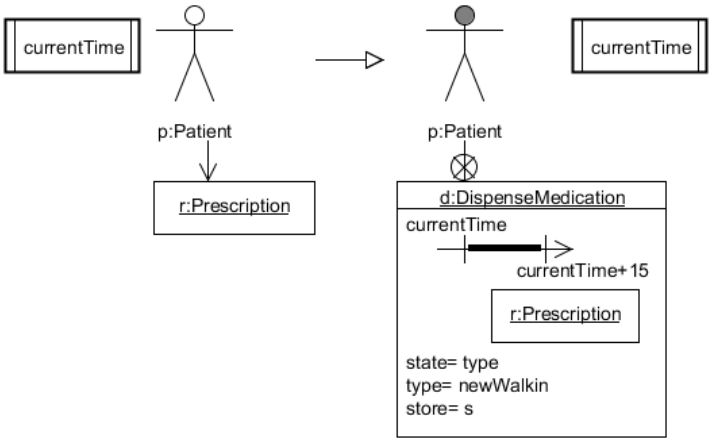


Figure 11.3: Rule Name: New Walk-in Prescription
pre: Patient walks into the store with new prescription.
post: Prescription is added to new case instance with deadline set to 15 mins from current time and the Patient takes on the role of customer and has presence on the new case instance.

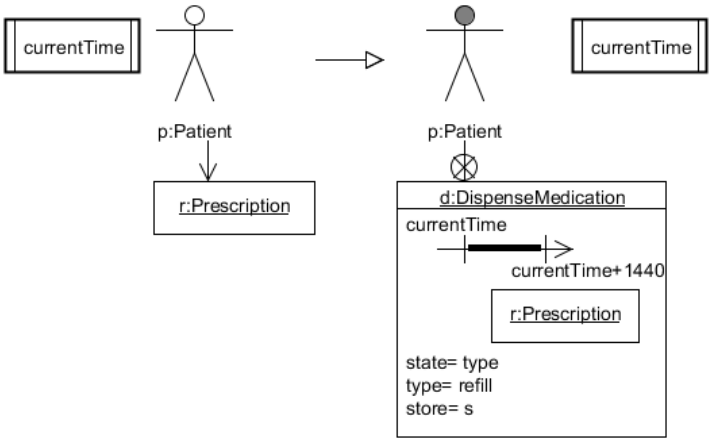


Figure 11.4: Rule Name: New Refill Prescription
pre: Patient submits refill prescription request via phone or web.
post: Prescription is added to new case instance with deadline set to 24 hrs from current time.

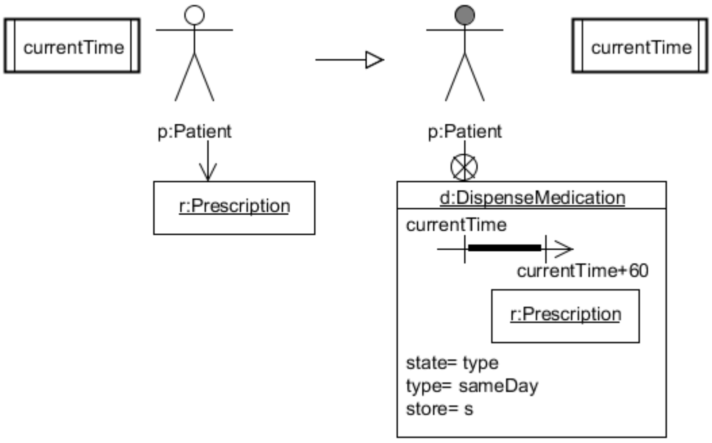


Figure 11.5: Rule Name: New Same Day Prescription
pre: Patient walks into the store with new prescription, but will return to collect the filled prescription later in the day.
post: Prescription is added to new case instance with deadline set to 1 hr from current time.

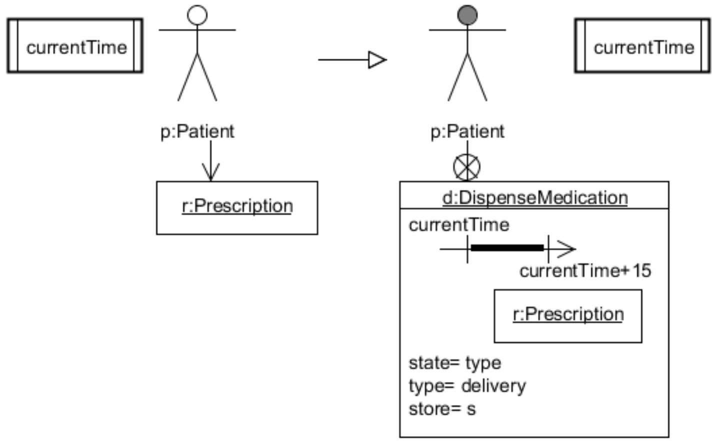


Figure 11.6: Rule Name: New Delivery Prescription Type
pre: Patient submits prescription for delivery.
post: Prescription is added to new case instance with deadline set to 15 minutes from current time.

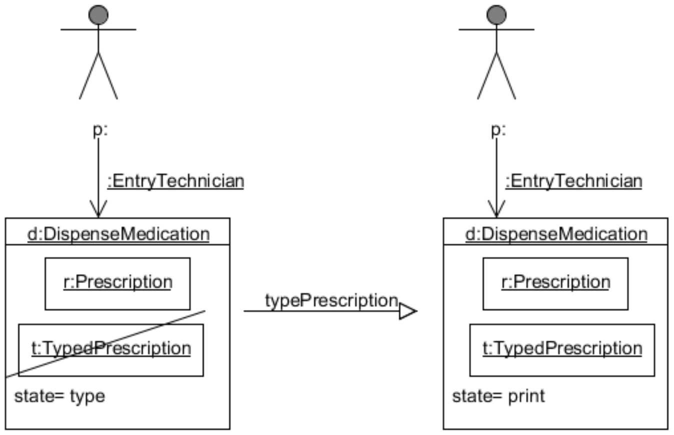


Figure 11.7: Rule Name: Type Prescription
pre: A prescription is not typed into the computer database and there is an entry technician assigned to the case instance.
post: The prescription is typed into the computer database.

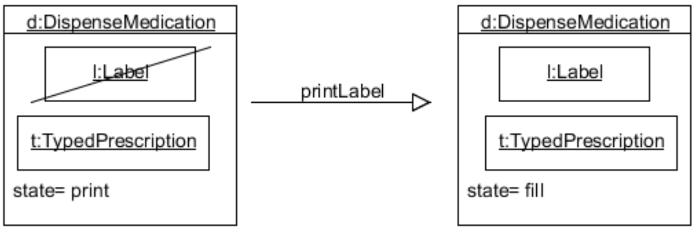


Figure 11.8: Rule Name: Print Label
pre: A typed prescription does not have a bottle label.
post: The prescription label is printed.

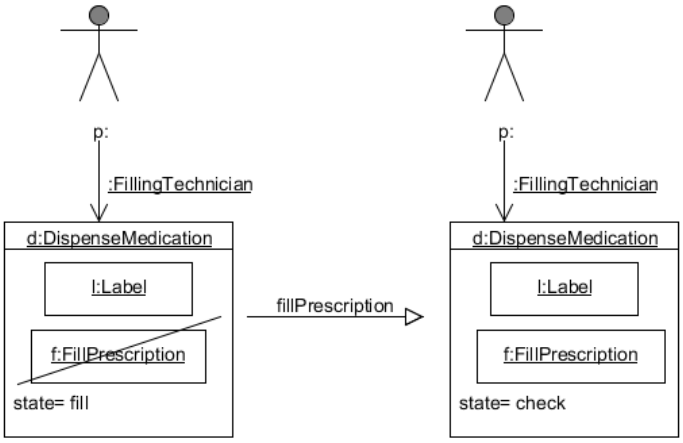


Figure 11.9: Rule Name: Fill Prescription
pre: A prescription is not filled and there is a filling technician assigned to the case instance.
post: The prescription is filled.

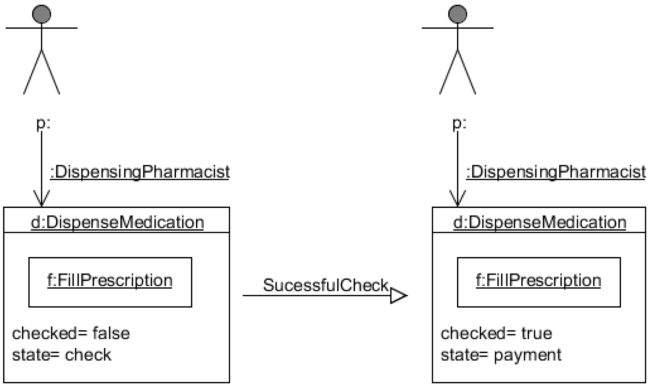


Figure 11.10: Rule Name: Successful Prescription check
pre: A filled prescription has not been checked and there is a dispensing pharmacist assigned to the case instance.
post: The successfully checked filled prescription and label are put into a prescription bag.

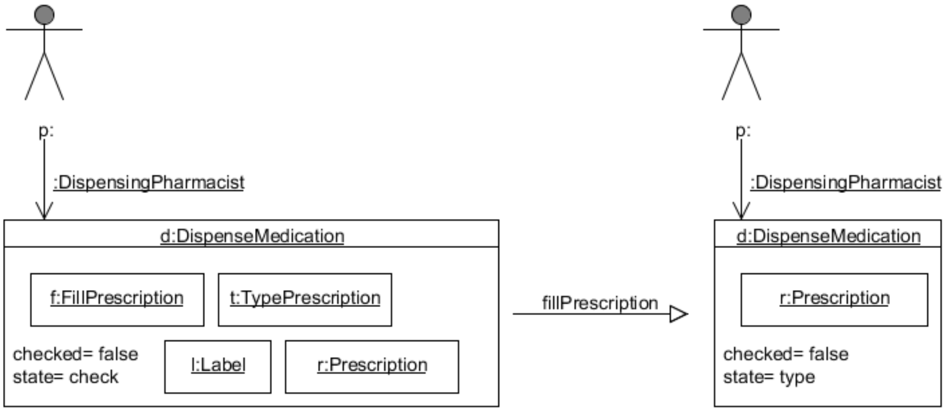


Figure 11.11: Rule Name: Unsuccessful Prescription check
pre: A filled prescription has not been checked and there is a dispensing pharmacist assigned to the case instance.
post: The unsuccessfully checked filled prescription, label, typed prescription are removed from the case, leaving the prescription to be redone.

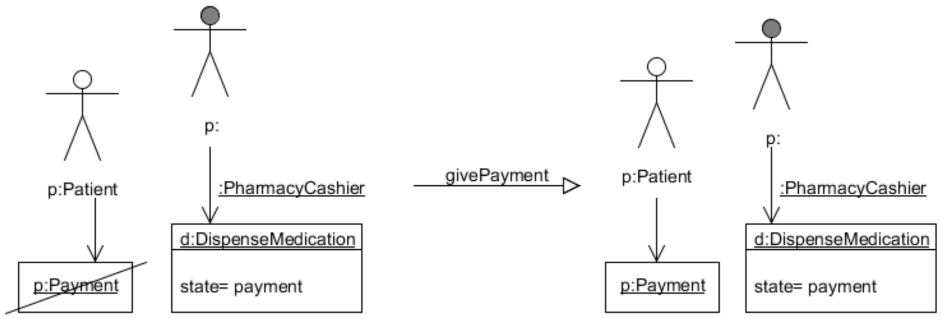


Figure 11.12: Rule Name: Give Payment
pre: Payment not received and there is a pharmacy cashier assigned to a case with a customer's presence.
post: Customers handover payment.

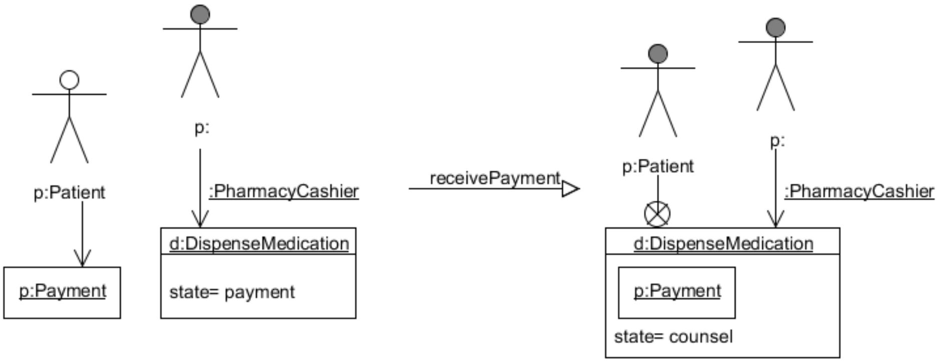


Figure 11.13: Rule Name: Receive Payment
pre: Customer is handing over payment to a pharmacy cashier assigned to the case.
post: Pharmacy Cashier takes payment from customer and puts it in the cash register.

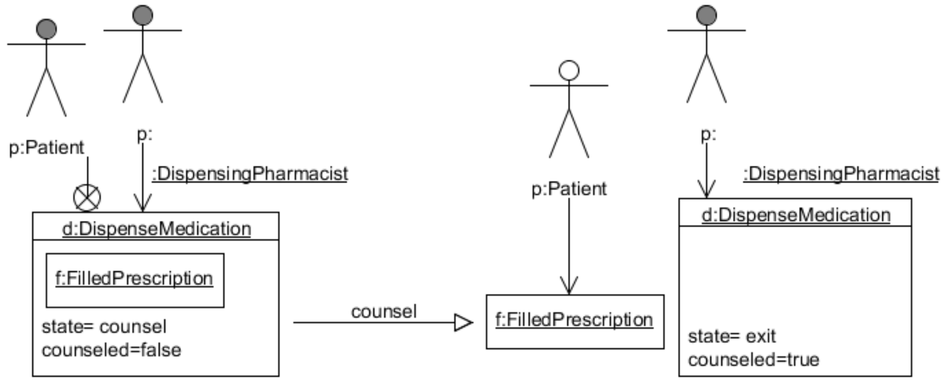


Figure 11.14: Rule Name: Counsel Patient
pre: Customer has not received counseling, but the prescription is ready to be picked up and there is a dispensing pharmacist who is assigned to case instance with the presence of the customer.
post: Pharmacist counsels the customer.

Consistency holds true between the domain specific GT rules & activity diagrams (Section 10.2), GT rules & use-case descriptions (Section 10.1) and between GT rules & StADy metamodel (Section 6.1), because they passed the following consistency tests:

1. Each activity defined in the activity diagram must correspond to one or many GT rules. If the activity corresponds to multiple GT rules, then the preconditions of those rules must be the same.
   - This holds true as shown in Table 11.4.
2. The objects defined as input and output parameters of the activity must also be present in the pre and post conditions of the corresponding rules.
   - This holds true as shown in Table 11.4.

3. Each use-case corresponds to at least one rule and the use-case description informally describes the contents of the pre and post conditions of the rule.
   - This holds true as shown in Table 11.4.
4. The GT Rules must be typed over the StADy metamodel.
   - As illustrated in the abstract representation (Figure 11.15) of print label GT rule (Figure 11.8).

Integration must take place to ensure consistency between the domain specific GT Rules and the activity diagrams [73]. Analysis of consistency across these two models is initially explored by refining the activity diagrams to the rules, as shown in Table 11.4. The table represents one to one correspondence between activity and GT rule. Jurack et al. [37] extended refined activity diagram of object flow, in order to produce partial rule dependencies to formalize the semantics of object flow.

Activity	**Use-Case Description**	***GT Rule***
New Walkin (in:Prescription) (out:DispenseMedication)	Table 10.2	newWalkin (Figure 11.3)
New Delivery (in:Prescription) (out:DispenseMedication)	Table 10.2	newDelivery (Figure 11.6)
New Same Day (in:Prescription) (out:DispenseMedication)	Table 10.2	newSameDay (Figure 11.5)
New Refill (in:Prescription) (out:DispenseMedication)	Table 10.2	newRefill (Figure 11.4)
Type into Computer (in:DispenseMedication) (out:TypedPrescription)	Table 10.5	typePrescription (Figure 11.7)
Print Label (in:TypedPrescription) (out:Label)	Table 10.8	printLabel (Figure 11.8)
Fill Prescription (in:Label) (out:FilledPrescription)	Table 10.11	fillPrescription (Figure 11.9)
Unsuccessful Check (in:FilledPrescription) (out:DispenseMedication)	Table 10.15	unsuccessfulCheck (Figure 11.11)
Successful Check (in:FilledPrescription) (out:FilledPrescription)	Table 10.15	successfulCheck (Figure 11.10)
Give Payment (in:FilledPrescription) (out:Payment)	Table 10.18	givePayment (Figure 11.12)
Receive Payment (in:Payment) (out:FilledPrescription)	Table 10.18	receivePayment (Figure 11.13)
Counsel Customer (in:FilledPrescription)	Table 10.21	counsel (Figure 11.14)

Table 11.4: Activity Diagram Refined into GT Rules

The following steps are required to show that a GT rule is typed over the StADy metamodel:

1. Translate the GT rule from concrete syntax to abstract syntax as illustrated in Figure 11.15 for the print label GT rule defined in Figure 11.8.

2. Check that every object, attribute, and association corresponds to the metamodel defined in Figure 6.2. The Figure 11.15 contains the StADy metamodel elements: Process, State, ArtifactType, Artifact, Case and StateInstance.

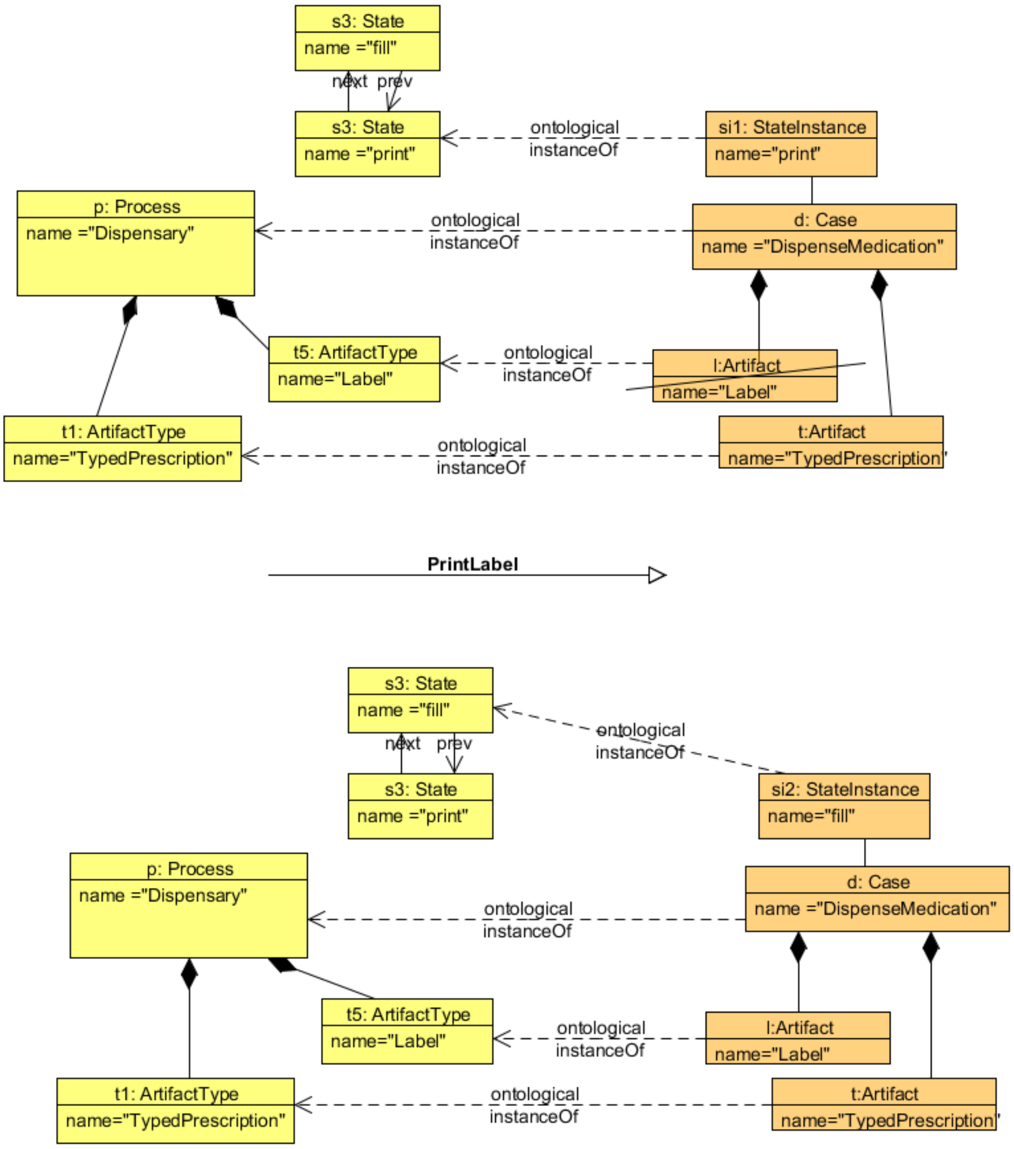


Figure 11.15: Abstract Representation of Print Label Rule

11.5 Probes and Observation Rules in StADy notation (Task 2.5)

This section illustrates the application of task 2.5 on the pharmacy case study (Chapter 5), by defining probes to aid in solving performance question 1-2 which were defined in Section 1.2. This task also uses versions 1-4 simulation models that were specified in task 2.1 in Section 11.1.

Question #1: Which scheduling method (by priority/deadline) increases the probability for a prescription to be completed on time? How does it affect the number of cases completed out of the total received?

Two probe rules can be used to answer the first half of question 1, 'latefromtotal' and 'lateCases'. The 'lateFromTotal' probe calculates the numbers of cases that are late from the total number received, whereas 'lateCases' counts the number of late cases, as shown in Figure 11.16. In order to answer the second half of question 1, the batch rule reports need to be analyzed in order to calculate the number of times the 'successfulCheck' rule was applied in relation to the sum of the execution of the new case rules. Therefore, the answer is gathered from the comparison of version 1-4 (number of successful prescription checks divided by the total number of new cases received).

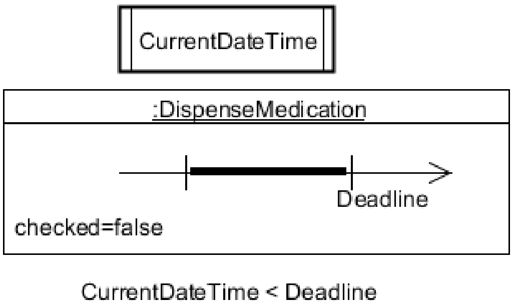


Figure 11.16: LateCase Probe

Question #2: Does the usage of assignment policies decrease the number of idle workers (not assigned to jobs) at any point in time?

The second question can be answered by defining a probe to count the number of idle workers (‘PersonAvailable’ probe), then in each intermediate simulation step the total number of idle workers are counted using the probe defined in Figure 11.17. These data are used to determine the average number of idle workers in a simulation. The probe rule can be used to answer question 2 by analyzing the smallest probe result in version 1-4.

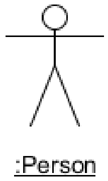


Figure 11.17: PersonAvailable Probe

Consistency holds true between the probe/observation rules and StADy metamodel (Section 6.1), because they passed the following consistency test:

The probe/obervation rules must be typed over the StADy metamodel: As illustrated in the abstract representation (Figure 11.18) of *PersonAvailable* probe (Figure 11.17).

The following steps are required to show that a probe/obervation rule is typed over the StADy metamodel:

1. Translate the probe/obervation rule from concrete syntax to abstract syntax as illustrated in Figure 11.18 for the *PersonAvailable* probe defined in Figure 11.17.

2. Check that every object, attribute, and association corresponds to the metamodel defined in Figure 6.2. The Figure 11.18 contains the StADy metamodel elements: Actor and Person. The Boolean *free* attribute in the *Person* object is set to true.

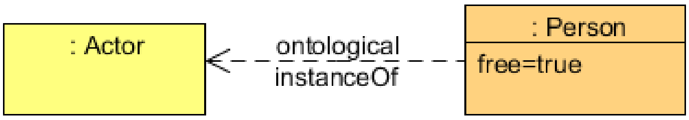


Figure 11.18: Abstract Representation of PersonAvailable Probe

11.6 Start-Graph in StADy DSL (Task 2.6)

This section illustrates the application of task 2.6 on the pharmacy case study (Chapter 5), by specifying the start graph. The initial model (start graph) has one registered pharmacist (Cindy), three technicians (Bob, Fred, Donald), one cashier (Gina) and one active dispense medication case present. All of the workers are currently available to take on jobs. A patient by the name of Emily is waiting for her prescription to be filled. The current clock time is 9:01 am on July 25th, 2010 and Emily's prescription case arrived earlier in the day at 8:50 am and the expected completion time was set to the same day at 9:05 am. This start graph is visually represented in Figure 11.19.

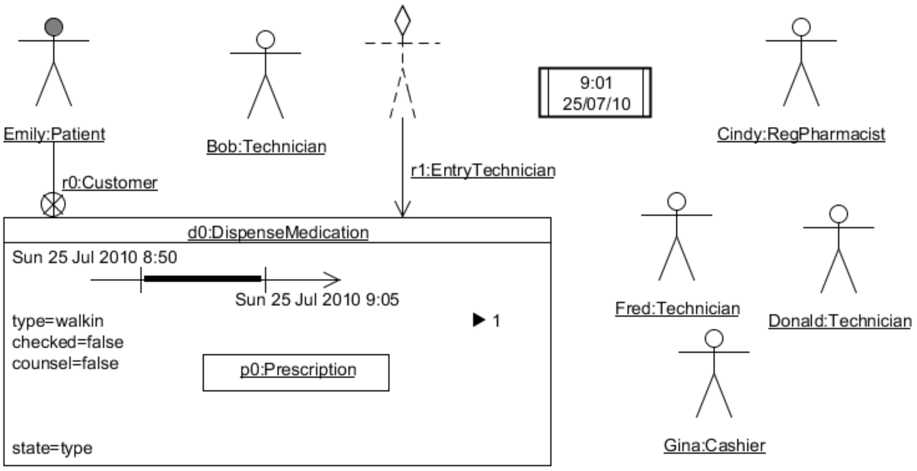


Figure 11.19: Start Graph for Simulation Type 1 Model

Consistency holds true between the start graph and StADy metamodel (Section 6.1), because it passed the following consistency test:

The start graph must be typed over the StADy metamodel: As illustrated in the abstract representation (Figure 11.20) of the simulations start graph (Figure 11.19).

The following steps are required to show that the start graph is typed over the StADy metamodel:

1. Translate the start graph from concrete syntax to abstract syntax as illustrated in Figure 11.20.

2. Check that every object, attribute, and association corresponds to the metamodel defined in Figure 6.2. The Figure 11.20 contains all of the StADy metamodel elements. This diagram is similar to the M1 abstract model discussed in Section 6.2, except the O0 side represents the model defined in Figure 11.19.

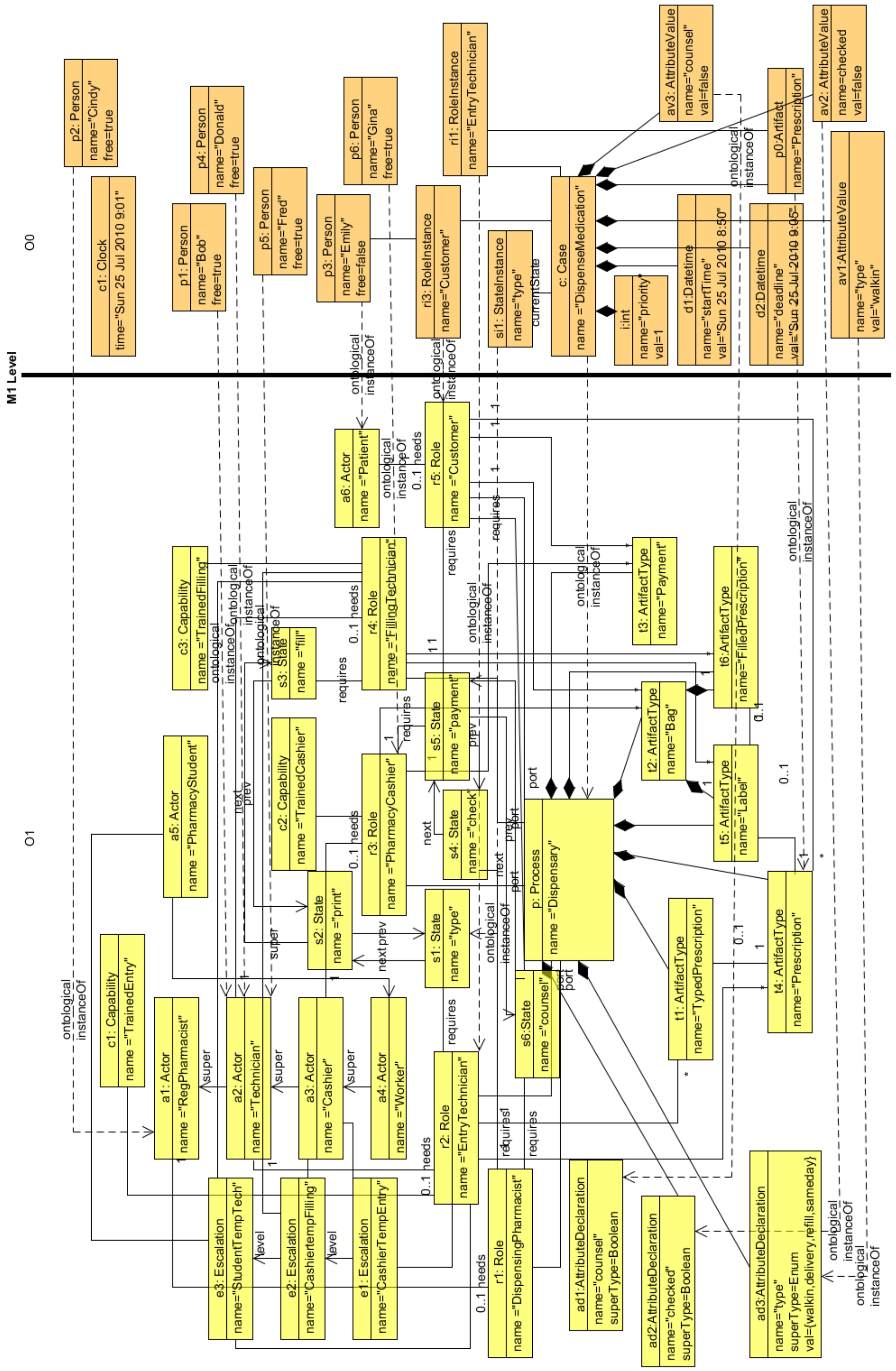


Figure 11.20: Abstract Syntax of Start Graph

11.7 Stochastic Graph Transformation System (Task 2.7)

This section illustrates the application of task 2.7 on the pharmacy case study (Chapter 5), by corresponding rate, mean and variance parameters for the GT rules specified in tasks 2.3-2.4 (Sections 11.3-11.4). The distribution parameters are based on measured pharmacy process data from [61, 69, 60] and discussions with Georgina Donyina, Pharmacist/Owner of a Shoppers Drug Mart pharmacy franchise located in Canada [15].

Distribution data were gathered from a dispensing service pilot project [69], which took place from 27 November 2005 to 4 December 2005. The sample size of the project consisted of 29 pharmacies. The dispensing procedure was separated into three phases. The first phase involved entry and evaluation (checking) of the prescription. The second phase involved filling, labelling and checking filled prescription. The third phase involved counseling the patient. Table 11.5 displays the average time spent per prescription in each phase, which resulted in the average total time of dispensing a prescription to be 16 minutes and 59 seconds with a standard deviation of 8 minutes and 29 seconds.

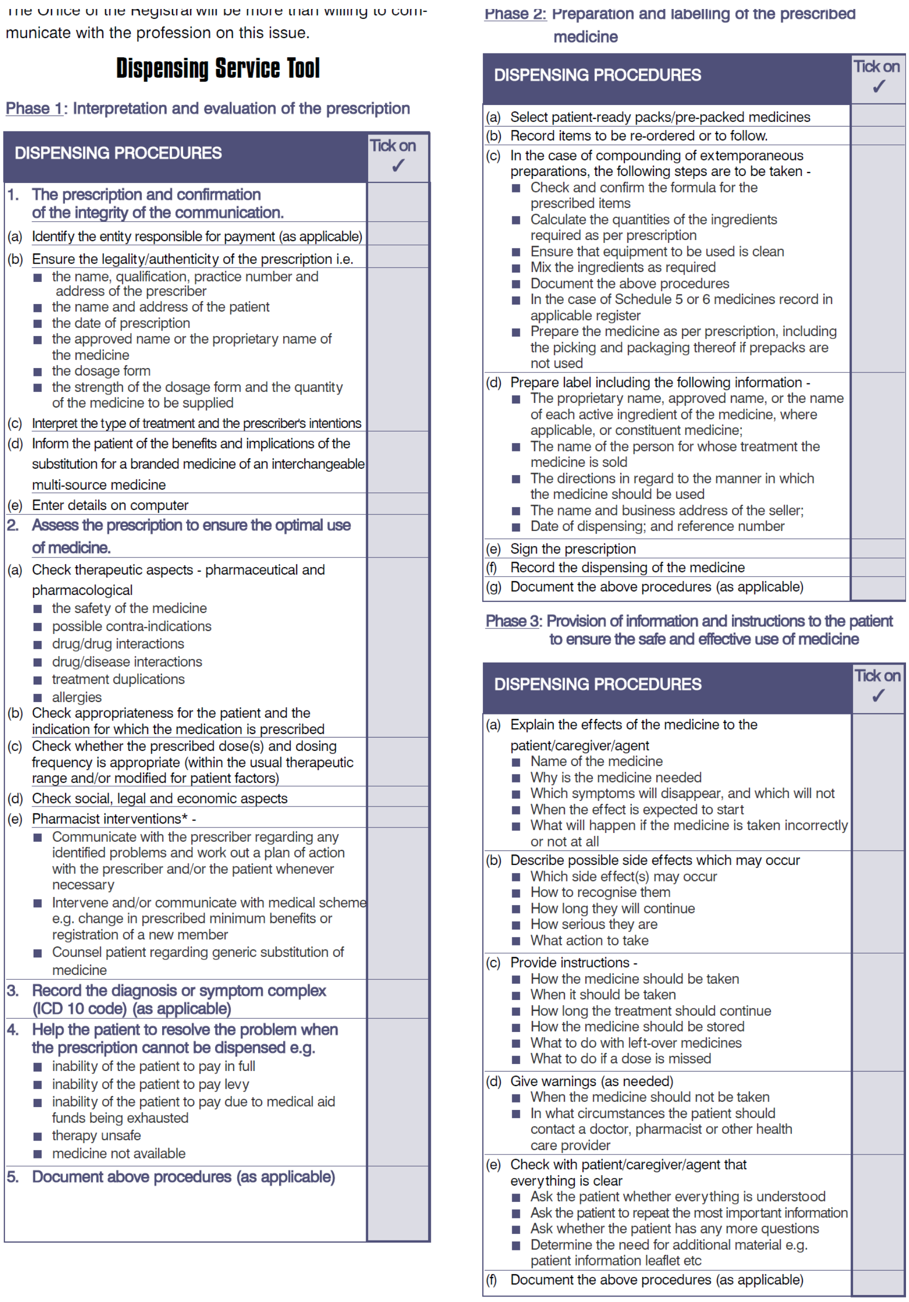
Since the sample data from the pilot project [69] were not available the pharmacy business process activity distributions could not be selected using distributing fitting [62]. The pharmacy business activities were categorized based on distinct distribution properties and theories. The activities that had a memoryless property [7] were categorized as an exponential distribution, whereas activities that captured the central limit theorem [76] were catego-

Phases	**Average time (min spent per prescription**	**Minimum (min)**	**Maximum (min)**
Phase 1	9 min 11 sec (± 5 min 37 sec)	1 sec	34 min 12 sec
Phase 2	4 min 37 sec (± 4 min 51 sec)	15 sec	40 min 54 sec
Phase 3	3 min 9 sec (± 2 min 34 sec)	4 sec	14 min 10 sec
Total	16 min 59 sec (± 8 min 29 sec)	3 min 23 sec	54 min 10 sec

Table 11.5: Average Time Spent per Prescription [69]

rized as normal distribution as discussed in Section 4.3. The memoryless property implies that the future of a random variable is independent of its past such as the arrival of new prescription cases. The new prescription GT rules in Figures 11.3-11.6 were classified as exponential distributions because the rate of arrivals for a customer is independent to the time it takes for the next customer to arrive. On the other hand, the central limit theorem classifies activities as normal distributions if it is an average of independent random variable. The remaining pharmacy business process actions (Figures 11.7-11.14) were classified as normal distributions because it generally takes an average amount of time to complete a dispensary task.

The descriptions of the pharmacy business process in Chapter 5 were mapped to the detailed dispensing procedure used in the pilot study (Figure 11.21). With the help from the pharmacy professional I concluded my distribution parameter estimates as shown in Table 11.6. Each state outlined in Section 7.1 corresponds to an event, which is the pattern match of a defined GT Rule.



The Office of the Registrar will be more than willing to communicate with the profession on this issue.

Dispensing Service Tool

Phase 1: Interpretation and evaluation of the prescription

DISPENSING PROCEDURES	Tick on ✓
1. The prescription and confirmation of the integrity of the communication.	
(a) Identify the entity responsible for payment (as applicable)	
(b) Ensure the legality/authenticity of the prescription i.e.	
■ the name, qualification, practice number and address of the prescriber ■ the name and address of the patient ■ the date of prescription ■ the approved name or the proprietary name of the medicine ■ the dosage form ■ the strength of the dosage form and the quantity of the medicine to be supplied (c) Interpret the type of treatment and the prescriber's intentions	
(d) Inform the patient of the benefits and implications of the substitution for a branded medicine of an interchangeable multi-source medicine	
(e) Enter details on computer	
2. Assess the prescription to ensure the optimal use of medicine.	
(a) Check therapeutic aspects - pharmaceutical and pharmacological ■ the safety of the medicine ■ possible contra-indications ■ drug/drug interactions ■ drug/disease interactions ■ treatment duplications ■ allergies (b) Check appropriateness for the patient and the indication for which the medication is prescribed	
(c) Check whether the prescribed dose(s) and dosing frequency is appropriate (within the usual therapeutic range and/or modified for patient factors)	
(d) Check social, legal and economic aspects	
(e) Pharmacist interventions* -	
■ Communicate with the prescriber regarding any identified problems and work out a plan of action with the prescriber and/or the patient whenever necessary ■ Intervene and/or communicate with medical scheme e.g. change in prescribed minimum benefits or registration of a new member ■ Counsel patient regarding generic substitution of medicine	
3. Record the diagnosis or symptom complex (ICD 10 code) (as applicable)	
4. Help the patient to resolve the problem when the prescription cannot be dispensed e.g. ■ inability of the patient to pay in full ■ inability of the patient to pay levy ■ inability of the patient to pay due to medical aid funds being exhausted ■ therapy unsafe ■ medicine not available	
5. Document above procedures (as applicable)	

Phase 2: Preparation and labelling of the prescribed medicine

DISPENSING PROCEDURES	Tick on ✓
(a) Select patient-ready packs/pre-packed medicines	
(b) Record items to be re-ordered or to follow.	
(c) In the case of compounding of extemporaneous preparations, the following steps are to be taken - ■ Check and confirm the formula for the prescribed items ■ Calculate the quantities of the ingredients required as per prescription ■ Ensure that equipment to be used is clean ■ Mix the ingredients as required ■ Document the above procedures ■ In the case of Schedule 5 or 6 medicines record in applicable register ■ Prepare the medicine as per prescription, including the picking and packaging thereof if prepacks are not used	
(d) Prepare label including the following information - ■ The proprietary name, approved name, or the name of each active ingredient of the medicine, where applicable, or constituent medicine; ■ The name of the person for whose treatment the medicine is sold ■ The directions in regard to the manner in which the medicine should be used ■ The name and business address of the seller; ■ Date of dispensing; and reference number	
(e) Sign the prescription	
(f) Record the dispensing of the medicine	
(g) Document the above procedures (as applicable)	

Phase 3: Provision of information and instructions to the patient to ensure the safe and effective use of medicine

DISPENSING PROCEDURES	Tick on ✓
(a) Explain the effects of the medicine to the patient/caregiver/agent ■ Name of the medicine ■ Why is the medicine needed ■ Which symptoms will disappear, and which will not ■ When the effect is expected to start ■ What will happen if the medicine is taken incorrectly or not at all	
(b) Describe possible side effects which may occur ■ Which side effect(s) may occur ■ How to recognise them ■ How long they will continue ■ How serious they are ■ What action to take	
(c) Provide instructions - ■ How the medicine should be taken ■ When it should be taken ■ How long the treatment should continue ■ How the medicine should be stored ■ What to do with left-over medicines ■ What to do if a dose is missed	
(d) Give warnings (as needed) ■ When the medicine should not be taken ■ In what circumstances the patient should contact a doctor, pharmacist or other health care provider	
(e) Check with patient/caregiver/agent that everything is clear ■ Ask the patient whether everything is understood ■ Ask the patient to repeat the most important information ■ Ask whether the patient has any more questions ■ Determine the need for additional material e.g. patient information leaflet etc	
(f) Document the above procedures (as applicable)	

Figure 11.21: Dispensing used in the Dispensing Service Pilot Project (3 Phase Breakdown) [69]

Domain Specific GT Rules

State	**Event**	**Mean** μ **(min)**	**Variance** σ^2 **(min)**
Type	TypePrescription GT Rule	1.25	0.4375
Print	PrintPrescription GT Rule	0.26	0.024806
Fill	FillPrescription GT Rule	1.17	0.395833
Check	SuccessfulCheck and UnsuccessfulCheck GT Rule	2.00	0.8125
Payment	GivePayment GT Rule	0.71	0.047292
Counsel	Counsel GT Rule	1.17	0.583333

Table 11.6: Estimated Average Times to Complete States

Tables 11.7 to 11.9 present the estimated average times for the following exception events: skip, backtrack, escalation trigger and transfer (load balancing). Each of the distributions were influenced by the estimated average times to complete the states in the pharmacy business process; for instance the mean of the skip events were calculated to be 0.25 factor higher than their corresponding fulfilled action distribution defined in Table 11.6 (μ_{skip}=$\mu_{fulfilled}$+0.25). On the other hand, the backtrack event means were set to the same mean as the fulfilment of the state defined in Table 11.6 because the assumption is made that it will take the same amount of time to detect an error in a case as it would take to perform the action. Similarly, the optional load balancing and escalation handling features are partially influenced by the other determined estimations.

Event	**Mean** μ **(min)**	**Variance** σ^2 **(min)**
SkipTypePrescription GT Rule	1.50	0.4375
SkipPrintPrescription GT Rule	0.51	0.024806
SkipFillPrescription GT Rule	1.42	0.395833
SkipSuccessfulCheck and SkipUnsuccessfulCheck GT Rule	2.25	0.8125
SkipGivePayment GT Rule	0.96	0.047292
SkipCounsel GT Rule	1.42	0.583333

Table 11.7: Estimated Average Times for Skip Events

Event	**Mean** μ **(min)**	**Variance** σ^2 **(min)**
Backtrack_PrintState GT Rule	0.26	0.024806
Backtrack_FillState GT Rule	1.17	0.395833
Backtrack_CheckState and	2	0.8125
Backtrack_PaymentState GT Rule	0.71	0.047292
Backtrack_CounselState GT Rule	1.17	0.583333

Table 11.8: Estimated Average Times for Backtrack Events

Event	**Description**	**Mean** μ **(min)**	**Variance** σ^2 **(min)**
DisTypePrescription GT Rule	Transfer Prescription to another store	1.25	0.4375
TempAssign GT Rules	Temporarily assign person to a role	0.05	0.004537
Trigger GT Rules	Trigger change in escalation levels	0.09	0.004537

Table 11.9: Estimated Average Times for Optional Features

The arrival of new prescription cases was categorized as an exponential distribution, because the random arrival of prescription orders is independent of other actions. The estimation values were influenced by data collection from a medium-sized New Jersey Hospital pharmacy unit [60]. Their results stated that the mean arrival rate for high priority orders was 0.096, whereas the mean arrival rate for regular orders was 0.259. After discussions with pharmacy professionals estimations shown in Table 11.10 were decided.

Event	**Frequency**	**Rate λ (mins)**	**Mean arrival rate λ^{-1}**
NewCase GT Rule	Every 5 mins	10	0.1
NewCaseMedPriority GT Rule	Every 10 mins	20	0.05
NewCaseHighPriority GT Rule	Every 30 mins	60	0.01667

Table 11.10: GT Rule (Exponential Distribution)

Managerial GT Rules

Managerial Rules	Mean (mins)	Variance
ClockTick	1.00	0.000055778
Request	0.09	0.004537
Unassignment	0.3125	0.048032407
Assignment	0.09	0.004537

Table 11.11: Estimated Distributions of Managerial GT Rules

The events that correspond to managerial GT rules were categorized as lognormal distributions because an action such as assignment is dependent on the request for assignment action. Since the domain specific actions only represented a proportion of the total dispensing time specified by the pilot research data in Table 11.5, the remaining dispensing time is covered by the managerial actions as defined in Table 11.11. The distribution values were selected using partial trial and error in order to align the results with measured pharmacy process statistics from [61, 69, 60], because after the values were randomly selected they were evaluated to ensure they represented meaningful data based on the statistics of these research.

Consistency holds true between the GT Rules defined in tasks 2.3-2.4 (Sections 11.3-11.4) and distribution parameters, because it passed the following consistency test:

Each defined GT Rule must have a distribution: This holds true because the distribution parameters for the GT Rules are defined in Tables 11.6-11.11.

11.8 Summary

This chapter illustrated the business process execution design tasks in stage 2 of the StADy methodology (Chapter 9). This stage extended UML use-case and class diagrams that were defined in stage 1 (Chapter 10) to represent additional information in terms of access control, escalation handling and capabilities. Also the StADy language (Chapter 6-7) was used to define the business process using graph transformation. The next chapter (Chapter 12) will illustrate how to encode the domain-specific StADy elements from this stage into VIATRA2 syntax.

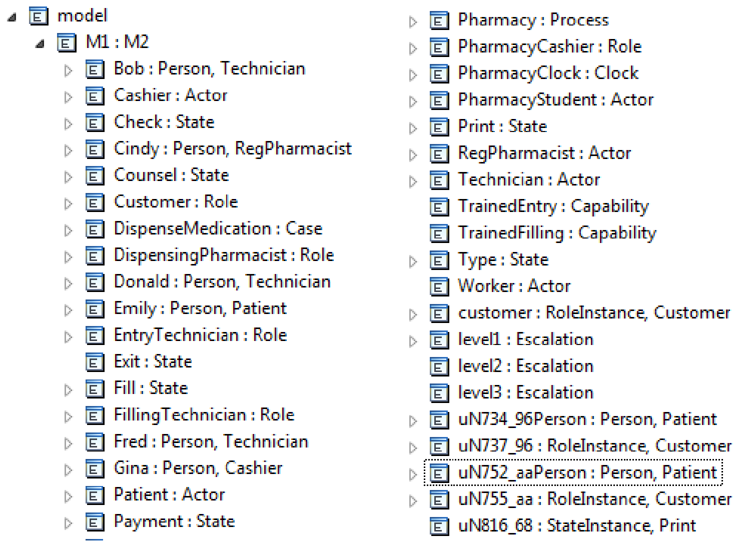
Chapter 12

Illustration of Stage 3 – Process Encoding

This chapter illustrates the tasks in stage 3 of the methodology (Chapter 9). Each task includes consistency checks and a corresponding illustration based on the pharmacy case study defined in Chapter 5. The process-encoding stage is for encoding the start graph from task 2.6 (Section 11.6), GT rules from tasks 2.3-2.4 (Section 11.3-11.4) and probes from task 2.5 (Section 11.5) into the VIATRA2 tool, in order to perform stochastic simulation on Graph-based Stochastic Simulation (GraSS) tool [30, 68] which extends VIATRA2. Chapter 8 discussed and illustrated the mapping from StADy concrete syntax to VIATRA2 textual syntax.

12.1 Start Graph in VIATRA2 Model Space (Task 3.1)

This section illustrates the application of task 3.1 on the pharmacy case study (Chapter 5), by translating the start graph that was designed using StADy syntax in task 2.6 (Section 11.6) into the VIATRA2 (VPM) Model Space [72]. Figure 12.1 is a screen shot of the start graph in VPM.



```
model
  M1 : M2
    Bob : Person, Technician
    Cashier : Actor
    Check : State
    Cindy : Person, RegPharmacist
    Counsel : State
    Customer : Role
    DispenseMedication : Case
    DispensingPharmacist : Role
    Donald : Person, Technician
    Emily : Person, Patient
    EntryTechnician : Role
    Exit : State
    Fill : State
    FillingTechnician : Role
    Fred : Person, Technician
    Gina : Person, Cashier
    Patient : Actor
    Payment : State
```

(a) Part 1

```
Pharmacy : Process
PharmacyCashier : Role
PharmacyClock : Clock
PharmacyStudent : Actor
Print : State
RegPharmacist : Actor
Technician : Actor
TrainedEntry : Capability
TrainedFilling : Capability
Type : State
Worker : Actor
customer : RoleInstance, Customer
level1 : Escalation
level2 : Escalation
level3 : Escalation
uN734_96Person : Person, Patient
uN737_96 : RoleInstance, Customer
uN752_aaPerson : Person, Patient
uN755_aa : RoleInstance, Customer
uN816_68 : StateInstance, Print
```

(b) Part 2

Figure 12.1: Start Graph for Type 1 Simulation Model

Consistency holds true between the start graph defined in StADy syntax (task 2.6- Section 11.6) and the start graph defined in VIATRA2 model space, because it passed the following consistency test: *The start graph in VPM element entities and relations must correspond to the entities and relations found in the abstract syntax representation of the graphical syntax:* This holds true because the abstract representation of the start graph shown in

Figure 11.20 is consistent with the start graph defined in the VIATRA2 model space, i.e., each element corresponds to an object in the model.

12.2 Domain-Independent GT Rules in VIATRA2 Syntax (Task 3.2)

This section illustrates the application of task 3.2 on the pharmacy case study (Chapter 5), by inserting the required managerial GT rules from Appendix A.1 into a VTML file, as selected in task 2.3 (Section 11.3).

12.3 Domain-Specific GT Rules in VIATRA2 Syntax (Task 3.3)

This section illustrates the application of task 3.3 on the pharmacy case study (Chapter 5), by translating the domain specific rules defined in task 2.4 (Section 11.4) into VIATRA2 textual syntax (Appendix A.3). An overview of the defined production-level GT rules is presented.

The textual representation of production-level domain specific rules take three standard formats. The following list categories which GT rules fit into each of the formats.

1. Fill, print, payment, type

2. Counsel, successful check, unsuccessful check

3. New walkin/delivery/sameday/refill case

Each of the three code standard formats are defined. Since within each subgroup they are only distinguishable from each other by their distinct string patterns, the following <PATTERN-NAME> notation is used within the VTML code, with a corresponding table to define the distinct string patterns.

Standard Format 1

The following VTML code corresponds to fill, print, payment and type GT rules defined in task 2.4 (Section 11.4). Table 12.1 outlines the string patterns located in the code, whereas Figure 12.2 illustrates the consistency with the StADy concrete notation.
<RULE>: GT rule name (String type).
<NAC-ARTIFACT>: Artifact name which do not want present (String type).
<ARTIFACT>: Artifact name which want present (String type).
<STATE>: Current state (String type).

<RULE> & **<STATE>**	**<NAC-ARTIFACT>**	**<ROLE>**	**<ARTIFACT>**
Fill (Chapter 7)	FilledPrescription	FillingTechnician	Label
Print	Label		TypedPrescription
Type	TypedPrescription	EntryTechnician	Prescription
Payment	Payment	PharmacyCashier Customer	

Table 12.1: Unique Procedural GT Rule String Patterns

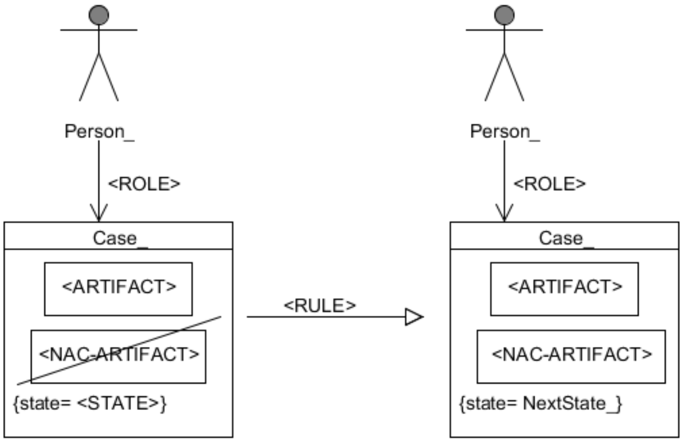


Figure 12.2: Standard Format 1: StADy GT Rule Structure

```
gtrule Rule_<RULE>() =
   {
        precondition pattern lhs(Case_,RoleInstance_,Role_,Person_,ArtifactType_,
                                StateInstance_, NextState_,R6_) =
           {
               //Declarations i.e. Type(Instance)
                Case(Case_);
                ArtifactType(ArtifactType_);
                State(State_);
                State(NextState_);
                StateInstance(StateInstance_);

               //Verify that New ArtifactType matches
                check(name(ArtifactType_)=="<ARTIFACT-NAC>");

               //Verify that required role(s) are assigned to the Case
                find <ROLE>assigned (Case_,RoleInstance_,Role_,Person_);

               //Negative application condition
               //<ARTIFACT-NAC> is not contained in the Case
                 neg find <ARTIFACT-NAC> Exist(Case_);

               //Verify that <ARTIFACT>  in the Case
                 find <ARTIFACT>Exist(Case_);

               //Retrieve Case's current State
                 Case.currentState(R6_,Case_,StateInstance_);

               //Find the next state from the current state
                 State.next(R7,State_,NextState_);

               //Check if current state is the required state
                 typeOf(State_,StateInstance_);
                 check(name(State_)=="<STATE>");
           }
             action {
```

```
       let
        NewArtifact_=undef,
        R1_=undef,
        NewStateInstance_=undef,
        R2_=undef

     in seq {
      //Remove Case's old state
       delete(StateInstance_);
       delete(R6_);

      //Change Case state to the next state
      // create Class inside the Case
       new(StateInstance(NewStateInstance_)in DSM.model.M1);
      //create instanceOf relation with the StateInstance:State
       new(instanceOf(NewStateInstance_,NextState_));
       new (Case.currentState(R2_,Case_,NewStateInstance_));

      //Add Filled Prescription Artifact to the Case
      // create Class inside the Case
       new(Artifact(NewArtifact_)in Case_);
      //create instanceOf relation with the Artifact:ArtifactType
       new(instanceOf(NewArtifact_,ArtifactType_));
      //create relation between Case and Artifact
       new (Case.contains(R1_,Case_,NewArtifact_));

    }

   }
  }

//<ROLE>Assigned Pattern
 pattern <ROLE>assigned (Case_,RoleInstance_,Role_,Person_) =
{
    //Declarations
     Person(Person_);
     Case(Case_);
     Role(Role_);
     RoleInstance(RoleInstance_);

    //Verify that role type is <ROLE>
     check (name(Role_)=="<ROLE>");

    //Verify that RoleInstance_ is of the same <ROLE> role type
     typeOf(Role_, RoleInstance_);

    //Verify that this RoleInstance exists on the Case
     RoleInstance.presence(Rel2, RoleInstance_, Case_);

    //Verify that a person is assigned to the RoleInstance_
    //i.e a person is assigned a RoleInstance_ of <ROLE>
    //which is associated the Case_
     RoleInstance.assignedTo(Rel,RoleInstance_,Person_);
}


//Case contains an instance of <NAC-ARTIFACT>:ArtifactType
 pattern <NAC-ARTIFACT>Exist(Case_,Artifact_, ArtifactType_)=
  {
    //declarations
     Artifact(Artifact_);
     ArtifactType(ArtifactType_);
     Case(Case_);

    //Verify that Case_ contains an Artifact_ of <NAC-ARTIFACT> type
     Case.contains(Rel, Case_, Artifact_);
     typeOf(ArtifactType_,Artifact_);
     check (name(ArtifactType_)=="<NAC-ARTIFACT>");
  }


  //Case contains an instance of  <ARTIFACT>:ArtifactType
   pattern <ARTIFACT>Exist (Case_)=
  {
    //declarations
     Artifact(Artifact_);
     ArtifactType(ArtifactType_);
     Case(Case_);

    //Verify that Case_ contains an Artifact_ of <ARTIFACT> type
     Case.contains(Rel, Case_, Artifact_);
     typeOf(ArtifactType_,Artifact_);
     check (name(ArtifactType_)==" <ARTIFACT>");
  }
```

Standard Format 2

The following VTML code corresponds to counsel, successful check and unsuccessful check GT rules defined in task 2.4 (Section 11.4). Table 12.2 outlines the string patterns located in the code, whereas Figure 12.3 illustrates the consistency with the StADy concrete notation.

<RULE>: GT rule name (String type).

<ATTRIBUTEDECLARATION>: Name of AttributeDeclaration of interest (String type).

<ROLE>: Role name (String type).

<STATE>: Current state (String type).

<ARTIFACT>: Artifact name which want present (String type).

<RULE>	**<ATTRIBUTE-DECLARATION>**	**<ROLE>**	**<STATE>**	**<ARTIFACT>**
Successful-Check	checked	Dispensing-Pharmacist	check	FilledPrescription
Unsuccessful-Check	checked	Dispensing-Pharmacist	check	FilledPrescription TypedPrescription Prescription Label
Counsel	counseled	Dispensing-Pharmacist &Customer	counsel	FilledPrescription

Table 12.2: Unique Procedural GT Rule String Patterns

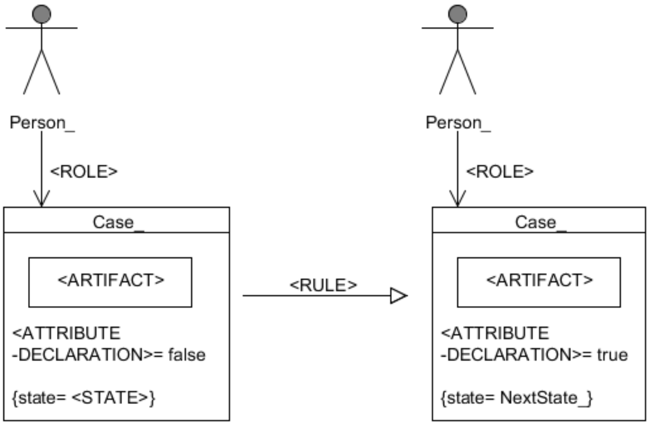


Figure 12.3: Standard Format 2: StADy GT Rule Structure

Exception for unsuccessful check; replace line 65 with the following:

```
//Initial state to backtrack to
check(name(NextState_)=="Type");
```

Exception for counsel rule; add before line 61:

```
//Ensure filled prescription has been checked
AttributeValue(AttributeValue2_);
neg find RequiresChecked(Case_,AttributeValue2_);
```

```
  gtrule Rule_<RULE>() =
{
      precondition pattern lhs(Case_,AttributeValue_,Artifact_,Artifact2_,
            StateInstance_, NextState_,R6_ ) =
    {
        //declarations
          Case(Case_);
          AttributeValue(AttributeValue_);
          State(State_);
          State(NextState_);
          StateInstance(StateInstance_);

        //check boolean value of state attribute
          find Requires<ATTRIBUTE-DECLARATION>(Case_,AttributeValue_);

        //Verify that required role(s) are assigned to the Case
          find <ROLE>assigned (Case_,RoleInstance_,Role_,Person_);

        //Verify that <ARTIFACT>  in the Case
          find <ARTIFACT>Exist(Case_,Artifact_,ArtifactType_);

        //Retrieve Case's current State
          Case.currentState(R6_,Case_,StateInstance_);

        //Find the next state from the current state
          State.next(R7,State_,NextState_);

        //Check if current state is the required state
          typeOf(State_,StateInstance_);
          check(name(State_)=="<STATE>");
    }
  action {

      let
      NewStateInstance_=undef,
      R2_=undef
      in seq{
            //Remove Case's old state
              delete(StateInstance_);
              delete(R6_);

            //Change Case state to the next state
            // create Class inside the Case
              new(StateInstance(NewStateInstance_)in DSM.model.M1);
            //create instanceOf relation with the StateInstance:State
              new(instanceOf(NewStateInstance_,NextState_));
              new (Case.currentState(R2_,Case_,NewStateInstance_));
    }

        //Change attributes boolean value
          setValue(AttributeValue_,"true");

    }
}

    //check if Case <ATTRIBUTE-DECLARATION>'s attribute value is false
      pattern Requires<ATTRIBUTE-DECLARATION>(Case_, AttributeValue_)={
        //declarations
          Case(Case_);
          AttributeValue(AttributeValue_);
          AttributeDeclaration(AttributeDeclaration_);
```

```
                    //attribute value connected to the case
                      Case.has(R1,Case_,AttributeValue_);

                    //Check that the AttributeValue is of <ATTRIBUTE-DECLARATION> type
                      typeOf(AttributeDeclaration_,AttributeValue_);
                      check (name(AttributeDeclaration_) == "<ATTRIBUTE-DECLARATION>");

                    //check if the attribute value is false
                      check(value(AttributeValue_)=="false");
                }
```

Standard Format 3

The following VTML code corresponds to new case GT rules defined in task 2.4 (Section 11.4). Table 12.3 outlines the string patterns located in the code, whereas Figure 12.4 illustrates the consistency with the StADy concrete notation.

<RULE>: GT rule name (String type).

<DIFF>: Deadline value from current time i.e. current time +<DIFF> (int type).

<PRIORITY>: Priority value (int type).

<TYPE>: Attribute declaration value type (value=walkin,refill,sameday,delivery).

<GROUP>: Distributed group number (int type).

<RULE>	**<DIFF>**	**<PRIORITY>**	**<TYPE>**	**<GROUP>**
new walkin case	15	-1	walkin	1
new refill case	1440	-1	refill	1
new sameday case	60	-1	sameday	1
new delivery case	15	-1	delivery	1

Table 12.3: Unique Procedural GT Rule String Patterns

Various other value combination can be constructed in Table 12.3 such as ranges in the priority level and different group locations.

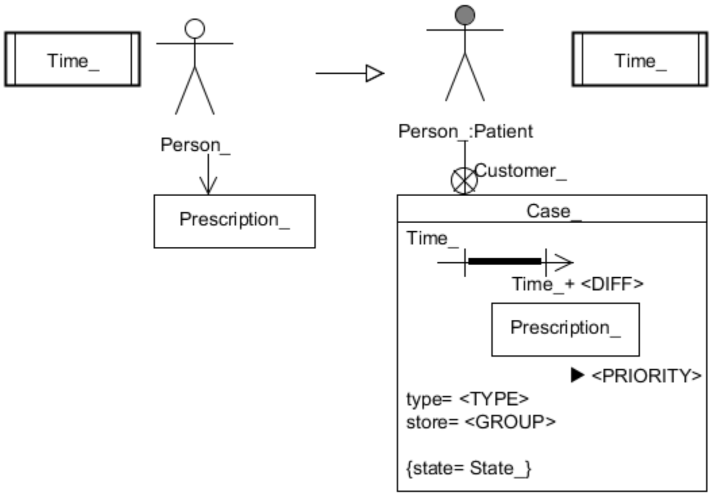


Figure 12.4: Standard Format 3: StADy GT Rule Structure

```
//New Case with Customer
          gtrule Rule_NewCase() =
        {
              precondition pattern lhs(State_,Process_, Prescription_,Patient_,
                Customer_,Time_, Day_,Month_,Year_,Hour_,Minute_,M2_) =
            {
                   State(State_);
                   Process(Process_);
                   ArtifactType(Prescription_);
                   Actor(Patient_);
                   Role(Customer_);
                   M2(M2_);
                   Clock(Clock_);

                   check(name(Prescription_)=="Prescription");
                   check(name(Patient_)=="Patient");
                   check(name(Customer_)=="Customer");
                   check(name(State_)=="Type");

                   Clock.Time(Time_);
                   Clock.attr(R1,Clock_,Time_);
                   find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
            }
               action {
                   let
                       Case_=undef, Val1_=undef,Val2_=undef, Val3_=undef,R1=undef,R2=undef,
                       R3=undef, R4=undef, R5=undef, R6=undef,  R7=undef, R8=undef,  R9=undef,
                       NewArtifact_=undef,  R1_=undef,  Person_=undef,  RoleInstance_=undef,
                       Priority_=undef, State_=undef,  Free_=undef, StartTime_=undef,
                       R2_=undef,  IntegerDay_= undef,   R3_=undef,  IntegerMonth_= undef,
                       R4_=undef,  IntegerYear_= undef,  R5_=undef,  IntegerHour_= undef,
                       IntegerMin_= undef,  R6_=undef,  Deadline_=undef,  R2R=undef,
                       R2_2=undef, IntegerDay2_= undef,  R3_2=undef,  IntegerMonth2_= undef,
                       R4_2=undef, IntegerYear2_= undef,  R5_2=undef, IntegerHour2_= undef,
                       IntegerMin2_= undef,  R6_2=undef,  Store_=undef, R10=undef, R11=undef
                   in seq {
                       new(Case(Case_)in M2_);
                       rename(Case_,name(Case_)+"Case");
                    //create instanceOf relation with the Case:Process
                       new(instanceOf(Case_,Process_));
                       new(AttributeValue(Val1_)in Case_);
                       new(AttributeValue(Val2_)in Case_);
                       new(AttributeValue(Val3_)in Case_);
                    //create instanceOf relation with the AttributeValue:Process
                       new(instanceOf(Val1_,DSM.model.M1.Pharmacy.checked));
                    //create instanceOf relation with the AttributeValue:Process
                       new(instanceOf(Val2_,DSM.model.M1.Pharmacy.counsel));
                    //create instanceOf relation with the AttributeValue:Process
                       new(instanceOf(Val3_,DSM.model.M1.Pharmacy.type));

                    //connect checked, counsel and type attribute vales to Case instance
                       new(Case.has(R1,Case_,Val1_));
                       new(Case.has(R2,Case_,Val2_));
                       new(Case.has(R3,Case_,Val3_));

                    //initialize values
                       setValue(Val1_,"false");
                       setValue(Val2_,"false");
                       setValue(Val3_,"<TYPE>");

                    //set priority level for the Case
                       new(Case.priority(Priority_) in Case_);
                       new(Case.attr2(R7,Case_,Priority_));
                       setValue(Priority_,"<PRIORITY>");

                    //Set  Case state to the initial type state
                       new(StateInstance(NewStateInstance_)in DSM.model.M1);
                    //create instanceOf relation with the StateInstance:State
                       new(instanceOf(NewStateInstance_,State_));
                       new (Case.currentState(R11_,Case_,NewStateInstance_));

                       new(Case.groupNo(Store_) in Case_);
                       new(Case.group(R10,Case_,Store_));
                       setValue(Store_,"<GROUP>");

                    //New case contains a prescription
                       new(Artifact(NewArtifact_)in Case_);
                    //create instanceOf relation with the Artifact:ArtifactType
                       new(instanceOf(NewArtifact_,Prescription_));
                       new (Case.contains(R1_,Case_,NewArtifact_));

                    //Add a person
                       new(Person(Person_)in DSM.model.M1);
                       new(instanceOf(Person_,Patient_));
                       rename(Person_,name(Person_)+"Person");
                       new (Person.free(Free_)in Person_);
                       setValue(Free_,"false");
                       new(Person.attr(R6,Person_,Free_));

                    //Add customer roleInstance
                       new(RoleInstance(RoleInstance_)in DSM.model.M1);
                       new(instanceOf(RoleInstance_,Customer_));

                    //add roleInstance to Case_
                       new(RoleInstance.presence(R4,RoleInstance_,Case_));
                    //assign Person_ to RoleInstance_
```

```
   new (RoleInstance.assignedTo(R5,RoleInstance_, Person_));


//Starttime defined
   new( Case.startTime(StartTime_) in Case_);
   new(Case.attr3(R8,Case_,StartTime_));
   new(Integer(IntegerDay_) in StartTime_);
   setValue(IntegerDay_,value(Day_) );
   new(DateTime.day(R2_,StartTime_,IntegerDay_));
   new(Integer(IntegerMonth_) in StartTime_);
   setValue(IntegerMonth_,value(Month_));
   new(DateTime.month(R3_,StartTime_,IntegerMonth_));
   new(Integer(IntegerYear_) in StartTime_);
   setValue(IntegerYear_,value(Year_));
   new(DateTime.year(R4_,StartTime_,IntegerYear_));
   new(Integer(IntegerHour_) in StartTime_);
   setValue(IntegerHour_,value(Hour_));
   new(DateTime.hour(R5_,StartTime_,IntegerHour_));
   new(Integer(IntegerMin_)in StartTime_);
   setValue(IntegerMin_,value(Minute_));
   new(DateTime.minute(R6_,StartTime_,IntegerMin_));

   if(toInteger(value(IntegerMin_)) >=10 ) seq{
       setValue (StartTime_, value(IntegerHour_)+":"+
            value(IntegerMin_)+" "+ value(IntegerDay_)+"/"+
            value(IntegerMonth_)+"/"+ value(IntegerYear_));
}
   else seq{
       setValue (StartTime_, value(IntegerHour_)+":0"+
            value(IntegerMin_)+" "+value(IntegerDay_)+"/"+
            value(IntegerMonth_)+"/"+ value(IntegerYear_));
}
//deadline defined
   new(Case.deadline(Deadline_) in Case_);
   new (Case.attr1(R2R, Case_,Deadline_));
   new(Integer(IntegerDay2_) in Deadline_);
   setValue(IntegerDay2_,toInteger(value(IntegerDay_)));
   new(DateTime.day(R2_2,Deadline_,IntegerDay2_));
   new(Integer(IntegerMonth2_) in Deadline_);
   setValue(IntegerMonth2_,toInteger(value(IntegerMonth_)));
   new(DateTime.month(R3_2,Deadline_,IntegerMonth2_));
   new(Integer(IntegerYear2_) in Deadline_);
   setValue(IntegerYear2_,toInteger(value(IntegerYear_)));
   new(DateTime.year(R4_2,Deadline_,IntegerYear2_));
   new(Integer(IntegerHour2_)in Deadline_);
   setValue(IntegerHour2_,toInteger(value(IntegerHour_)));
   new(Integer(IntegerMin2_) in Deadline_);

   if((toInteger(value(IntegerMin_))+<DIFF>) < 60) seq{
       setValue(IntegerMin2_,toInteger(value(IntegerMin_))+<DIFF>);
       setValue(IntegerHour2_,toInteger(value(IntegerHour_)));
}
   else if (toInteger(value(IntegerHour_))+1 < 24 ||
        ((toInteger(value(IntegerHour_))+1==24 &&
        ((toInteger(value(IntegerMin_))+<DIFF>)-60)==0 ))) seq{
       setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+<DIFF>)-60));
       setValue(IntegerHour2_,toInteger(value(IntegerHour_))+1);
}//assumption that all months have 31 days
   else if((toInteger(value(IntegerDay_))+1)<31) seq {
       setValue(IntegerDay2_,toInteger(value(IntegerDay_))+1);
       setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+<DIFF>)-60));
       setValue(IntegerHour2_,1);
```

```
            }
               else if((toInteger(value(IntegerMonth_))+1)<13) seq{
                   setValue(IntegerMonth2_,toInteger(value(IntegerMonth_))+1);
                   setValue(IntegerDay2_,1);
                   setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+<DIFF>)-60));
                   setValue(IntegerHour2_,1);
            }
               else seq{
                   setValue(IntegerYear2_,toInteger(value(IntegerYear_))+1);
                   setValue(IntegerMonth2_,1);
                   setValue(IntegerDay2_,1);
                   setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+<DIFF>)-60));
                   setValue(IntegerHour2_,1);
            }

               new(DateTime.hour(R5_2,Deadline_,IntegerHour2_));
               new(DateTime.minute(R6_2,Deadline_,IntegerMin2_));

               if(toInteger(value(IntegerMin2_)) >=10 ) seq{
                   setValue (Deadline_,   value(IntegerHour2_)+":"
                      +value(IntegerMin2_)+" " +value(IntegerDay2_)+"/"+
                      value(IntegerMonth2_)+"/"+ value(IntegerYear2_));
            }
               else seq{
                   setValue (Deadline_,   value(IntegerHour2_)+":0"+
                        value(IntegerMin2_)+" "+value(IntegerDay2_)+"/"+
                        value(IntegerMonth2_)+"/"+ value(IntegerYear2_));
            }
                   setAggregation(R2_,true);
                   setAggregation(R3_,true);
                   setAggregation(R4_,true);
                   setAggregation(R5_,true);
                   setAggregation(R6_,true);
                   setAggregation(R2_2,true);
                   setAggregation(R3_2,true);
                   setAggregation(R4_2,true);
                   setAggregation(R5_2,true);
                   setAggregation(R6_2,true);

                   println("Case added ");
        }
    }
}


//Returns the current clock time
   pattern getTime(Time_, Day_,Month_,Year_,Hour_,Minute_)={
       Clock.Time(Time_);
       Integer(Day_);
       Integer(Month_);
       Integer(Year_);
       Integer(Hour_);
       Integer(Minute_);

    //Retrieve current time attribute values
       DateTime.day(R2_,Time_,Day_);
       DateTime.month(R3_,Time_,Month_);
       DateTime.year(R4_,Time_,Year_);
       DateTime.hour(R5_,Time_,Hour_);
       DateTime.minute(R6_,Time_,Minute_);
}
```

Consistency holds true between the StADy syntax GT rules defined in task 2.3-2.4 (Sections 11.3-11.4) and the GT rules defined in VTCL syntax, because they passed the following consistency test: *The GT rule textual and a graphical representation are equivalent if they generate the same abstract representations of a particular model or there exist a mapping from the graphical representation to textual representation based on the discussion in Section 8.3:* This holds true because the mapping from the textual to graphical represents is shown in Figures 12.2-12.4 for the 3 standard code formats.

12.4 Probes StADy in VIATRA2 Syntax (Task 3.4)

This section illustrates the application of task 3.4 on the pharmacy case study (Chapter 5), by translating the probes defined in task 2.5 (Section 11.5) into VIATRA2 textual syntax. The probes for late, case and idle are as follows:

Probe Late

```
gtrule ProbeLate()={
      precondition pattern lhs (Case_,Clock_,Time_,Deadline_,Day_,Month_,Year_,
            Hour_,Minute_,Day2_,Month2_,Year2_,Hour2_,Minute2_)={
                Case(Case_);
                Clock(Clock_);
             Clock.Time(Time_);
             Case.deadline(Deadline_);
             Clock.attr(R1,Clock_,Time_);
             find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
             Case.attr1(R2, Case_,Deadline_);
             find getEndTime(Deadline_, Day2_,Month2_,Year2_,Hour2_,Minute2_);

             check ((60*toInteger(value(Hour_)) + toInteger(value(Minute_)))
                < (60*toInteger(value(Hour2_)) +toInteger(value(Minute2_))));
      }
```

```
     action {
          println("Case late");
      }
   }
```

Probe Cases

```
gtrule ProbeCases()={
    precondition pattern lhs (Case_)= {
     Case(Case_);
    }
     action {
          println("Cases");
      }
   }
```

Probe Idle

```
gtrule ProbeRule_PersonAvailability()={
  precondition pattern lhs(Person_,Free_) =
           {
            Person(Person_);
            Person.free(Free_);
            Person.attr(P1,Person_,Free_);
            check (value(Free_)=="true");
           }
            action {
                print ("Person is free");
            }
 }
```

Consistency holds true between the StADy syntax probe defined in task 2.5 (Section 11.5) and the probes defined in VTCL syntax, because they passed the following consistency test: *The probes in textual and a graphical representation are equivalent if they generate the same abstract representations of a particular model or there exist a mapping from the graphical representation to textual representation based on the discussion in Section 8.3:* This holds true because the abstract representation of the StADy probes correspond to the VTML textual representation probes. As illustrated in

Figure 11.18 for the *PersonAvailable* probe it contains elements: Actor and Person. The Boolean *free* attribute in the *Person* object is set to true, which is equivalent to $Person.attr(P1, Person\_, Free\_)$; $check(value(Free\_) == "true")$; VTML code.

12.5 Simulation Parameters (Task 3.5)

This section illustrates task 3.5. This task is for defining the simulation parameters into the VIATRA2 model space (VPML file) and XML files.

The parameter.xml is similar to the sample parameter file defined in the Methodology in Section 9.7.5, except for the CDF_Input, Output_Folder, Depth_Limit, Time_Opt and Batch_Size attribute values. The CDF_Input value is for specifying the four unique stochastic input files which correspond to versions 1-4 simulation models defined in task 2.1 (Section 11.1). Similarly the Output_Folder corresponds to four unique folder names for containing the results of four different simulation runs. All the simulation runs: Depth_Limit are set to 2500, Time_Opt are set to false and Batch_Size are set to 3.

The distribution values defined in task 2.7 (Section 11.7) are entered into the stochastic input xml files. The input values refer to cumulative distribution functions (CDF) of the required GT rules encoded in tasks 3.2–3.3 (Sections 12.2–12.3). Below is the stochastic input file for version #3 (Scheduling by deadline and assignment policy) of the simulation models designed in task 2.1 (Section 11.1). Each of the rules are defined as observable actions rules, in order to gather statistical data of their applications in the simulation run.

```
<?xml version="1.0" encoding="UTF-8"?>
<allrules>
<ruleset name="random">

    <rule name="Rule_NewCase" type="AO">
        <event name="0" type="exp">
            <rate value=".1"/>
        </event>
    </rule>

    <rule name="Rule_NewCaseMedPriority" type="AO">
        <event name="0" type="exp">
            <rate value="0.05"/>
        </event>
    </rule>

    <rule name="Rule_NewCaseHighPriority" type="AO">
        <event name="0" type="exp">
            <rate value="0.01667"/>
        </event>
    </rule>

    <rule name="Rule_TypePrescription" type="AO">
        <event name="0" type="norm">
            <mean value="1.25" />
            <variance value="0.4375" />
        </event>
    </rule>



    <rule name="SkipRule_TypePrescription" type="AO">
        <event name="0" type="norm">
            <mean value="1.50" />
            <variance value="0.4375" />
        </event>
    </rule>

    <rule name="Rule_Counsel" type="AO">
        <event name="0" type="norm">
            <mean value="1.17" />
            <variance value="0.583333" />
        </event>
    </rule>

    <rule name="SkipRule_Counsel" type="AO">
        <event name="0" type="norm">
            <mean value="1.42" />
            <variance value="0.583333" />
        </event>
    </rule>

    <rule name="Rule_CounselLate" type="AO">
        <event name="0" type="norm">
            <mean value="1.17" />
            <variance value="0.583333" />
        </event>
    </rule>
    <rule name="SkipRule_CounselLate" type="AO">
        <event name="0" type="norm">
            <mean value="1.42" />
            <variance value="0.583333" />
        </event>
    </rule>

    <rule name="Rule_Unassign" type="AO">
        <event name="0" type="norm">
            <mean value="0.3125" />
            <variance value="0.048032407" />
        </event>
    </rule>



    <rule name="Rule_AssignPolicy" type="AO">
        <event name="0" type="norm">
            <mean value="0.09" />
            <variance value="0.004537" />
        </event>
    </rule>

    <rule name="Rule_RequestDeadline" type="AO">
        <event name="0" type="norm">
            <mean value="0.09" />
            <variance value="0.004537" />
        </event>
    </rule>

    <rule name="Rule_ClockTick" type="AO">
        <event name="0" type="norm">
            <mean value="1.00" />
            <variance value="0.000055778" />
        </event>
    </rule>

    <rule name="BacktrackRule_PrintState" type="AO">
        <event name="0" type="norm">
            <mean value="0.26" />
            <variance value="0.024806" />
        </event>
    </rule>

    <rule name="ProbeRule_PersonAvailability" type="P">   </rule>

    <rule name="ProbeLate" type="P"></rule>

    <rule name="ProbeCases" type="P"> </rule>

    <probeset>
        <probe name="NumCases" op="ProbeCases"> </probe>
        <probe name="PersonAvailable" op="ProbeRule_PersonAvailability"> </probe>
        <probe name="LateCases" op="ProbeLate"> </probe>
        <probe name="LateFromTotal" op="div">
            <arg name="ProbeCases" pos="2"></arg>
            <arg name="ProbeLate" pos="1"></arg>
        </probe>
    </probeset>
</ruleset>
</allrules>
```

Consistency holds true between the GT rule/probe distributions and encodings of their distributions, because they passed the following consistency test:

1. All GT Rule distribution parameters that were selected for a particular simulation model version rules must be encoded into the stochastic input file.
   - The illustrated stochastic input file for version #3 simulation model contains the required scheduling by deadline and assignment policy GT rules which corresponds to *Rule_RequestDeadline* and *Rule_AssignPolicy* action rules.
2. Each probe formulated in task 3.4 should be added to the probe definition list in the stochastic input file.
   - *ProbeLate*, *ProbeRule_PersonAvailability*, *ProbeCases* are defined in *LateCases*, *PersonAvailable* and *NumCases*, respectively. Also probe *LateFromTotal* is defined using a combination of two of the previously defined probes with a division operation.

12.6 Summary

This chapter illustrated the business process encoding tasks in stage 3 of the StADy methodology (Chapter 9). The start graph, GT rules and probes that were defined in stage 2 were translated in VIATRA2 syntax in order to perform stochastic simulation in stage 4 (Chapter 13).

Chapter 13

Illustration of Stage 4 – Performance Evaluation

This chapter describes and illustrates the tasks in stage 4 of the methodology (Chapter 9). Each task includes consistency checks and a corresponding illustration based on the pharmacy case study defined in Chapter 5. The performance-evaluation stage is for running simulations, analyzing results and producing conclusions.

13.1 Stochastic Simulation (Task 4.1)

This section illustrates the application of task 4.1 on the pharmacy case study (Chapter 5), by describing the simulation experiment. The average simulation runtime was 30 minutes representing 3 hours of simulated time based on the average number of times the *clock tick* rule was applied (198.4125 minutes). The clock tick rule defined in Figure 7.13 was equipped with a normal distribution in Section 11.7 to reflect average one minute increment for each clock tick. The number of simulation steps was limited to 2500, with a batch size of 3, and was run on four distinct versions as described in Section 11.1. These four distinct versions were used to determine which rules and probes should be applied in each simulation run as discussed in Sections 11.3-11.5. Two of the cases were without specific assignment policies, with scheduling influenced by deadlines or priorities. The other two included the effect of the policy in addition to the choice of scheduling strategy (Section 7.2.2). The initial model for each simulation run contained one registered pharmacist, three technicians, one cashier and one active dispense medication case.

13.2 Results and Conclusions (Task 4.2)

This section illustrates the application of task 4.2 on the pharmacy case study (Chapter 5), by analysing simulation results to concluding answers to performance questions 1 and 2 (Section 1.2) based of 4 different simulation model versions specified in Section 9.6.1.

The bar graphs below visually represent the comparison of results obtained for the four versions. The probability for a prescription to be completed late is presented in Table 13.1.

Version	***P*(Case is late)**
V1 (No Policy, Deadline)	0.247
V2 (No Policy, Priority)	0.256
V3 (Policy, Deadline)	0.245
V4 (Policy, Priority)	0.244

Table 13.1: Probability That a Prescription is Completed Late

The average number of idle workers is presented in Figure 13.1. The results indicate that assignment policies and scheduling choices have an effect on idle workers while the influence on the probability of completing on time is limited. Therefore an additional measure was used to determine the number of completed cases at the end of the 3 hour period. The results are shown in Figure 13.2 and confirm the expected tendency, i.e., that version 4 (assignment policy and scheduling by priority) resulted in the most favourable results.

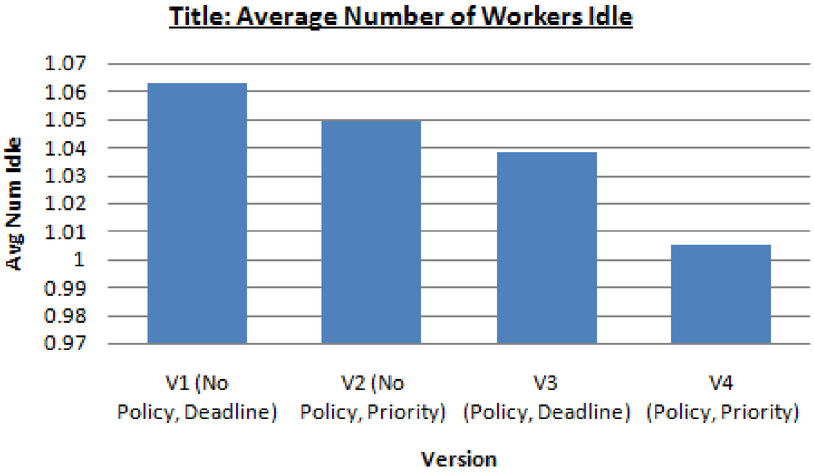


Figure 13.1: Number of Workers Idle out of 5 in Dispensary

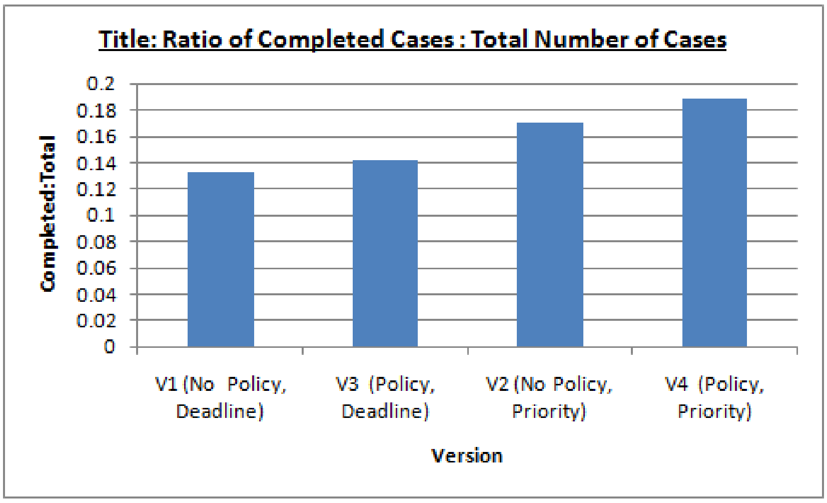


Figure 13.2: Number of Cases Completed out of the Total Received

13.3 Summary

This chapter illustrated the business performance evaluation tasks in stage 4 of the StADy methodology (Chapter 9) and illustrated them. This stage is for performing the simulation runs and analysing results. This chapter answered performance questions 1 and 2 (Section 1.2) for the pharmacy case study (Chapter 5). The next chapter will answer performance question 3 by comparing load balancing and escalation handling.

Chapter 14

Analysis of Load Balancing and Escalation Handling

This chapter illustrates application to the methodology (Chapter 9) for the type 2 simulation described in Section 9.6.1, by specifically tailoring the models to load balancing and escalation handling aspects. The type 2 simulation is required for answering question 3 of the performance questions (Section 1.2). The chapter illustrates elements that are unique to the methodology application of the type 2 simulation.

The application to the following tasks are equivalent to the illustrations of the type 1 simulation defined in Chapters 10–13:

- task 1.1 (Section 10.1)
- task 1.2 (Section 10.2)
- task 2.1 (Section 11.1)
- task 2.2 (Section 11.2)

- task 2.7 (Section 11.7)
- task 3.5 (Section 12.5)

14.1 Process Execution Design

The section presents the application on tasks 2.3–2.6 for version 5-8 simulation models defined in Table 11.2.

14.1.1 Domain-Independent GT rules in StADy DSL

All of the 4 versions used require the following: assignment policy (Figure 7.10), scheduling by deadline (Figure 7.11), assignment (Figure 7.7), unassignment (Figure 7.8), request (Figure 7.6) and clock (Figure 7.13) rules.

14.1.2 Domain-Specific GT Rules in StADy DSL

In addition to the domain specific rules defined in Section 11.4, simulation models versions 5-8 (Section 11.1) also require the following support-level GT rules based on Table 11.2.

Version 5: Load balancing (Section 7.3.3), escalation handling (Section 7.3.1)

Version 6: Escalation handling (Section 7.3.1)

Version 7: Load balancing (Section 7.3.3)

Version 8: N/A

14.1.3 Observation Rules in StADy DSL

This section specifies the observation rules that are used in answering performance question 3.

Question #3: Does escalation and/or load balancing increase the percentage of cases that are completed within a given deadline, or reduce the time that cases run past their deadline?

Question 3 can be answered by analyzing the rule batch reports for versions 5-8, after normalizing them to the same time factor based on number of executions of ‘clockTick’ rule.

The GT rules of interest are ‘SucessfulCheck’ and ‘Counsel’; however since neither of them captures the deadline completion time ratio, additional GT rules need to be defined with these additional conditions. In order to define additional pre-conditions with no side effects to the simulation results the distribution for the event should be set to the distribution values for the original ‘SucessfulCheck’ and ‘Counsel’ GT rules. The additional precondition in these rules should determine whether the prescription was on time, less than 5 minutes late, or more than 5 minutes late, for instance Figure 14.1 defines the *SucessfulCheckOntime* GT rule.

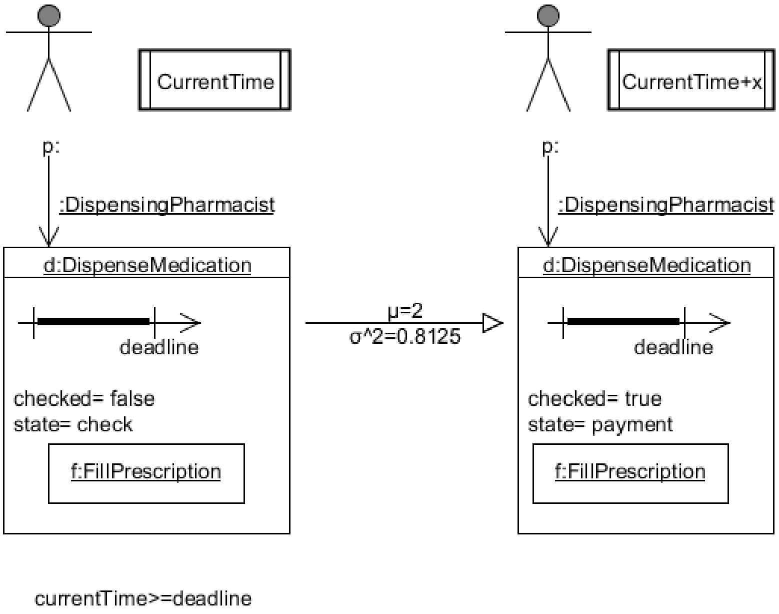


Figure 14.1: SucessfulCheck on Time

14.1.4 Start Graph in StADy DSL

This section describes start graph used in the simulation models for type 2 simulation. The initial model has a set of two pharmacies. The first pharmacy consists of one registered pharmacist (Cindy), two technicians (Bob, Donald), one cashier (Gina), one pharmacy student (Molly) and two active dispense medication cases. The second pharmacy consists of two registered pharmacists (John, Sally) and one technician (Fred). Two patients (Jill, Emily) are waiting for their prescription to be filled at store number 1. The date is Sunday July 25th, 2010 and the current time 9:01 am. Emily's prescription case has been escalated to level 1 because the expected completion time is 4 minutes from the current time. Jill's prescription arrived a minute ago. This start graph is visually represented in Figure 14.2.

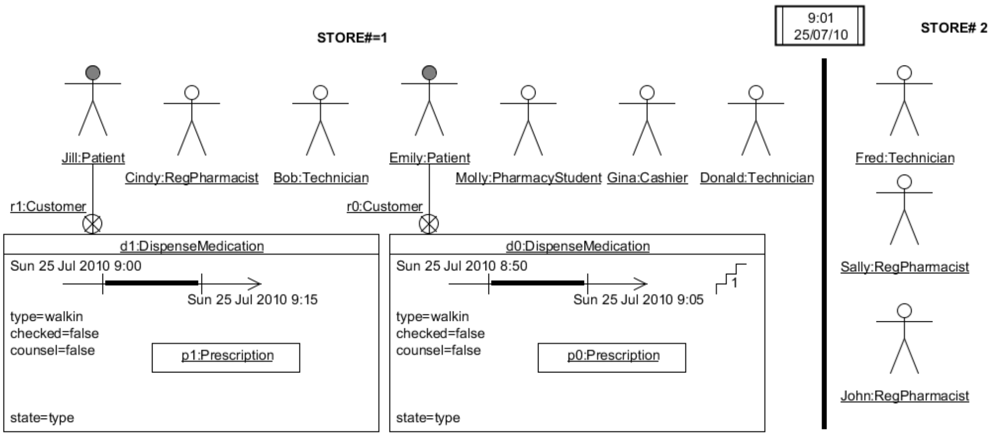


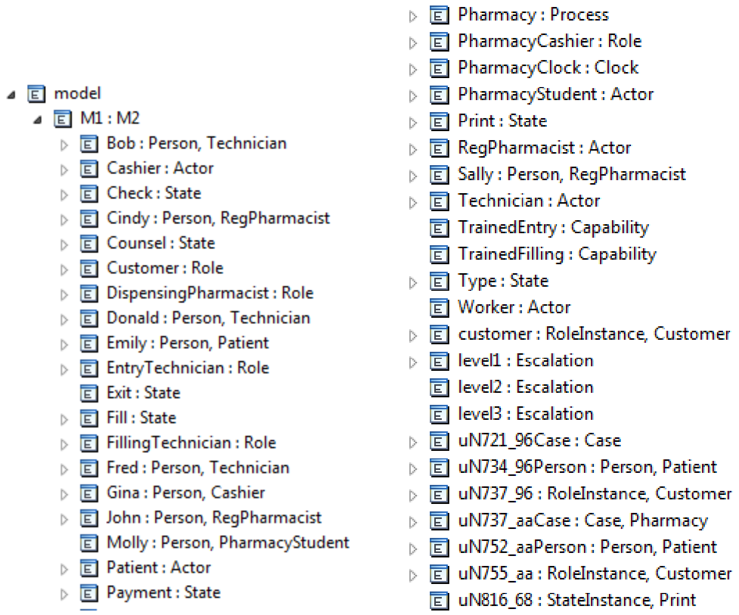
Figure 14.2: Start Graph for Simulation Type 2 Model

14.2 Process Encoding

The section presents the application on tasks 3.1–3.5 for version 5-8 simulation models defined in Table 11.2.

14.2.1 Start Graph in VIATRA2 Model Space

The start graph that was designed using StADy syntax in Section 14.1.4 is implemented in the VIATRA2 (VPM) Model Space [72] as shown in Figure 14.3.



```
model
  M1 : M2
    Bob : Person, Technician
    Cashier : Actor
    Check : State
    Cindy : Person, RegPharmacist
    Counsel : State
    Customer : Role
    DispensingPharmacist : Role
    Donald : Person, Technician
    Emily : Person, Patient
    EntryTechnician : Role
    Exit : State
    Fill : State
    FillingTechnician : Role
    Fred : Person, Technician
    Gina : Person, Cashier
    John : Person, RegPharmacist
    Molly : Person, PharmacyStudent
    Patient : Actor
    Payment : State
```

```
Pharmacy : Process
PharmacyCashier : Role
PharmacyClock : Clock
PharmacyStudent : Actor
Print : State
RegPharmacist : Actor
Sally : Person, RegPharmacist
Technician : Actor
TrainedEntry : Capability
TrainedFilling : Capability
Type : State
Worker : Actor
customer : RoleInstance, Customer
level1 : Escalation
level2 : Escalation
level3 : Escalation
uN721_96Case : Case
uN734_96Person : Person, Patient
uN737_96 : RoleInstance, Customer
uN737_aaCase : Case, Pharmacy
uN752_aaPerson : Person, Patient
uN755_aa : RoleInstance, Customer
uN816_68 : StateInstance, Print
```

(a) Part 1

(b) Part 2

Figure 14.3: Start Graph for Type 1 Simulation Model

14.2.2 Observation Rules in VIATRA2 Syntax

This section defines the observation rules that were discussed in Section 14.1.3. The new GT rules are modification to the *successfulcheck* and *counsel* GT rule that were defined in Section 12.3.

The Rule_SucessfullCheckOntime is the sucessfullcheck GT rule with the following lines of code added to the precondition:

```
find getEndTime(Deadline_, Day_,Month_,Year_,Hour_,Minute_);
find getTime(Time_, Day1_,Month1_,Year1_,Hour1_,Minute1_);

check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_)))
 <=((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));
```

The Rule_SucessfullCheckLessthan5 is the sucessfullcheck GT rule with the following lines of code added to the precondition:

```
find getEndTime(Deadline_, Day_,Month_,Year_,Hour_,Minute_);
find getTime(Time_, Day1_,Month1_,Year1_,Hour1_,Minute1_);

check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_)))
    >((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));
check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_))-5)
    <=((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));
```

The Rule_SucessfullCheckMorethan5: is the sucessfullcheck GT rule with the following lines of code added to the precondition:

```
find getEndTime(Deadline_, Day_,Month_,Year_,Hour_,Minute_);
find getTime(Time_, Day1_,Month1_,Year1_,Hour1_,Minute1_);

check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_)))
    >((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));
check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_))-5)
    >((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));
```

The Rule_CounselLate: is the counsel GT rule with the following lines of code added to the precondition:

```
Clock(Clock_);
Clock.Time(Time_);
Case.deadline(Deadline_);
Clock.attr(R1,Clock_,Time_);
find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
Case.attr1(R2, Case_,Deadline_);
find getEndTime(Deadline_, Day2_,Month2_,Year2_,Hour2_,Minute2_);

check ((60*toInteger(value(Hour_)) + toInteger(value(Minute_)))
    < (60*toInteger(value(Hour2_)) +toInteger(value(Minute2_))));
```

The Rule_CounselOntime: is the counsel GT rule with the following lines of code added to the precondition:

```
Clock(Clock_);
Clock.Time(Time_);
Case.deadline(Deadline_);
Clock.attr(R1,Clock_,Time_);
find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
Case.attr1(R2, Case_,Deadline_);
find getEndTime(Deadline_, Day2_,Month2_,Year2_,Hour2_,Minute2_);

check ((60*toInteger(value(Hour_)) + toInteger(value(Minute_)))
    >= ((60*toInteger(value(Hour2_))) +toInteger(value(Minute2_))));
```

14.2.3 Simulation Parameters

In addition to the simulation parameters defined in Section 12.5 the stochastic input files will require the following parameters to encode escalation handling, load balancing, and observation rules (Section 14.1.3).

Escalation

```
    <rule name="TempRule_AssignEntryTechnician1" type="AO">
        <event name="0" type="norm">
            <mean value="0.05" />
            <variance value="0.004537" />
        </event>
    </rule>

    <rule name="Rule_trigger1" type="AO">
        <event name="0" type="norm">
            <mean value="0.09" />
            <variance value="0.004537" />
        </event>
    </rule>
```

Load Balancing

```
 <rule name="DisRule_TypePrescription" type="AO">
        <event name="0" type="norm">
            <mean value="1.25" />
            <variance value="0.4375" />
        </event>
    </rule>
```

Observation Rules

```
<rule name="Rule_SucessfullCheckOntime" type="A">
    <event name="0" type="norm">
        <mean value="1.5" />
        <variance value="0.8125" />
    </event>
</rule>

<rule name="Rule_SucessfullCheckLessthan5" type="A">
    <event name="0" type="norm">
        <mean value="1.5" />
        <variance value="0.8125" />
    </event>
</rule>
```

```
<rule name="Rule_SucessfullCheckMorethan5" type="A">
    <event name="0" type="norm">
        <mean value="1.5" />
        <variance value="0.8125" />
    </event>
</rule>

<rule name="Rule_CounselLate" type="AO">
    <event name="0" type="norm">
        <mean value="1.17" />
        <variance value="0.583333" />
    </event>
</rule>

<rule name="Rule_CounselOntime" type="AO">
    <event name="0" type="norm">
        <mean value="1.17" />
        <variance value="0.583333" />
    </event>
</rule>
```

14.3 Performance Evaluation

This section discusses the type 2 simulation experiment, analyses the results and draws conclusions.

14.3.1 Stochastic Simulations

The average times for activities in the pharmacy process were defined according to [61, 69]. Rules for creating new cases have been given exponential distribution values while the remaining rules were defined using normal distribution values. The results represent 2.77 hours of simulated time based on the number of times the *clock tick* rule was applied (166 mins). The number of simulation steps was limited to 2500, with a batch size of 3, and was run on four distinct versions as described in Chapter 9.

Two of the versions implemented escalation handling with three predefined levels. The escalation would be triggered if the case was within 5 minutes to the deadline, at the deadline or more than 5 minutes past the deadline. At escalation level 1, pharmacy cashiers are temporarily permitted to be assigned as entry technicians for that particular case. At escalation level 2, pharmacy cashiers are temporarily permitted to be assigned as filling technicians. At escalation level 3 untrained pharmacy students are temporarily permitted to be assigned as filling technicians and/or entry technicians. The other two versions implement load balancing across two pharmacies, providing the option to transfer prescriptions to another store.

All of the four versions were tested on the same model (instance graph). The initial model was composed of two pharmacies. The first pharmacy consists of one registered pharmacist, two technicians, one cashier, one pharmacy student and two active dispense medication cases. The second pharmacy consists of two registered pharmacists and one technician.

14.3.2 Results and Conclusion

The bar graphs below visually represent the comparison of results obtained for the four versions. Question #3 raised in Section 1.2 is answered as follows. The percentage of cases that completed particular states in a process within 2.77 hours is presented in Figure 14.4. The percentage of cases that completed the check and counsel states on time, versus the range of lateness for the completeness of check and counsel states, is presented in Figure 14.5. The results indicate that the addition of escalation and load balancing has a positive influence on the probability of completing on time. This confirms the expected tendency, i.e., that version 5 (escalation and load balancing) results in the most favourable performance.

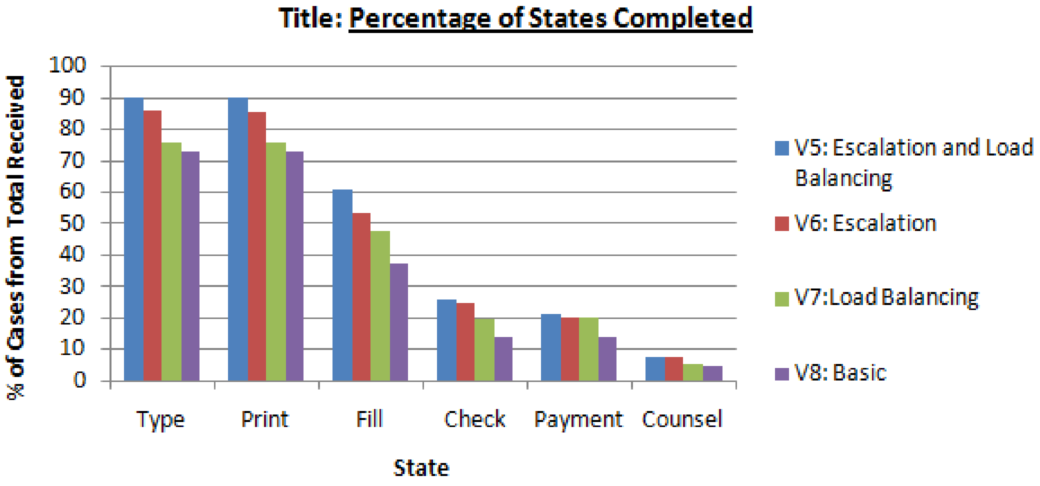


Figure 14.4: State Completion Comparison

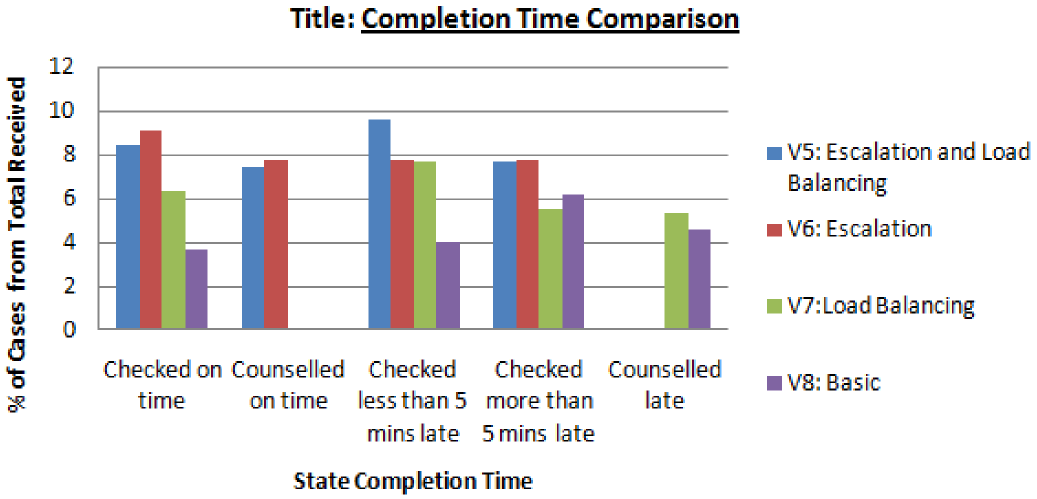


Figure 14.5: Completion Time Comparison

The results illustrated that a combination of escalation handling and load balancing could improve the business process. These quantitative findings suggest various improvements that can take place in a busy pharmacy, such as increasing staff, transferring prescriptions to the owner's other nearby pharmacy, and temporarily permitting people to take on higher roles as far as permitted by law.

14.4 Summary

This chapter answered performance question 3 (Section 1.2) for the pharmacy case study (Chapter 5), by specifically tailoring the simulation models to load balancing and escalation handling aspects. The next chapter will provide an evaluation (Chapter 15) of the StADy approach.

Chapter 15

Evaluation

In this chapter, the StADy approach is evaluated based on its requirement coverage. Since the evaluation of the StADy methodology is limited to the case study performed (Chapters 10-14), Sections 15.1-15.4 discuss how the requirements from Chapter 2 were implemented and how the case study provides evidence that they work. Sections 15.1 and 15.2 evaluate the StADy language coverage of the role management and task management concept requirements, respectively. The language requirements are evaluated against the StADy language syntax in Section 15.3, whereas the method requirements are evaluated against the StADy methodology in Section 15.4. Section 15.5 illustrates the requirement coverage using an application scenario. The chapter is concluded with an evaluation on the possible threats to validity on the simulation results.

15.1 Role Management

The following sections illustrate the occurrence of the role management concept requirements in the StADy modelling approach.

15.1.1 Dynamic (Re)-Assignment

The dynamic assignment/re-assignment concept requires the ability to dynamically re-assign tasks to people based on their availability and capabilities. A person's availability is defined in the *Person.free* StADy metamodel attribute, whereas a person's capability corresponds to his or her job position (*Actor*) or acquired capabilities. The dynamic aspect occurs with GT pattern matching during simulation runs. (Re)-assignment is based on the following combination of GT rules: request rule, assignment rule, and unassignment rule. If the start graph is empty then the number of applications of these rules to be relative to each other such that be $requestRule \geq assignmentRule \geq unassignRule$. This holds true for the simulation run of simulation model version #5 (Section 11.1), because the average application of these three rules from 3 batches was as follows: request rule=306.334; assignment rule= 304.001 and unassignment rule=299.333. Each of these rules use *Person–RoleInstance–Case* relation for representation of assignment, such that *RoleInstance–Case* with the absence of a Person connection denotes request for assignments. These GT rules are further discussed and illustrated in Section 7.2.1. Assignment policies conditions can be specified by using a combination of the *Actor–Actor super* relation and *Person.free* attribute. If the pharmacy owner specified that if a filling technician role is required to be assigned and both a registered pharmacist and a pharmacy technician are available, then the technician should take on that role. This example corresponds to the existing predefined managerial assignment policy global application condition in Figure 7.10, hence it can be used in the process execution design stage.

15.1.2 Role Promotion, Demotion and Temporary Promotion

The role promotion/demotion and temporary promotion concept requires a means to denote acquired capabilities and temporary permission to roles. A person's capabilities are associated with the *Role–Capability–Person* relation in the StADy metamodel, such that if a person has the required capability to perform a role then he or she is considered promoted. For example, if a pharmacist student gains training to be a filling technician then he or she can be promoted to the role of filling technician. The required capability is the training to be acquired by the person. Role demotion is represented when the Capability–Person relation is removed. Temporary promotion occurs in response to an escalation, therefore the *Actor–Escalation–Role* and *Escalation–Case* relations are used to specify which job position (Actor) is temporarily permitted to take on a role for one or more escalated cases. The number of times an escalation triggers is relative to the number of times the escalation routine can be applied; hence application of: $(escalation\_trigger1) \geq (escalation\_routine1)$, where the routine is temporary promotion in this case. For instance, in simulation model version #5's

3 batches of simulation runs, escalation number 1 was triggered on average 95.00 which resulted in temporarily assigning a pharmacy cashier to the entry technician role 26.667. Evaluation of the of escalation trigger rules is further discussed in Section 15.2.3 and the temp assign GT rule is further illustrated and discussed in Section 7.3.2.

15.1.3 Role-Based Access Control

The access control feature requires a means to distinguish a persons' permitted roles and access rights to artifacts. A person's permitted role is defined using the StADy metamodel *Actor<- -Person* ontological *instanceOf* relation and the *Actor–Role* relation. Also role permission can be inherited from other Actors using the *super* relation. The access rights to particular artifacts within cases are specified using the *RoleInstance–Artifact* association and *Artifact–Case* containment relationship. At the model level the access rights are directly encoded into the StADy hierarchy and artifact models, as illustrated and discussed in Section 6.3. RBAC is demonstrated in the simulation results for performance question #2 (Section 13.2) by presenting the impact of using assignment policies.

15.2 Task Management

The following sections illustrate the occurrence of the task management concept requirements in StADy modelling approach.

15.2.1 Scheduling

The scheduling feature requires scheduling protocols which take into account task deadlines and/or priorities, with the prime goal to maximise throughput and minimise the number of tasks completed past the deadline. The StADy metamodel attributes that are used in the predefined managerial scheduling GT rules in Section 7.2.2 are as follows: *Case.priority*, *Case.deadline* and *Clock.time*. Hence, if the process is scheduled by priority, then an urgent case is set to a high priority level in order to take preference over other cases. Scheduling by deadline or priority were demonstrated in the simulation results for performance question #1 (Section 13.2), which compared their impact to the business process.

15.2.2 Non-Deterministic Duration of Tasks

The non-deterministic duration of task feature is a means for defining the estimated time it takes for a person or technical resource to complete a task. The estimated times it takes to complete a task is encoded into the stochastic graph transformation system as discussed and illustrated in Sections 4.3, 7.1 and 11.7, by assigning each GT rule with normal or exponential distributions. The non-determinism aspect of the feature is reflected in the use of GT rules. Therefore, the GT rule will have a corresponding mean and variance values, which are visually denoted on the GT transition arrow. For instance, the clock tick rule is a normal distribution which occurs about every minute. Since the StADy model only uses one local clock there is no problem with synchronization of multiple local clocks as discussed in [30]. The clock tick rule increments the single clock node which behaves as a global notion of time because it is referenced by all cases in the system. A normal distribution was selected to reflect average one minute increment for each clock tick. The overall simulation performance does not suffer because the clock ticks by the minute and there is only one clock, resulting in the other business process actions not completely being overtook by the clock tick executions.

15.2.3 Temporal Escalation Handling

The temporal escalation handling feature requires ability to escalate based on time. There should also be a means for specifying the handling routine, in particular it may permit people to overstep their permissions by temporarily promoting them to a required role on an escalated case. Procedures should be in place specifying how to handle escalations so that they comply with legal requirements and are allowed to react efficiently. In StADy a *Case*'s escalation level is denoted by using *Escalation–Case* relation and the *Actor* (job position) temporary permitted *Role* is denoted using the *Actor–Escalation–Role* relation. An escalation is influenced by the current *Clock.time* and *Case.deadline* metamodel attributes. An escalation needs to be specified by the business user and encoded by the developer. Escalation handling is demonstrated in the simulation results for performance question #3 (Section 14.3.2), which showed its impact towards a business process. If the escalation triggers are related by degree then the number of rule application should have a relative relation such that $Trigger1 \geq Trigger2 \geq Trigger3$. For instance, in the 3 batch simulation runs of model #5 the average application of escalation rules was as follows: trigger 1=95.00; trigger 2= 93.667 and trigger 3= 89.00. These values correspond to the expected relation. Domain specific trigger GT rules is further illustrated in Section 7.3.1 which defines triggers influenced by a prescription case deadline.

15.2.4 Load Balancing

The load balancing feature requires the means to dynamically assign/reassign tasks to other locations. In order to capture this aspect the StADy metamodel used *Case.groupNo* and *Person.groupNo* attributes to specify the location of the cases and people involved, hence this implicitly adds another condition to the assignment GT rules. A load balancing assignment needs to be specified by the business user and encoded by the developer. For instance, Section 7.3.3 illustrates a domain specific transfer (global assignment) GT rule, which globally reassigns prescriptions during ‘type’ state if the requested prescription has been ordered for delivery. Load balancing is demonstrated in the simulation results for performance question #3 (Section 14.3.2), which showed its impact towards a business process by distributing prescriptions to another pharmacy based on the condition that they were of delivery type. In the 3 batch simulation runs of model #5 (Section 11.1) on average 52.5% of the prescription cases of delivery type were distributed to another pharmacy.

15.2.5 Human Error and Unpredictability

The human error and human unpredictability feature needs to reflect human beings’ autonomous behaviour, i.e. even if people are correctly instructed, they may or may not perform their allocated tasks. Human error is a result of mistakes that should be captured and removed through error detection methods. The StADy *State* and *StateInstance* elements are used to capture the case's current, preceding and succeeding states. In order to reflect people not performing tasks or creating errors in a task, *skip* GT rules are defined and set to low probability distributions. These GT rules move the state into the next state without changing any of the case's content, while a person is assigned to perform a role to that particular case. Hence, the person is correctly instructed yet does not perform his or her allocated tasks. *Backtrack* GT rules are defined for an error detection routine, such that the internal content of the case will be validated and if it is not consistent with the state label then it will be reverted to the previous state. The state diagram in Figure 7.4 outlined the required state order of the pharmacy business process, hence if a worker skips the *type* state error detections will later backtrack the case in the *print* state. In the 3 batch simulation runs of model #5 (Section 11.1) application of these two rules are as follows: skip type= 33.00 and backtrack print state= 16.667. This illustrates the errors that have been detected thus far in a simulation period. Skip and Backtrack rules are further discussed in Section 7.3.4

15.3 Language Requirements

This section specifies the location of the language feature requirements coverage in the StADy approach.

15.3.1 Visual Representation

The visual representation feature requires that there should be a visual notation to define the business process being modelled. This was accomplished with the development of the StADy domain specific language, such that the concrete syntax graphically describes the structure of the business process being modelled in a pictorial form. The StADy concrete syntax is defined in Sections 6.3 and 6.4. Chapter 11 illustrates its use in stage 2 of the methodology.

15.3.2 Stochastic Specification of Task Selection and Duration

The language required a stochastic component in order to attribute time delays or probability distributions. Therefore the stochastic graph transformations approach was employed as described in Chapter 7. Task 2.7 (Section 11.7) illustrated how to select the probability distribution values, whereas task 3.5 (Section 12.5) illustrated how to encode the probability distribution values into the GraSS XML parameter files.

15.3.3 Flexible Specification of Unstructured Aspects

The language achieved flexibility by using the graph transformation rule-based approach as described in Chapter 7 and illustrated in tasks 2.3-2.4 (Sections 11.3-11.4), 3.2-3.3 (Sections 12.2-12.3). The rule-based approach provided a means to permit dynamic non-deterministic decisions, by specifying rules to define how the system's state may change. Non-deterministic behaviour remains in the pattern matching of the stochastic graph transformation rules, even though non-determinism is partially restricted in the stochastic simulation by the probability distributions. There is still complete freedom in any individual choice, but statistically choices are known for large numbers of rules.

15.4 Method Requirements

This section specifies the location of the method feature requirements coverage in the StADy approach.

15.4.1 Integrate with Standard Business Modelling

The language requires integration to existing standard modelling languages such as Unified Modeling Language (UML) and Business Process Modelling Notation (BPMN). This feature enables the ability to represent the model at different levels of abstractions in other existing modelling languages. StADy process execution design elements such as hierarchy model, artifact model and the GT rules are integrated with standard UML notation in the business modelling element, as specified (Chapter 9) and illustrated in stages 1-2 in the methodology (Chapters 10-11).

15.4.2 Simulation Mechanism

The language requires simulation mechanisms in order to provide a flexible analysis technique for testing protocols prior to employing them in day to day routine. The StADy approach uses existing Graph-based Stochastic Simulation (GraSS) tool [30, 68] for running stochastic simulations, as described (Chapter 9) and illustrated in stages 3 and 4 in the methodology (Chapters 12-13). The GraSS simulation tool has been tested in a number of other domains such as Peer to Peer (P2P) networks and biological modelling [68].

15.5 Application Scenario

The following section illustrates the concept requirements in an application scenario using the StADy graphical syntax. A sequence of instance graphs resulting from the application of defined GT rules (Sections 7.2, 7.3 and 11.4) in particular dynamic reassignment, assignment polices, scheduling by priority, human error and non-deterministic time is presented.

The initial model in Figure 15.1 has one registered pharmacist (Cindy), one cashier (Bob) and one active dispense medication case (d0) present. Both of the workers are currently available to take on jobs. A patient by the name of Emily is waiting for her prescription to be filled. The current clock time is 11:34:41 am on March 1 2010 and Emily's prescription case is a refill prescription that arrived yesterday (28 February 2010) at 11:35 am. She was informed that the expected completion time was 11:35 am on 1 March 2010. Since, d0's deadline is approaching, it is currently at escalation level 1. Also the current state of the case is *check*, hence a dispensing pharmacist is required to check the filled prescription.

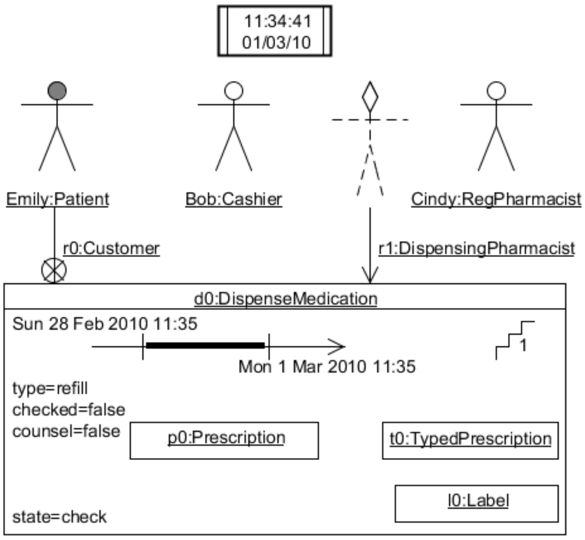


Figure 15.1: Scenario Step 1: On 01/03/10 at 11:34:41

Twenty seconds later (Figure 15.2) a temporal escalation is triggered (R2.3-trigger GT rule) on *d0* case, raising the escalation level to two and temporarily permitting (R1.2) the cashier to take on the role as a filling technician for that case. Also, Cindy is assigned (R1.1- assignment GT rule) to check the filled prescription.

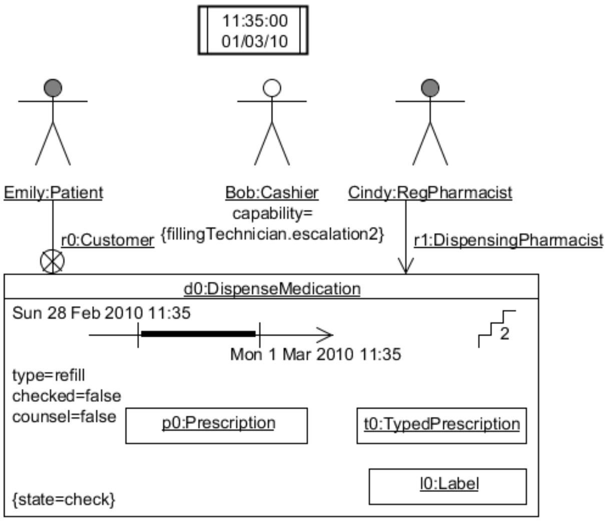


Figure 15.2: Scenario Step 2: On 01/03/10 at 11:35:00

A few minutes later (Figure 15.3) a new high priority delivery prescription (d1) case arrives. At the same time, as Cindy attempts to check the filled prescription in case d0, she realizes that the case is incomplete because the filled prescription artifact is missing. The *check* state backtrack rule verifies the existence of a filled prescription artifact through pattern matching. The check prescription GT rule would not match to a case in *check* state if there exist a missing filled prescription artifact. This results in Cindy backtracking (R2.5- backtrack GT rule) the d0 case to the *fill* state. Cindy leaves the d0 case aside for a filling technician to fill the prescription. Meanwhile, the new case requires an entry technician to type the prescription as shown in Figure 15.4.

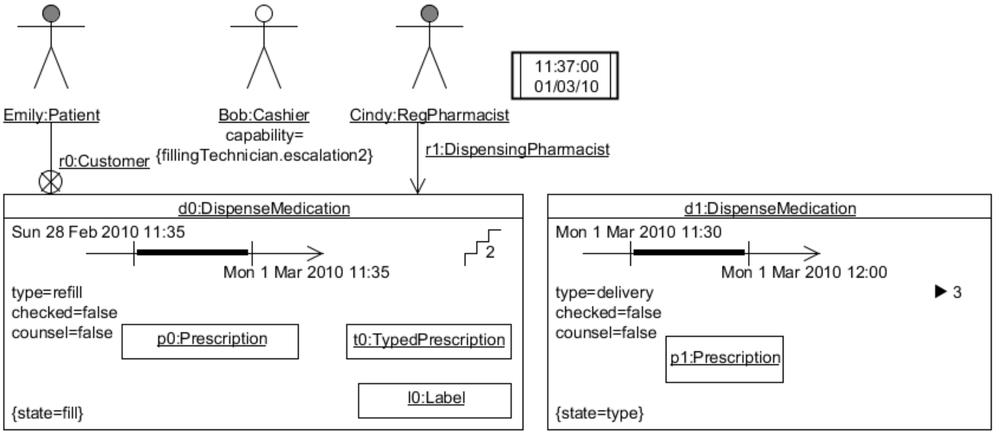


Figure 15.3: Scenario Step 3: On 01/03/10 at 11:37:00

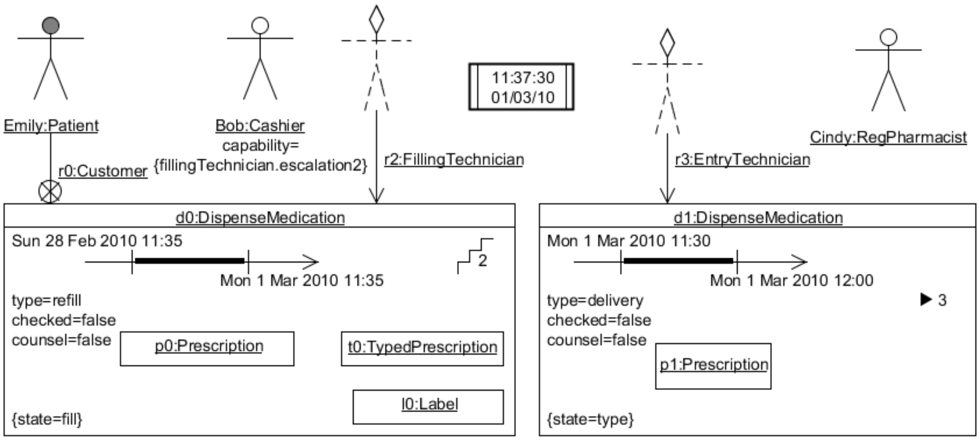


Figure 15.4: Scenario Step 4: On 01/03/10 at 11:37:30

The assignment (R1.1- assignment GT rule) of the entry technician and filling technician roles are shown in Figure 15.5. Cindy is assigned to take on the entry technician role and Bob is assigned to take on the filling technician role, due to the predefined assignment policy (R1.1), which states that the least qualified available worker able to do the job should be assigned. A minute later Cindy types d1's prescription and sends it to the printer, as shown in Figure 15.6. Within a few seconds Bob completes filling d0's prescription, also d1's label is printed, as shown in Figure 15.7.

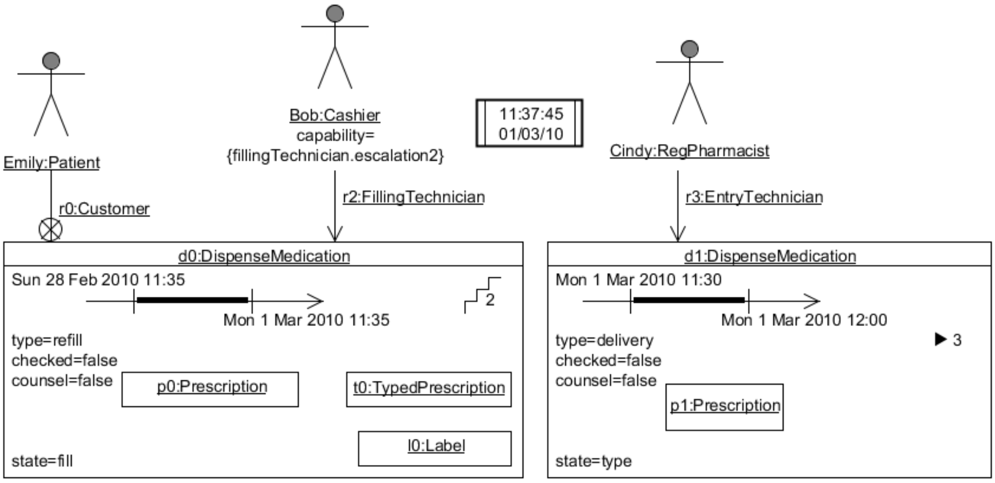


Figure 15.5: Scenario Step 5: On 01/03/10 at 11:37:45

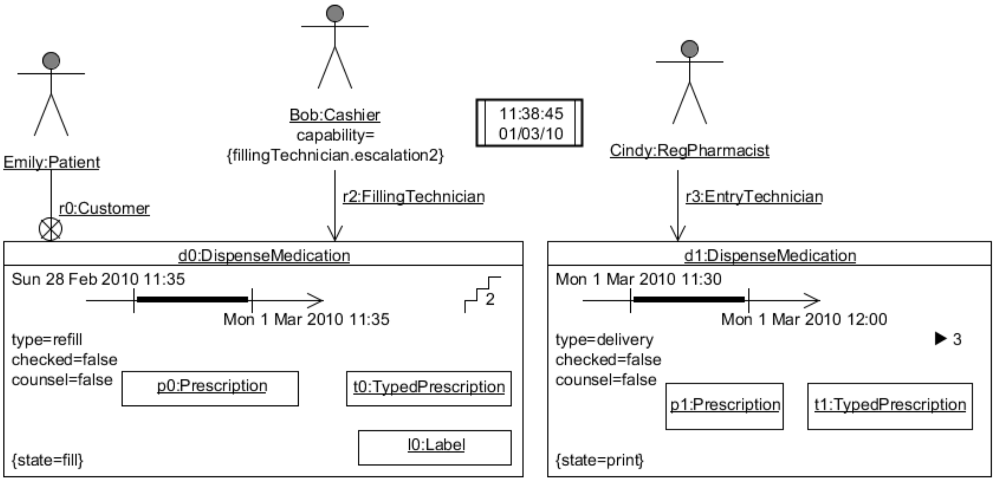


Figure 15.6: Scenario Step 6: On 01/03/10 at 11:38:45

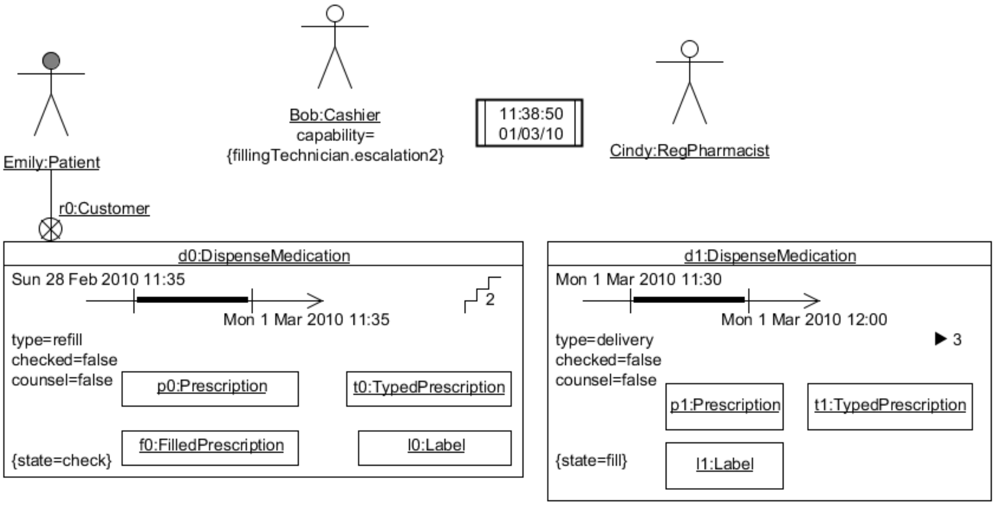


Figure 15.7: Scenario Step 7: On 01/03/10 at 11:38:50

At 11:38:50 d0 requires a dispensing pharmacist to check the filled prescription, whereas d1 requires a filling technician to fill the prescription, as shown in Figure 15.8. Cindy is the only qualified worker that can take on either role. The task order decision is made based on priority level (R2.1), therefore Cindy is assigned to take on the filling technician role on d1 because it has a higher priority, as shown in Figure 15.9. The pharmacy business process continues with various other tasks.

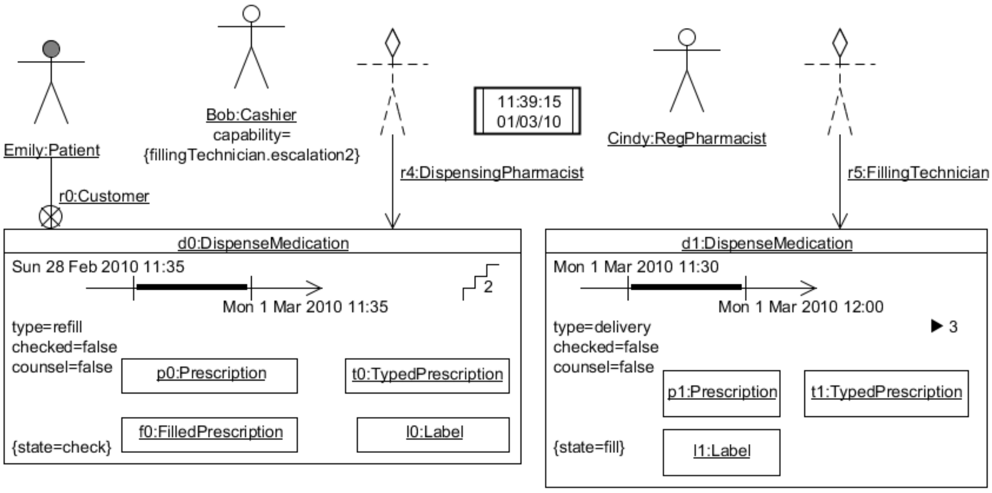


Figure 15.8: Scenario Step 8: On 01/03/10 at 11:39:15

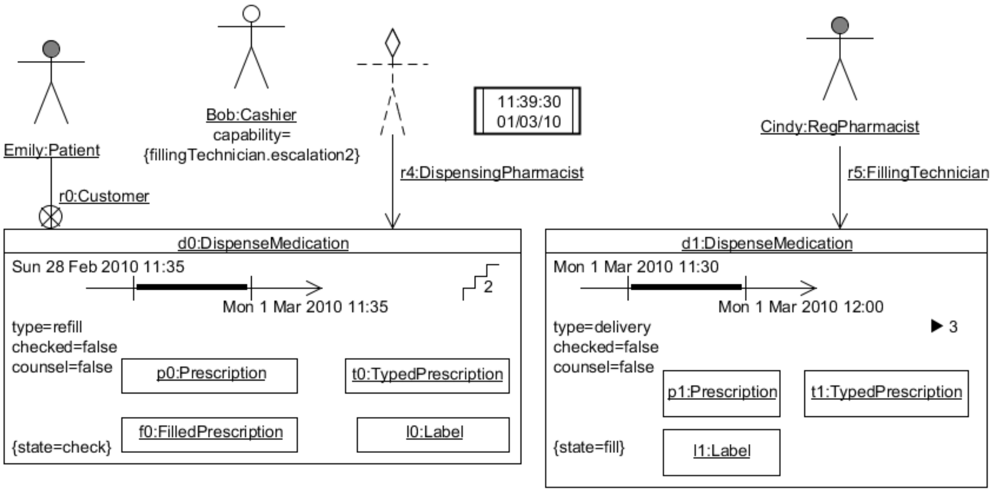


Figure 15.9: Scenario Step 9: On 01/03/10 at 11:39:30

This application scenario is an illustration of the StADy language, because the configuration modelling sub-language (Chapter 6) visually represented steps in the scenario and the transformation modelling sub-language (Chapter 7) was used to transform instance graphs between the steps. The application scenario demonstrated the usefulness of the StADy approach because it was able to describe a realistic scenario using the features (Figure 2.2) available in StADy. The features that were illustrated included: escalation handling (R2.3), temporary promotion (R1.2), dynamic (re)-assignment (R1.1), scheduling (R2.1), human error (R2.5) and role-based access control (R1.3). This application illustrates how the StADy language can visually represent business processes using an easy to understand UML-like notation.

15.6 Threats to Validity

This section discusses possible threats to the validity of the simulation results, because possible errors and bias could arise from decisions made during the process execution design and process encoding stage. During process execution design the start graph was chosen; whereas during the process encoding stage simulation parameters such as simulation depth and distribution values were chosen. Each of these artifacts can influence the simulation results. The following sections discuss and analyse their impact towards the simulation results.

15.6.1 Correctness of GT Rules

Even though all the GT rules were tested there is a possibility of minor errors in the translation of the GT rules from StADy notations into VIATRA2 syntax that may impact the resulting simulation results. Section 8.3 describes how the domain-independent GT rules that were tested, whereas task 3.3 (Section 9.7.3) of the methodology defines consistency constraints for testing the domain-specific GT rules.

15.6.2 Impact of Start Graph Selection

This section discusses the impact of start graph selection by using the start graph defined in the second simulation as an example. Figure 15.10 is a recap of the start graph defined in Section 14.1.4. The initial model has a set of two pharmacies. The first pharmacy consists of one registered pharmacist, two technicians, one cashier, one pharmacy student and two active dispense medication cases. The second pharmacy consists of two registered pharmacists and one technician. Two patients are waiting for their prescription to be filled at store number 1.

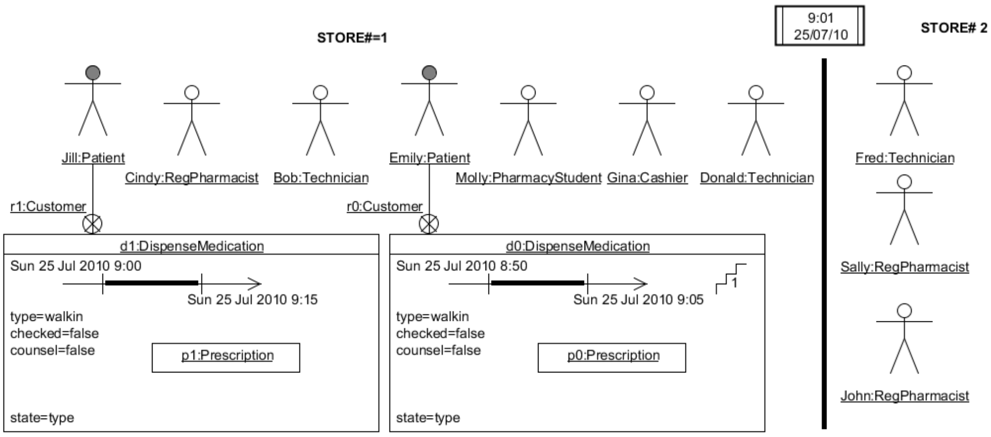
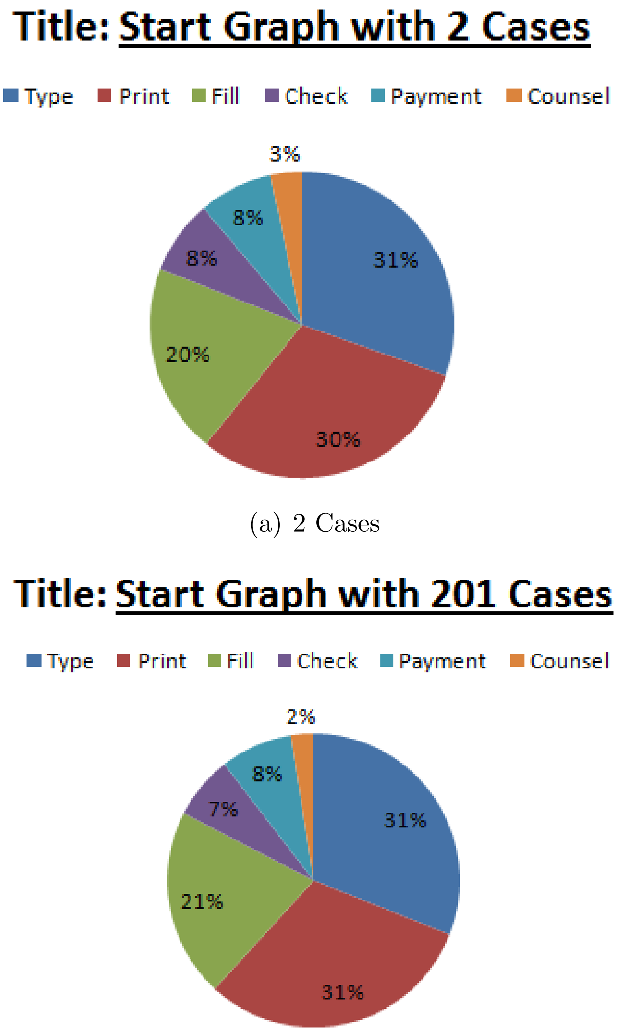


Figure 15.10: Start Graph for Simulation Type 2 Model

To see the effect of the number of initial cases in a start graph, an additional start graph was created with identical parameters which only varied in the number of initial cases i.e. 201 cases opposed to 2 cases. Both simulation models included escalation and load balancing features and simulation depth was set to 2500. Figure 15.11 illustrates the proportion of cases that completed particular states in pie charts. Since the pie charts are almost identical, these results show that the number of initial cases has minimal impact on the results.



(a) 2 Cases

(b) 201 Cases

Figure 15.11: Effect of Start Graph's Number of Cases to State Completion Results

Next, I decided to analyze the impact of increasing qualified workers in the start graph. Hence, the initial start graph with 2 cases was used and modified to increase three technician qualifications to registered pharmacist. The results in Figure 15.12 pie chart, show that the check percentage noticeably increased. This is due to the fact that the check prescription task can only be completed by a registered pharmacist. Hence the start graph influences the results based on the qualifications of the workers to complete tasks. This illustrates that the StADy approach gives us the ability to see how many staff is required in order to achieve a particular completion percentage in a state. For instance, in order to achieve 11% completion rate in the *check* state, six pharmacists are required.

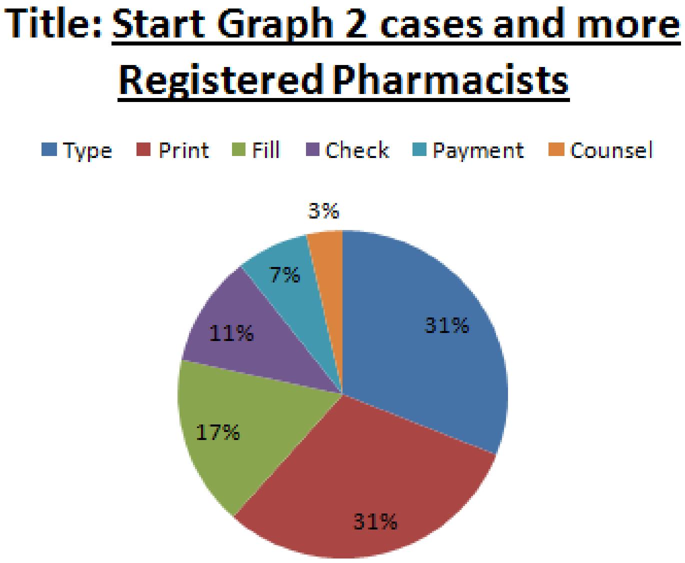


Figure 15.12: Start Graph with More Qualified Workers

15.6.3 Impact of Simulation Depth

This section discusses the impact on the selected simulation max depth value. For all of the simulation the depth was set to 2500 in the parameter files. Figure 15.13 illustrates the effect on case state completion if the simulation depth is 3000 and doubled to 6000. The line graph shows that simulation depth only increases the percentage of cases completed in each state. The overall trend of the results is the same, i.e. high percentage for type state and low percentage for check state.

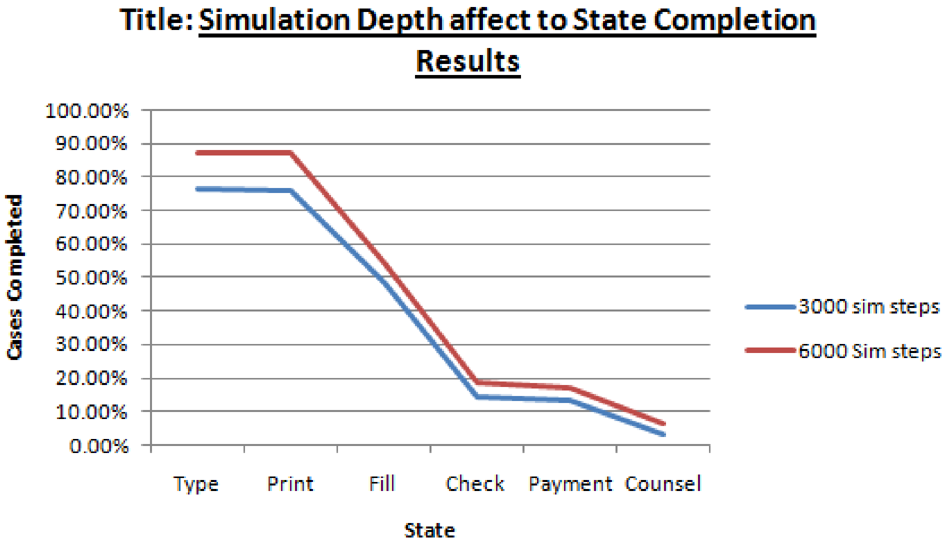


Figure 15.13: Simulation Depth Affect on State Completion

15.6.4 Impact of Distribution Selection

This section discusses the impact of the exponential distribution values for case arrivals. Table 15.1 lists high and low rates that are relative to the selected medium rate for case arrivals.

Event	**Rate** λ **(mins)**		
	high $(0.5 * med)$	**med**	**low** $(1.5 * med)$
NewCase GT Rule	5	10	15
NewCaseMedPriority GT Rule	10	20	30
NewCaseHighPriority GT Rule	30	60	90

Table 15.1: Arrival Rates

Various simulations were run to see if overall tendency ‘checked on time’, ‘counselled on time’ and ‘checked late’ was the same for the 4 simulation models defined in simulation number 2. The results for the ‘checked on time’ feature in the graph in Figure 15.14 show that the tendency for medium and low rates are the same; however at a high rate the results fluctuate. Therefore for the ‘checked on time’ feature escalation is not effective when the rate of arrivals is high; whereas it is an ideal feature when the rate of arrivals is low or medium speed. The results for the ‘counseled on time’ feature in the bar graph in Figure 15.15 shows that at medium speed V5 (escalation,load balancing) and V6 (escalation) are the more favourable simulation models for ensuring that the prescription is counseled to customer on time; however at low speed V5 is 5x more favourable then V6 and at high speed all of the models except V5 did not counsel their prescriptions on time. The ‘counsel on time’ results illustrates how the distribution selection can be extremely sensitive. The results for the ‘checked late’ feature in Figure 15.16 illustrates a tendency that is relatively stable between the three rates, except for few minor fluctuations reflected in the distribution selection of the three rates in the V7 (Load Balancing) model.

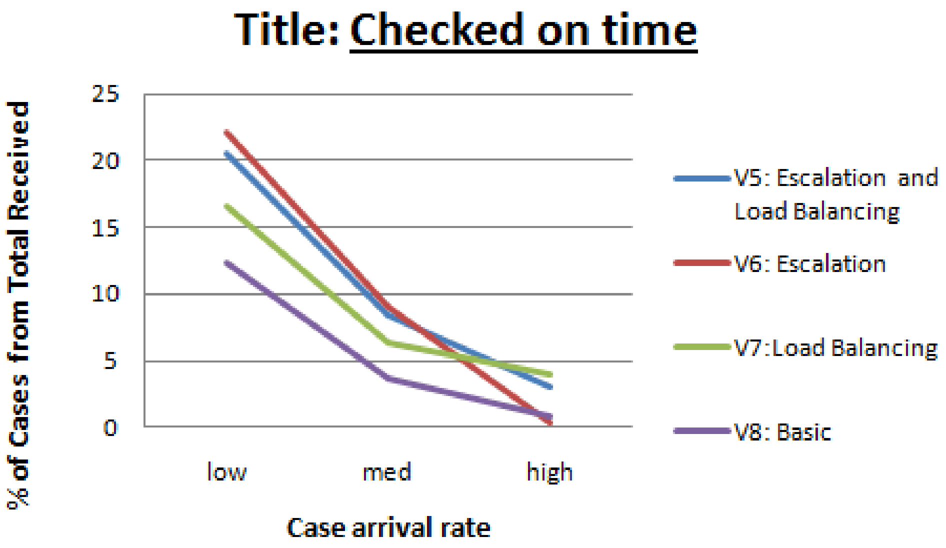


Figure 15.14: Results for Check on Time with Different Arrival Rates

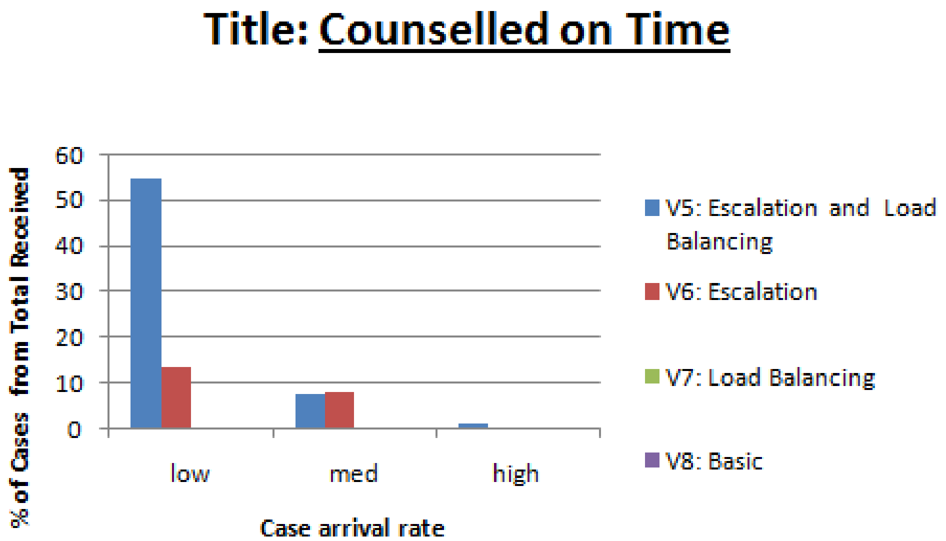


Figure 15.15: Results for Counselling on Time with Different Arrival Rates

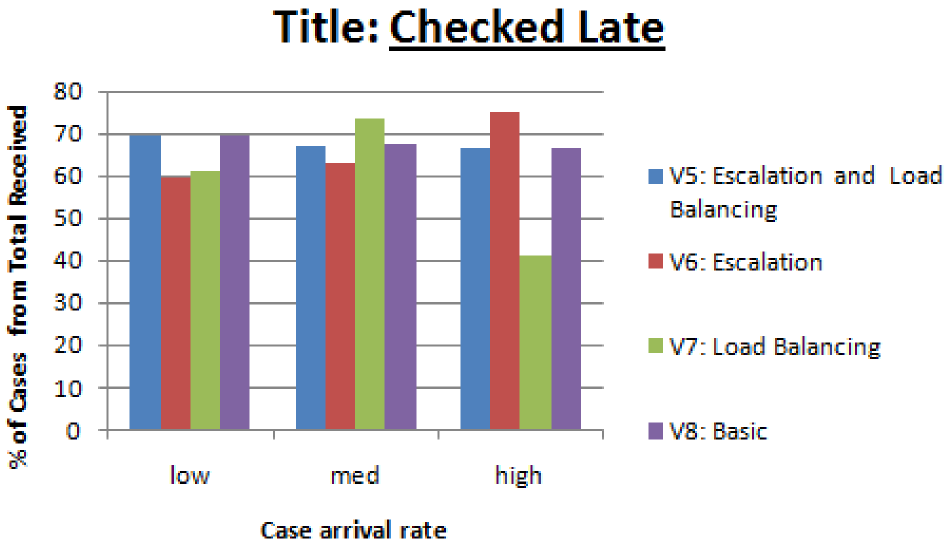


Figure 15.16: Results for Checking Prescription Late with Different Arrival Rates

15.7 Summary

This chapter presents an evaluation of the StADy approach by discussing the requirement coverage, illustrating an application scenario and discussing threats to validity. The next chapter (Chapter 16) will discuss related work in relation to the StADy approach.

Chapter 16

Related Work

This chapter discusses representative languages, approaches and tools that can be used for modelling, executing and analyzing human-resource allocation. The chapter concludes with a summary of the analysis of related work using comparison charts based on the criteria defined in the requirements specification in Chapter 2.

16.1 Modelling Approaches

There are various modelling approaches that can be used for modelling humans in business processes which include abstract, concrete and specialized techniques. Problem Frames [36] are good examples of an existing approach which can be used to define a problem domain abstractly. The Business Process Modelling Notation (BPMN) [47] is common graphical notation that can be used to model people in business processes. Business process ontologies such as Axenath et al.'s [6] can be used for modelling behaviour, information and organisational aspects of a business process. There are also various other specialized techniques such as Depke et al.'s [6] technique for modelling agent-based systems [19], which can be applied to human agents.

16.1.1 Problem Frames

The Problem Frames [36] approach developed by Michael A. Jackson provides an abstract representation of the problem interaction for software requirement specification. It consists of three kinds of diagrams called: context, problem and problem-frame diagrams (R3.1).

Context diagrams focus on customers' needs, responsibility and scope of authority by only displaying interfaces that are within the scope of the customer. StADy also considers the viewpoint of the user by explicitly connecting the user to objects and roles that they have access rights to; however it also displays an overview of the surroundings by connecting the interfaces that go beyond their scope in a single model. The problem diagrams decompose the problem into subproblems and rights and privileges of membership which can be exercised with the addition of annotations. Problem-frame diagrams classify problems into three distinct domains: casual, biddable and lexical. The biddable domain consists of people and their "lack of positive predictable internal causality" [36] (R2.5), which is similar to the domain of human actors in StADy, because both approaches emphasize the fact that humans may be ignorant of their operations and have unpredictable actions. The key difference is in the perspectives: StADy provides a detailed look at the biddable domain whereas problem frame diagrams provide an overview of the interaction across three different domains. A person in this domain spontaneously causes events in a system without external stimulus, which is similar to the StADy notion of pro-activity.

16.1.2 Business Process Modelling Notation (BPMN)

The Business Process Modelling Notation (BPMN) [47] is a DSL for process oriented aspects of an application. Unfortunately it has little graphical support for human-resource modelling, although it provides a representation for roles and their corresponding responsibilities by using *swimlane* notation (R3.1), which is similar to the UML syntax discussed in Section 3.4. Swimlanes provide a static partitioning of activities that can be used to denote a fixed *participant–activity* (R1.3) relation, hence it specifies which actor can perform particular activities by constraining the activities into a single *pool*. The BPMN metamodel contains various elements which do not have a corresponding graphical representation such as *participant*, *process* and *assignment* [65], whereas the StADy models graphically express them. The BPMN *participant* element attributes can be used to specify details such as the participant name pertaining to a corresponding business entity or business role. However, this is not graphically expressed because the swimlanes exclude this high level concept detail. Since BPMN swimlanes can only be used to statically specify which actor can perform particular activities, BPMN does not address the dynamic assignment of roles to individual actors, whereas the StADy approach provides the facility to change the assignment at runtime.

On the other hand, BPMN has a visual representation (R3.1) for escalation (R2.3) and backtracking (R2.5) through *exception flows*, and *compensation flows* respectively. The exception flow denotes a flow that occurs outside normal flow based on a specific trigger (i.e. deadline exceeded) that redirects the flow. Compensation associations are triggered if the outcome of an action is determined to be undesirable and it is necessary to 'undo' the activity. BPMN employs escalation handling and compensation associations in a manner comparable to the specification of the StADy approach. However, StADy approach adds the use of graph transformation rules to specify exception triggers and specifies escalation levels to each case. On the other hand, StADy's backtracking error handling facilities is similar to BPMN's compensation association because it is used to capture undesirable outcomes and undo the preceding activity that caused it.

BPMN provides a formal representation of the business processes. However, it is constrained by some high-level modeling techniques. Currently, Business Process Management (BPM) engines lack the ability to follow an ad hoc process and Business Process Diagrams (BPD) lack the graphical mechanisms to highlight the *point of view* of its participants [47]. These views are important for BPMN users to understand the behavior of the process in relation to the viewer (participant). Other concepts of high-level modelling techniques that are beyond the scope of *BPMN* are as follows: organizational structure and resources, functional breakdowns, data and information models, strategy, and business models [47].

16.1.3 Business Process Ontology

Axenath et al.'s [6] developed an ontology based on three core aspects of business processes: behaviour, information and organisation. They created three distinct metamodels based on the core aspects and within each metamodel, they divided the elements into static and dynamic aspects. The static aspects are modelling aspects such as process and tasks, whereas the dynamic aspects are instance concepts such as cases and resources. The behaviour aspects define the order or partial order that tasks should be executed in. This is accomplished in Axenath et al.'s behaviour metamodel which associates task elements to their initial and final states. The informational aspect defines the artifacts involved in a business process. Axenath et al.'s information metamodel defines documents (artifacts) and associates them to tasks. The organisational aspect defines the structure of the organisation. Axenath et al.'s organisational metamodel specifies which resource can execute particular tasks. The ontology is formalised by UML class diagrams (R4.1), which provides a technical basis of defining the interfaces and a graphical representation (R3.1).

On the other hand, the StADy metamodel is a single model composed of a combination of behaviour, information and organisation core business process aspects. Also the structure of Axenath et al.'s [6] metamodels for core business process aspects is similar to the linguistic class–object metamodel structure used in StADy metamodel because their metamodels are divided into static and dynamic elements, such that class elements are similar to static elements and object elements are similar to dynamic elements.

16.1.4 Agent-based Systems

Depke et al. use graph transformation systems (GTSs) (R3.3) to define agent-based systems [19]. This is beneficial because human actors can be categorized as a kind of autonomous agent with the capacity to regulate and coordinate their own behaviours [50]. In general, an *autonomous agent* is defined as a system situated within an environment that senses that environment and reacts to it over time in pursuit of its own agenda [25]. In other words agents are reactive to their environment and act in pursuit of their own goals. Autonomous agents can be used to model a wide variety of subjects, including software and human agents. Depke et al. [19, 18] agent-oriented modelling technique captures autonomy (R2.5) and cooperation of autonomous agents. Their methodology is based on various UML diagrams (use-case, sequence, class, state) and graph transformation (R4.1) to specify agent operations in a way that is comparable to the specification of the StADy business activities.

Some of the main behavioural characteristics of autonomous agents are *reactivity, autonomy and pro-activity* [19]. *Reactivity* is defined as the capability to be sensitive to the environment and react to changes. *Autonomy* is defined as the ability to make one's own choices regarding the (non-)execution of a task, leading to non-deterministic choices [19]. *Pro-activity* is defined as the determination to reach a certain goal. These characteristics can positively or negatively impact a business process. For instance, because of autonomy, a person could refuse to perform an expected task for his or her assigned role, resulting in another person pro-actively taking over the role in order for a task to be successfully completed. Because of these shared characteristics, modelling techniques used for software agents could also be used for human actors. Like software agents, humans differ from standard software components by exhibiting properties such as reactivity, autonomy and pro-activity. While humans can be regarded as autonomous agents, the StADy approach extends this approach by adding all of the features and performance analysis requirements defined in Chapter 2.

16.2 Executable Approaches

Executable approaches are based on executing encoded instructions and making decisions at run time. There are various executable approaches that capture various features of the specified requirement (Chapter 2); however the prime focus of the thesis is simulating workflows as opposed to interacting with workflows in real time. Hence the thesis is not intended to compete with executable approaches, it is only interested in how they capture role and task allocation. Examples of executable approaches that capture human-resource allocation include: web services, workflow management systems, resource scheduling optimization approaches and specialised software process management techniques.

16.2.1 Web Services for Modelling Human Behaviour

The first vision of the integration of humans in service-oriented environments was presented in a joint whitepaper by IBM and SAP describing an enactment of business processes “beyond the orchestration of activities exposed as Web services" by incorporating "people as an additional type of participant" [34].

WS-HumanTask [2] is an extension of the *WSDL* [4] web service standards which is motivated by requirements for specifying humans as part of service-oriented systems or processes. *WS-HumanTask* can specify task *assignment* (R1.1), recoverable errors (R2.5), and *timeouts*, and triggers appropriate *escalation* (R2.2) actions.

Similarly, *BPEL4People* [64] is an extension of *WS-BPEL 2.0* [63] that incorporates activities performed by people, using capabilities such as data manipulation. Concepts in *BPEL4People* such as human tasks and notifications are inherited from the *WS-HumanTask* specifications, whereas generic human roles and people assignment are extended from *WS-HumanTask 1.0.* They are both broken down into three distinct types. The generic human roles are either process initiator, process stakeholders or business administrators, while people assignments can be achieved by using logical people groups, literals or expressions. The *process initiator* is the person associated with triggering the process instance at its creation time, whereas *process stakeholders* are people who can influence the process instance, by performing actions such as adding ad-hoc (R2.5) attachments [64]. The *business administrator* administrates the process by performing actions such as resolving missed deadlines.

Since the XML syntax of *WS-HumanTask* [2] and *BPEL4People* [64] are intended for execution of web services they are not suitable for domain and business experts because they lack visual representations. The StADy language extends concepts defined in *BPEL4People* and WS-HumanTask by defining it as a high-level model with the addition of stochastic durations, means for defining organisational structures, and facilities for changing the role assignment at run time. For instance, WS-HumanTask's escalation handling protocol is used in a manner comparable to the StADy approach; however the StADy uses graph transformation rules to specify triggers and escalation levels specific to each case.

16.2.2 Workflow Management Systems

A workflow management system (WfMS) [20] is an executable generic application that supports coordination and cooperation within an organisation's workflow. In particular, allocation of work is accomplished through task lists which are given to resource participants based on their availability. Once the task is complete, the workflow software ensures that the individuals responsible for the next task are notified and receive the data they need to execute their stage of the process. There are various existing WfMSs that support human collaboration, such as IBM's FlowMark [33] and Xerox XSoft InConcert [58].

FlowMark uses scheduling information of actors and roles at runtime (R2.1). InConcert is based on a graphical coordination model (R3.1). They both focus on role allocation in distributed systems (R1.1,2.4) using a control flow approach as opposed to a rule based approach.

Exception handling is applied in other WfMSs such as [12, 3]. [12] implements a rule-based exception handler (R2.3,3.3) for workflow management systems, by using directed graphs to represent the flow structures.

MILANO [3] is a net-theoretical modelling framework that captures aspects of activities, roles, control flow and exception handling (R1.1,2.3) in a static and dynamic way.

Although workflow management systems (WfMS) capture basic role allocation properties they do not match StADy's intended goal of providing businesses with the means to test scheduling protocols, policies and regulations, prior to employing them in their day-to-day operation, because the resource planning and resource assignment decisions are integrated with WfMS during real time of a workflow opposed to a simulated workflow. Also the StADy model focuses on human decisions instead of software decisions by taking into account the non-deterministic and often non-predictable behaviour of humans.

16.2.3 Resource Scheduling

Automated schedulers have been created to optimise resource allocation such as ones formalized using Little-JIL [29] (Section 16.3.3) and the General Algebraic Modeling System (GAMS) [14], which are in contrast to the StADy approach because they are automated schedulers as opposed to the simulation of policies influenced by human decisions.

Xiao et al. [77] use Little-JIL (R4.1) to model and simulate (R4.2) dynamic approaches to scheduling. Their dynamic scheduling (R2.1) approach takes into account disruptions that might occur such as incorrect or unexpected process executions (R2.2), sudden arrival of urgent activities and staff turnover. They developed optimisation strategies to determine the balance between stability and utility in a process. A resource repository preserves resources that are available for rescheduling based on a human-resource model (R1.1). This human-resource model contains unique identification for the resources and detailed information such as skill set, workload and experience data. Xiao et al. [77] addresses many of the requirements by representing capabilities, priorities (R2.1), and constraints in a graphical syntax (R3.1), but use this data to optimise schedules at runtime, whereas the StADy approach selects the scheduling protocols prior to execution and simulates policies influenced by human decisions.

The issue of optimum allocation is explored and implemented using a high-level modeling system for mathematical programming and optimization called General Algebraic Modeling System (GAMS) in [42]. They employ a real-time scheduling algorithm in order to ensure an acceptable quality of service (QoS) (R2.1) to the user. The mathematical notation expresses resources, application, services and time intervals (R2.2) for advanced reservations. The mathematical formalization is in terms of constraints and objects of the function to be optimised. A stochastic service level agreement (SLA) (R2.2), is based on information about minimum probability that a time constraint is respected and minimum probability that a workflow is actually available (R3.2). Konstanteli et al.'s [42] approach is used to decide what the optimum allocation of the workflow to available resources are, based on revenue gain and penalties on QoS constraints. The StADy approach is similar; however it uses human resources as opposed to processor resources. It is also defined in a visual domain specific visual notation, hiding the mathematical formalisation from the user.

There are various other resource scheduling methods in the manufacturing domain [1, 16, 49, 53], however resources in manufacturing are usually machines hence do not usually possess the same characteristics as humans.

16.2.4 DYNAMITE

Dynamic Task Nets for software process management (DYNAMITE) [32] uses a formalism based on graphs and graph transformation rules. The DYNAMITE approach is intended to be used to model dynamic tasks in software development and maintenance processes at the generic, specific and instance modelling levels. Task nets are composed of task entities, task interfaces and task realisations. The task entity describes the work that needs to be done, whereas the task interface provides details on what to do in terms of input, output, precondition, postcondition, start dates, and due dates. Task realisation defines how to do the work in terms of atomic (without subtasks) or complex (with subtasks) structures. Ordering (R2.1) of the subtasks is based on the control-flow structure of an acyclic graph, which includes feedback relations to capture error detection (R2.5). The formal specifications of this approach is defined using PROGRES [59] graph rewriting system tool. The task net generic model is composed by graph schema (type graph) and a set of productions (graph transformation rules), which are defined at the meta model level (R3.3). DYNAMITE [32] captures various StADy requirements features; however the scope of this approach is different from StADy because actor (re)-assignment and user-role views are only briefly discussed, whereas the structure of task nets is emphasised.

16.3 Analysis Techniques

Some of the techniques that can be used for analysing human-resource allocation use simulations. There are various simulation tools used in industry as well as a number of research-oriented environments which are mostly based on the flow-oriented style of modelling, such as ADONIS [10] and Little-JIL [29], respectively. There are also specialised analysis models such as Koch et al.'s security policy analysis model [41].

16.3.1 Stochastic Simulations

There are implementations of stochastic simulation based on process algebra, such as the Performance Evaluation Process Algebra (PEPA) [27] and stochastic π-calculus (Sπ) [51] (R4.2). PEPA is an algebraic language which can be used for capturing performance information of computer systems by building models. Sπ is an extension of π-calculus which is intended for performance modelling. In both approaches activities are associated with exponential distribution in order to use a random variable to calculate the duration (R3.2). Hence, when an activity is enabled, it will delay for a period determined by the associated distribution. This method is similar to the GraSS simulation algorithm (Section 4.4) used in the StADy methodology (Section 9), however GraSS also permits activities to be associated to normal distribution. On the hand, Semi-Markov PEPA [11] and extended stochastic π-calculus [52] are extension of PEPA and Sπ that allow activities to be associated with general distributions, as done in GraSS. Despite the fact that Semi-Markov PEPA and extended stochastic π-calculus use similar simulation algorithms to schedule activities, they do not capture the majority of the requirements outlined in Chapter 2 because they are intended for simulating a broader range of processes, whereas the StADy approach focuses on analysing resource allocation protocols on business processes.

16.3.2 ADONIS

ADONIS is an industrial simulation tool (R4.2) that is intended for companies to model and simulate processes in order to reach ISO 9000 quality standards. The ADONIS meta-model is composed of various existing modelling techniques such as BPMN and UML (R4.1). Hence, it captures the same modelling requirements discussed in Section 16.1.2 (R1.3,2.3,2.5,3.1) with the addition of performance specification, and performance analysis. The simulation option provides the facility for business process optimization through various analysis techniques such as path and cycle time analysis. Path analysis reconsiders process flows by testing different possibilities, whereas cycle time analysis is based on duration estimates to calculate waiting time, resting time and transport time. The ADONIS software provides static or dynamic evaluation on ADONIS models. This modelling and simulation tool is used as a broader scope which results in it only capturing general aspects of resource allocation (R1.1), whereas the StADy approach specialises in resource allocation.

16.3.3 Little-JIL

Little-JIL is a domain-independent visual agent-coordination language for modelling (R3.1) and simulating processes (R4.2). An agent may be human or automated. A *step* is one unit of work assigned to an agent specified in pre and post-requisites, which are written as annotations on top of the *step*. Little-JIL *step* is similar to the GT rules defined in the StADy approach except in Little-JIL the pre and post-requisites are written as annotations on top of a *step*, whereas the pre and post conditions in StADy approach are graphically represented in the graph transformation rules.

Little-JIL employs various features such as first-in-first-out FIFO scheduling routine (R2.1), opt-out agent capability (R2.5), exception handling (R2.3) mechanisms and deadlines (R2.1). However Little-JIL's focus is primarily on the process steps, with very little emphasis on human-resource allocation. The authors in [54], proposed a resource allocation method using Little-JIL, which incorporates timing constructs to specify the minimum and maximum time (R2.2) required to complete a task on their process model. The Little-JIL simulation language JSim (R4.1), has a means for specifying timing elements (R2.2,3.2) based on fixed, linear range and triangle range values, as opposed to the StADy methodology which uses normal and exponential distributions to define task durations in GraSS. The use of distributions provides an accurate depiction of human behavior, because the time it takes to complete a task varies depending on factors.

16.3.4 Security Policy Model

Graph transformation systems (GTSs) (R3.3) are used in analysis models for security policy frameworks [41], such that the formalisms are based on graphs and graph transformation rules.

Koch et al.'s security policy framework [41] is intended to be used for comparing policy models, such as role-based (R1.1), lattice-based and access control (R1.3) lists. The analysis models are formalised using graph transformation systems (GTS) (R3.3). The models use GTS and a combination of negative and positive constraints to detect inconsistencies in policies. The StADy language also uses a similar approach to Koch et al.'s security policy framework [41] where access control is concerned, but integrates this aspect with other business process aspects. As opposed to comparing security protocols, it uses one standard role-based policy and focuses on being an analysis model for comparing schedule and assignment protocols.

16.4 Comparison to Requirements

Tables 16.1 to 16.3 present an overview of the related work's requirement (Chapter 2) coverage. The role management concept requirement table (Table 16.1) shows the degree of which dynamic (re)-assignment, role promotion/demotion and access control are captured in the representative examples. A majority of the selected execution and analysis approaches captured the dynamic re-assignment feature (R1.1), with the exception of Little-JIL simulation tool, DYNAMITE and a specialised WfMS exception handler [12], whereas the selected modelling techniques lacked the dynamic re-assignment capabilities. On the other hand, none of the approaches captured the role promotion, demotion and temporary promotion feature (R1.2). This could be due to the fact that a majority of them tend to focus on people being directly assigned to a task as opposed to a particular role. Also, the access control feature (R1.3) is not captured by the execution approaches and is only sparsely captured by some of the modelling and analysis techniques such as BPMN and the ADONIS simulation.

Category	***Related Work***	***Requirement Coverage***		
		R1.1	**R1.2**	**R1.3**
Modelling	Problem Frames			
	BPMN			✓
	Agent-based using GT [19]			
Execution	WS-HumanTask & BPEL4People	✓		
	FlowMark	✓		
	InConcert	✓		
	MILANO	✓		
	WfMS exception handler [12]			
	Resource Scheduler [77, 42]s	✓ [77]		
	DYNAMITE [32]			
Analysis	Stochastic Simulations [27, 51, 11, 52]			
	ADONIS	✓		✓
	Little-JIL			
	Security Policy model [41]	✓		✓

Table 16.1: Related Work vs. Role Management (R1) Requirements

The task management concept requirement table (Table 16.2) shows the degree of which process scheduling, non-deterministic duration of tasks, escalation handling, load balancing, human error and human unpredictability are captured in the representative examples. The selected modelling approaches do not capture process scheduling (R2.1) and non-deterministic durations of tasks (R2.2); however a subset of the execution and analysis techniques capture them such as resource schedulers [77, 42] and Little-JIL simulation tool. All of the categories have at least one representative that supports escalation handling (R2.3) such as BPMN, MILANO and Little-JIL, whereas the load balancing (R2.4) is found in a few executional workflow management systems. On the other hand, human error and unpredictability (R2.5) is captured in all of selected modelling approaches and a subset of the executional and analysis approaches.

Category	***Related Work***	***Requirement Coverage***				
		R2.1	**R2.2**	**R2.3**	**R2.4**	**R2.5**
Modelling	Problem Frames					✓
	BPMN			✓		✓
	Agent-based [19]					✓
Execution	WS-HumanTask & BPEL4People		✓	✓		✓
	FlowMark	✓			✓	
	InConcert				✓	
	MILANO			✓		✓
	WfMS exception handler [12]			✓		
	Resource Schedulers[77, 42]	✓	✓			
	DYNAMITE [32]	✓				✓
Analysis	Stochastic Simulations [27, 51, 11, 52]					
	ADONIS					
	Little-JIL	✓	✓	✓		✓
	Security Policy Model [41]					

Table 16.2: Related Work vs. Task Management (R2) Requirements

The language and methodology requirement table (Table 16.3) identifies languages, which are domain specific, use flexible approach, have a visual representation and contain stochastic elements. This table also determines if the method was integrated with standard business modelling and if there exist simulation mechanism in the representative methodology. Graphical syntax (R3.1) is used in the majority of the representatives; however it is not captured in web services and resource schedulers discussed. On the other hand, the stochastic requirement (R3.2) is only covered in stochastic simulations [27, 51], Little-JIL and Konstanteli et al.'s resource scheduler [42]. Similarly, integration with standard business modelling techniques (R4.1) and simulation mechanisms (R4.2) are all sparsely captured.

Category	*Related Work*	*Requirement Coverage*				
		R3.1	**R3.2**	**R3.3**	**R4.1**	**R4.2**
Modelling	Problem Frames	✓				
	BPMN	✓				
	Agent-based [19]	✓		✓	✓	
Execution	WS-HumanTask & BPEL4People					
	FlowMark	✓				
	InConcert	✓				
	MILANO	✓				
	WfMS exception handler [12]			✓		
	Resource Schedulers [77, 42]	[77]	✓ [42]		✓ [77]	✓ [77]
	DYNAMITE [32]			✓		
Analysis	Stochastic Simulations [27, 51, 11, 52]		✓			✓
	ADONIS	✓			✓	✓
	Little-JIL	✓	✓			✓
	Security Policy Model [41]	✓		✓		

Table 16.3: Related Work vs. Language (R3) and Methodology Requirements (R4)

Since the existing approaches do not completely support all of the specified requirements, there was a need for an approach to encompass all of the requirements in order to accurately model and simulate the dynamic behaviour of humans in business processes while taking into account the flexibility of human behaviour. Hence, the StADy methodology (Chapter 9), StADy configuration modelling language (Chapter 6) and StADy transformation modelling language (Chapter 7) were developed in this thesis.

Chapter 17

Conclusion

This chapter concludes the thesis with a summary of the contributions, a discussion of future work and final concluding statements.

17.1 Summary of Contributions

This thesis proposed a new visual framework based on graph transformation for modeling business processes where human beings are involved. This was achieved, by satisfying the requirements laid out in Chapter 2 as discussed in Chapter 15. StADy modelling language is newly developed consisting of two sub languages called configuration modelling language and transformation modelling language. The StADy configuration modelling language consists of a metamodel, models and a graphical notation as introduced in Chapter 6. On the other hand the StADy transformation language used a graph transformation rule-based approach to model dynamic role allocation by defining business concepts and configurations in terms of the type graph (StADy's metamodel) and instance graphs respectively, as discussed in Chapter 7. The instance graphs can be visually denoted using the StADy graphical notation as illustrated in the application scenario in Section 15.5. StADy's hierarchy and artifact models extended UML's use-case and class diagrams to contain additional information in terms of access control, escalation handling and capabilities. The thesis also defined a catalogue of GT rules (Sections 7.2-7.3 and Appendix A) which captured the following concepts: dynamic (re)-assignment (R1.1); scheduling (R2.1); escalation handling (R2.3); role promotion/demotion and temporary promotion (R1.2); human error and unpredictability (R2.5). Guidelines for translating design models into simulation models are available in Chapter 8.

The thesis also proposed a novel methodology for stochastic modelling and simulation of dynamic human-resource allocation as introduced in Chapter 9. Chapters 10-14 illustrated the use of the StADy methodology on the Shoppers Drug Mart Canada [15] pharmacy franchise, that uses systematic standardized protocols for all of their pharmacies across Canada. The StADy can benefit the pharmacy franchise by testing new protocols prior to deploying them into their nationwide stores. In addition to testing protocols, the graphical syntax can be used to define visual models to aid in training facilities for the employees.

The StADy methodology can be applied to various other businesses that involve various people taking on different roles such as at a bank. A bank has many job positions such as bank teller, bank manager and mortgage brokers. Each of these positions has various particular roles within the bank. For instance, if a bank wanted to test protocols for serving their customers efficiently in a timely manner and prioritize customers by the size of their assets, the StADy methodology priority scheduling protocol can assist in simulating different techniques.

Even though the StADy approach is useful for businesses it unfortunately poses limitation in terms of usability for business users because at the current moment it requires a developer's involvement in the process encoding stage of the methodology. Hence, user-friendly tool support is being considered for future development and is further discussed in Section 17.2.

17.2 Future Work

In this section, a discussion of several possibilities for extending this work in the future is discussed. The future work includes aspects of language expansion, tool development and further quantitative validation. The language expansion aspects are within the scope of thesis, whereas the tool development and quantitative aspects go beyond the scope of this thesis.

Language expansions that have been considered to be added to the concept requirements are aspects of: flexible work hours, malicious unexpected actions, reaction to emergency and removal of job position. Flexible work hours provide the opportunity to model when a person starts or ends his or her shift and when the person is taking a break during his or her shift. The flexible aspect of the work hours is reflected in the fact that, dependent on the current process work load a person may end a shift early or work overtime. This component is partially encoded into the StADy metamodel using the *Person.onDuty* and *Person.arrival* attributes; however it has not been tested. The additional feature to capture malicious unexpected action would need to capture actions that are not expected in the business process, for instance if a customer jumps the pharmacy counter and attempts to fill his or her own prescription. Also, at the current moment the reaction to emergency is partially captured through the use of priorities. In the future it can be stressed strongly by permitting skipping of procedural steps. For instance, if a customer runs into the pharmacy with a diabetic attack then procedures should be overruled permitting any worker to quickly provide the customer with insulin. An additional requirement that could be modelled is the aspect of firing a worker, because at the current moment actor or job positions are predefined and non-terminating, unlike temporary role promotion.

StADy approach is intended to help business users; however the encodings in stage 3 of the methodology requires assistance from developers. Therefore, the development of a user-interface intended for business users would be ideal. The tool can visually represent the business process using the StADy graphical syntax and hide the mathematical formalisation from the user, i.e., keeping abstract details in the back-end. Hence, the StADy GT rules would need to automatically translate into the VIATRA2 syntax as opposed to developers manually encoding it. The tool will also need to integrate with existing VIATRA2 and GraSS tools. The development of a user-friendly tool is an interesting new area that can be explored; however it goes beyond the modelling and analysis thesis scope.

Stochastic simulations were used to test the hypotheses and form general conclusions. These conclusions were validated using quantitative analysis of numerical data gathered from a pilot project. Empirical evidence from the pharmacy confirms the results of the model, but a more thorough comparison is still outstanding. With a more complete model and systematic derivation of distribution parameters from real-life statistics, such results could be used by a pharmacy to increase their productivity. In the future, further evaluation of the simulation results through quantitative analysis would be ideal by comparing the results to observations at pharmacies that performed the exact same protocols and then comparing the validity of the data. However, since the pharmacy case study is only intended to illustrate the use the StADy approach this thesis placed more emphasis on the actual approach oppose to empirical study at a pharmacy.

17.3 Concluding Statements

This thesis proposed a visual framework based on graph transformations (Chapters 6 and 7) for modeling business processes where human beings are involved. With respect to other formalisms, the emphasis is placed on several requirements (Chapter 2) that such a modeling framework has to satisfy in order to represent faithfully and as completely as possible the interactions with human beings. The achievement of each of these requirements by the StADy approach was discussed in Chapter 15.

The StADy methodology consists of four stages, which were described in Chapter 9 and illustrated in Chapters 10-14. The initial stages involve modelling of the business processes, whereas the final stage consists of quantitative analysis based on stochastic graph transformation which is a useful mechanism to answer relevant questions about the process being modelled.

Chapter 16 outlined the current state of the art relative to the requirements defined in Chapter 2 and concluded that the StADy approach can handle dynamic changes in the processes and in particular dynamic allocation of resources better than current approaches.

The thesis presented a new innovative approach for stochastic modelling and simulation of dynamic human-resource allocation, by using a unique application to visual modelling and graph transformations.

Appendix A

StADy GT Rule VIATRA2 Syntax Catalogue

A.1 Managerial-Level Rules

The following section illustrates and discusses the managerial GT rules in terms of the features defined in Figure 2.2.

A.1.1 Dynamic (Re)-assignment (R1.1)

Request Rule

```
gtrule Rule_Request() =
   {
           precondition pattern lhs(Case_,Role_) =
            {
                Case(Case_);
                Role(Role_);
                State(State_);
                StateInstance(StateInstance_);

                Case.currentState(R6,Case_,StateInstance_);

                typeOf(State_,StateInstance_);
                State.requires(R5,State_,Role_);
                neg find RoleInstance(Case_,Role_);
            }

            action {
                let
                      NewRoleInstance_=undef,
                      R1_=undef

                in seq {
                    new(RoleInstance(NewRoleInstance_)in DSM.model.M1);
                    new(instanceOf(NewRoleInstance_,Role_));
                    new (RoleInstance.presence(R1_,NewRoleInstance_,Case_));
                    println(name(Role_)+" Requested ");
            }
        }
    }
```

Assignment Rule

```
     gtrule Rule_AssignPerson() =
   {
           precondition pattern lhs(Case_,RoleInstance_, Person_, Free_, Role_) =
            {
                Case(Case_);
                RoleInstance(RoleInstance_);
                Person(Person_);
                Actor(Actor_);

                //RoleInstance is not assigned
                neg find isAssign(RoleInstance_);


                //Role to fill
                Role(Role_);

                instanceOf(RoleInstance_,Role_);

                //Find qualified person
                Actor.permitted(R2_,Actor_,Role_);
                find  freePerson(Actor_,Person_, Free_, Case_);
            }
            action {

                println ("Person " +name(Person_ )+" is assigned to "+name(Role_)+" role");
                let
                    R1_=undef

                in seq {
                    new (RoleInstance.assignedTo(R1_,RoleInstance_,Person_));
                    setValue(Free_,"false");
                    println("Role assigned ");
                }
            }
    }
```

Unassign Rule

```
gtrule Rule_UnassignRole() =
  {
          precondition pattern lhs(Person_,Free_, R1_, RoleInstance_, Role_) =
           {
               Case(Case_);
               Role(Role_);

               State(State_);
               StateInstance(StateInstance_);
               RoleInstance(RoleInstance_);

               Case.currentState(R6,Case_,StateInstance_);

               typeOf(State_,StateInstance_);

               RoleInstance.presence(R3_,RoleInstance_,Case_);
               typeOf(Role_,RoleInstance_);
               neg find RequiresState(State_,Role_);
               Person(Person_);
               Person.attr(R2_,Person_,Free_);
               Person.free(Free_);
               RoleInstance.assignedTo(R1_,RoleInstance_,Person_);

           }
           action {

               println ("Person " +name(Person_ )+" is unassigned from "+name(Role_)+"");
               delete (R1_);
               delete(RoleInstance_);

               setValue(Free_,"true");

           }
   }
```

Assignment Policy

```
 gtrule Rule_AssignTechnician2() =
  {
          precondition pattern lhs(Case_,RoleInstance_, Person_, Free_, Role_) =
           {
               Case(Case_);
               RoleInstance(RoleInstance_);

               //RoleInstance is not assigned
               neg find isAssign(RoleInstance_);
               //Find qualified person

               //Role to fill
               Role(Role_);
               check ((name(Role_)=="FillingTechnician") xor (name(Role_)=="EntryTechnician")  );
               instanceOf(RoleInstance_,Role_);
               neg find  freeTechnician(Person2_, Free2_, Case_);
```

```
            find  freePharmacist(Person_, Free_, Case_);
        }
        action {

            println ("Person " +name(Person_ )+" is assigned to a role");
            let
              R1_=undef

            in seq {
                new (RoleInstance.assignedTo(R1_,RoleInstance_,Person_));

                setValue(Free_,"false");
                println("Role assigned ");
            }
        }
}



gtrule RuleAssignPolicy()=
  {
       precondition pattern lhs(Case_,RoleInstance_, Person_, Free_, Role_) =
        {
            Case(Case_);
            RoleInstance(RoleInstance_);
            Person(Person_);
            Actor(Actor_);

            //RoleInstance is not assigned
            neg find isAssign(RoleInstance_);

            //Role to fill
            Role(Role_);

            instanceOf(RoleInstance_,Role_);
            //Find qualified person
            Actor.permitted(R2_,Actor_,Role_);
            find  freePersonPolicy(Actor_,Person_, Free_, Case_);
        }
        action {
            println ("Person " +name(Person_ )+" is assigned to "+name(Role_)+" role");

            let
            R1_=undef

            in seq {
                new (RoleInstance.assignedTo(R1_,RoleInstance_,Person_));
                setValue(Free_,"false");
                println("Role assigned ");
            }
        }
    }
```

A.1.2 Scheduling (R2.1)

By Deadline

```
gtrule Rule_RequestDeadline() =
   {
           precondition pattern lhs(Case_,Role_) =
            {
                Case(Case_);
                Case(Case2_);

                Role(Role_);
                State(State_);
                StateInstance(StateInstance_);
                StateInstance(StateInstance2_);

                Case.currentState(R6,Case_,StateInstance_);
                Case.currentState(R7,Case2_,StateInstance2_);

                typeOf(State_,StateInstance_);
                typeOf(State_,StateInstance2_);

                State.requires(R5,State_,Role_);

                neg find RoleInstance(Case_,Role_);
                neg find RoleInstance(Case2_,Role_);

                Case.deadline(Deadline1_);
                Case.deadline(Deadline_);
                Case.attr1(R1, Case_,Deadline1_);
                find getEndTime(Deadline1_, Day_,Month_,Year_,Hour_,Minute_);
                Case.attr1(R2, Case2_,Deadline_);
                find getEndTime(Deadline_, Day2_,Month2_,Year2_,Hour2_,Minute2_);

                check ((60*toInteger(value(Hour_)) + toInteger(value(Minute_)))
                    <((60*toInteger(value(Hour2_))) +toInteger(value(Minute2_))));

            }

            action {

            let
                NewRoleInstance_=undef,
                R1_=undef

            in seq {
                new(RoleInstance(NewRoleInstance_)in DSM.model.M1);
                new(instanceOf(NewRoleInstance_,Role_));
                new (RoleInstance.presence(R1_,NewRoleInstance_,Case_));
                println(name(Role_)+"Requested ");
            }
          }
    }
```

By Priority

```
gtrule Rule_RequestPriority() =
   {
            precondition pattern lhs(Case_,Role_) =
                    {

                    Case(Case_);
                    Case(Case2_);

                    Role(Role_);
                    State(State_);
                    StateInstance(StateInstance_);
                    StateInstance(StateInstance2_);

                    Case.currentState(R6,Case_,StateInstance_);
                    Case.currentState(R7,Case2_,StateInstance2_);

                    typeOf(State_,StateInstance_);
                    typeOf(State_,StateInstance2_);

                    State.requires(R5,State_,Role_);


                    neg find RoleInstance(Case_,Role_);
                    neg find RoleInstance(Case2_,Role_);


                    Case.priority(Priority_);
                    Case.priority(Priority2_);
                    Case.attr2(R1,Case_,Priority_);
                    Case.attr2(R2,Case2_,Priority2_);

                    //if Case_ higher priority then Case2_
                     check ( toInteger(value(Priority_))
                            >= toInteger(value(Priority2_)));

                    }
            action {
                let
                    NewRoleInstance_=undef,
                    R1_=undef

                in seq {
                    new(RoleInstance(NewRoleInstance_)in DSM.model.M1);
                    new(instanceOf(NewRoleInstance_,Role_));
                    new (RoleInstance.presence(R1_,NewRoleInstance_,Case_));
                    println(name(Role_)+" Requested ");
            }
          }
    }
```

Clock Tick Rule

```
gtrule Rule_ClockTick() =
{
        precondition pattern lhs(Time_,Day_,Month_,Year_,Hour_,Minute_) =
        {
            Clock(Clock_);
            Clock.Time(Time_);
            Clock.attr(R1,Clock_,Time_);
            find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
        }
        action {
               seq {

                        //If increases min would be in same hr
                        if((toInteger(value(Minute_))+1) < 60) seq{
                            setValue(Minute_,toInteger(value(Minute_))+1);
                        }
                        //if increases hr is within same day
                        else if (toInteger(value(Hour_))+1 < 24 ||
                            ((toInteger(value(Hour_))+1==24 &&
                                ((toInteger(value(Minute_))+1)-60)==0 ))) seq{
                            setValue(Minute_,((toInteger(value(Minute_))+1)-60));
                            setValue(Hour_,toInteger(value(Hour_))+1);
                        }
                        //Check if increase day will be in same month
                        else if((toInteger(value(Day_))+1)<31) seq {
                            setValue(Day_,toInteger(value(Day_))+1);
                            setValue(Minute_,((toInteger(value(Minute_))+1)-60));
                            setValue(Hour_,1);
                        }//check if increasing month is within year
                        else if((toInteger(value(Month_))+1)<13) seq{
                            setValue(Month_,toInteger(value(Month_))+1);
                            setValue(Day_,1);
                            setValue(Minute_,((toInteger(value(Minute_))+1)-60));
                            setValue(Hour_,1);
                        }
                        //new year
                        else seq{
                            setValue(Year_,toInteger(value(Year_))+1);
                            setValue(Month_,1);
                            setValue(Day_,1);
                            setValue(Minute_,((toInteger(value(Minute_))+1)-60));
                            setValue(Hour_,1);
                        }
                        if(toInteger(value(Minute_)) >=10 ) seq{
                            setValue(Time_, value(Hour_)+":"+value(Minute_) +" "+
                                value(Day_)+"/"+ value(Month_)+"/"+ value(Year_));
                        }
                        else seq{
                            setValue(Time_, value(Hour_)+":0"+value(Minute_) +" "+
                                value(Day_)+"/"+ value(Month_)+"/"+ value(Year_));
                        }

                    }
            println(" Clock tick!!");
        }
}
```

A.2 Support-Level Rules

The following support level rules are defined using domain specific notation.

A.2.1 Escalation Handling (R2.3)

Trigger Rule

```
gtrule Rule_trigger1() =
   {
           precondition pattern lhs(Case_, Level_) =
            {
                Case(Case_);
                Escalation(Level_);
                neg find Trigger1Exist (Case_);
                check (name(Level_)=="level1");

                Case.deadline(Deadline_);
                Clock.Time(Time_);
                Case.attr1(R1, Case_,Deadline_);
                find getEndTime(Deadline_, Day_,Month_,Year_,Hour_,Minute_);
                find getTime(Time_, Day1_,Month1_,Year1_,Hour1_,Minute1_);

                check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_)))
                    >=((60*toInteger(value(Hour_))) +toInteger(value(Minute_))-5));


            }
            action{

                let
                   R1_=undef

                in seq {
                    new (Case.escalations(R1_,Case_, Level_));
                    println("Case escalated");
                }
       }
   }
```

A.2.2 Role Promotion/Demotion and Temporary Promotion (R1.2)

Temp Assign Rule

```
gtrule TempRule_AssignEntryTechnician1() =
   {
           precondition pattern lhs(Case_,RoleInstance_, Person_, Free_, Role_) =
            {
                Case(Case_);
                RoleInstance(RoleInstance_);

                //RoleInstance is not assigned
                neg find isAssign(RoleInstance_);
                //Find qualified person

                //Role to fill
                Role(Role_);
                check (name(Role_)=="EntryTechnician")  ;
                instanceOf(RoleInstance_,Role_);
                Escalation(Level_);
                Case.escalations(R1,Case_, Level_);
                check (name(Level_)=="level1");
                find  freeCashier(Person_, Free_, Case_);
            }
            action {

                println ("Temp assign cashier  " +name(Person_ )+"
                    is assigned to entrytechnician role");

            let
                  R1_=undef

                in seq {
                    new (RoleInstance.assignedTo(R1_,RoleInstance_,Person_));
                    setValue(Free_,"false");
                    println("Role assigned ");
                }
            }
    }
```

A.2.3 Load Balancing (R2.4)

Transfer

```
gtrule DisRule_TypePrescription() =
    {
            precondition pattern lhs(Case_,RoleInstance_,Role_,Person_,
                TypedPrescription_,  Store_,StateInstance_, NextState_,R6_) =
            {
                Case(Case_);
                ArtifactType(TypedPrescription_);
                //States
                State(State_);
                State(NextState_);
                StateInstance(StateInstance_);

                Case.currentState(R6_,Case_,StateInstance_);

                typeOf(State_,StateInstance_);
                check(name(State_)=="Type");

                State.next(R7,State_,NextState_);

                check(name(TypedPrescription_)=="TypedPrescription");
                find ETassigned (Case_,RoleInstance_,Role_,Person_);
                neg find TypedPrescriptionExist(Case_);
                find PrescriptionExist(Case_);

                AttributeValue(AttributeValue_);
                Case.has(R1,Case_,AttributeValue_);
                AttributeDeclaration(AttributeDeclaration_);
                typeOf(AttributeDeclaration_,AttributeValue_);
                check (name(AttributeDeclaration_) == "type");
                check(value(AttributeValue_)=="delivery");

                Case.groupNo(Store_);
                Case.group(R4, Case_, Store_);
            }
            action {
                let
                    NewArtifact_=undef,
                    R1_=undef,
                    NewStateInstance_=undef,
                    R2_=undef


                in seq {
                    // create Class inside the Case
                    new(Artifact(NewArtifact_)in Case_);
                    //create instanceOf relation with the Artifact:ArtifactType
                    new(instanceOf(NewArtifact_,TypedPrescription_));
                    //create relation between Case and Artifact
                    new (Case.contains(R1_,Case_,NewArtifact_));

                    delete(StateInstance_);
                    delete(R6_);
                    // create Class inside the Case
                    new(StateInstance(NewStateInstance_)in DSM.model.M1);
                    //create instanceOf relation with the Artifact:ArtifactType
                    new(instanceOf(NewStateInstance_,NextState_));
                    new (Case.currentState(R2_,Case_,NewStateInstance_));

                    setValue(Store_,"2");
                    println("Typed Prescription added ");
                }
                println ("Type Prescription for "+name(Case_)+" Case");
            }
    }
```

A.2.4 Human Error (R2.5)

Skip Rule

```
gtrule SkipRule_FillPrescription() =
    {

            precondition pattern lhs(Case_, StateInstance_, NextState_,R6_) =
            {
                Case(Case_);
                ArtifactType(FilledPrescription_);
                check(name(FilledPrescription_)=="FilledPrescription");
                find FTassigned (Case_,RoleInstance_,Role_,Person_);
                neg find FilledPrescriptionExist(Case_);
                find LabelExist(Case_);

                State(State_);
                State(NextState_);
                StateInstance(StateInstance_);

                Case.currentState(R6_,Case_,StateInstance_);
                State.next(R7,State_,NextState_);
            }
            action {
                let
                    NewStateInstance_=undef,
                    R2_=undef
                in seq{
                        delete(StateInstance_);
                        delete(R6_);
                        new(StateInstance(NewStateInstance_)in DSM.model.M1);
                        new(instanceOf(NewStateInstance_,NextState_));
                        new (Case.currentState(R2_,Case_,NewStateInstance_));

                        println("no Filled Prescription added ");
                  }
           }
    }
```

Backtrack Rule

```
 gtrule BacktrackRule_checkState()=
    {
      precondition pattern lhs(Case_, StateInstance_, PrevState_,R6_) =
           {
                Case(Case_);
                AttributeValue(AttributeValue_);
                find RequiresChecked(Case_,AttributeValue_);
                find DPassigned (Case_,RoleInstance_,Role_,Person_);

                neg find FilledPrescriptionExist(Case_,Artifact_,ArtifactType_);


                State(State_);
                State(PrevState_);
                StateInstance(StateInstance_);

                Case.currentState(R6_,Case_,StateInstance_);
                State.previous(R7,State_,PrevState_);
            }
            action {
           let
                NewStateInstance_=undef,
                  R2_=undef
             in seq{
                    delete(StateInstance_);
                    delete(R6_);
                    new(StateInstance(NewStateInstance_)in DSM.model.M1);
                    new(instanceOf(NewStateInstance_,PrevState_));
                    new (Case.currentState(R2_,Case_,NewStateInstance_));

                    println("error (backtrack to fill state)");
            }
          }
    }
```

A.3 Production-Level Rules

A.3.1 New Case Rules

New Walkin Low Priority Case

```
gtrule Rule_NewCase() =
{
    precondition pattern lhs(Process_, Prescription_,Patient_,Customer_,
        Time_, Day_,Month_,Year_,Hour_,Minute_,M2_,State_) =
    {
        Process(Process_);
        ArtifactType(Prescription_);
        check(name(Prescription_)=="Prescription");

        Actor(Patient_);
        check(name(Patient_)=="Patient");

        Role(Customer_);
        check(name(Customer_)=="Customer");

        Clock(Clock_);
        Clock.Time(Time_);
        Clock.attr(R1,Clock_,Time_);
        find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
        M2(M2_);

        State(State_);
        check(name(State_)=="Type");

    }
```

```
action {
    let
        Case_=undef,
        Val1_=undef,
        Val2_=undef,
        Val3_=undef,
        R1=undef,
        R2=undef,
        R3=undef,
        R4=undef,
        R5=undef,
        R6=undef,
        R7=undef,
        R8=undef,
        R9=undef,
        NewArtifact_=undef,
        R1_=undef,
        Person_=undef,
        RoleInstance_=undef,
        Priority_=undef,
        State_=undef,
        Free_=undef,
        StartTime_=undef,
        R2_=undef,
        IntegerDay_= undef,
        R3_=undef,
        IntegerMonth_= undef,
        R4_=undef,
        IntegerYear_= undef,
        R5_=undef,
        IntegerHour_= undef,
        IntegerMin_= undef,
        R6_=undef,
        Deadline_=undef,
        R2R=undef,
        R2_2=undef,
        IntegerDay2_= undef,
        R3_2=undef,
        IntegerMonth2_= undef,
        R4_2=undef,
        IntegerYear2_= undef,
        R5_2=undef,
        IntegerHour2_= undef,
        IntegerMin2_= undef,
        R6_2=undef,
        Store_=undef,
        R10=undef,
        NewStateInstance_=undef
    in seq {

        new(Case(Case_)in M2_);
        rename(Case_,name(Case_)+"Case");
        //create instanceOf relation with the Case:Process
        new(instanceOf(Case_,Process_));
        new(AttributeValue(Val1_)in Case_);
        new(AttributeValue(Val2_)in Case_);
        new(AttributeValue(Val3_)in Case_);
        //create instanceOf relation with the AttributeValue:Process
        new(instanceOf(Val1_,DSM.model.M1.Pharmacy.checked));
        //create instanceOf relation with the AttributeValue:Process
        new(instanceOf(Val2_,DSM.model.M1.Pharmacy.counsel));
        //create instanceOf relation with the AttributeValue:Process
```

```
new(instanceOf(Val3_,DSM.model.M1.Pharmacy.type));
new(Case.has(R1,Case_,Val1_));
new(Case.has(R2,Case_,Val2_));
new(Case.has(R3,Case_,Val3_));
setValue(Val1_,"false");
setValue(Val2_,"false");
setValue(Val3_,"walkin");
new(Case.priority(Priority_) in Case_);
new(Case.attr2(R7,Case_,Priority_));
setValue(Priority_,"-1");

new(instanceOf(NewStateInstance_,State_));
new (Case.currentState(R2_,Case_,NewStateInstance_));


new(Case.groupNo(Store_) in Case_);
new(Case.group(R10,Case_,Store_));
setValue(Store_,"1");

//New case contains a prescription
new(Artifact(NewArtifact_)in Case_);
new(instanceOf(NewArtifact_,Prescription_));
new (Case.contains(R1_,Case_,NewArtifact_));

//Add a person
new(Person(Person_)in DSM.model.M1);
new(instanceOf(Person_,Patient_));
rename(Person_,name(Person_)+"Person");
new (Person.free(Free_)in Person_);
setValue(Free_,"false");
new(Person.attr(R6,Person_,Free_));

//Add customer roleinstance
new(RoleInstance(RoleInstance_)in DSM.model.M1);
new(instanceOf(RoleInstance_,Customer_));

//add roleInstance to Case_
new(RoleInstance.presence(R4,RoleInstance_,Case_));
//assign Person_ to RoleInstance_
new (RoleInstance.assignedTo(R5,RoleInstance_, Person_));


//Starttime defined
new( Case.startTime(StartTime_) in Case_);
new(Case.attr3(R8,Case_,StartTime_));
new(Integer(IntegerDay_) in StartTime_);
setValue(IntegerDay_,value(Day_) );
new(DateTime.day(R2_,StartTime_,IntegerDay_));
new(Integer(IntegerMonth_) in StartTime_);
setValue(IntegerMonth_,value(Month_));
new(DateTime.month(R3_,StartTime_,IntegerMonth_));
new(Integer(IntegerYear_) in StartTime_);
setValue(IntegerYear_,value(Year_));
new(DateTime.year(R4_,StartTime_,IntegerYear_));
new(Integer(IntegerHour_) in StartTime_);
setValue(IntegerHour_,value(Hour_));
new(DateTime.hour(R5_,StartTime_,IntegerHour_));
new(Integer(IntegerMin_)in StartTime_);
setValue(IntegerMin_,value(Minute_));
new(DateTime.minute(R6_,StartTime_,IntegerMin_));

if(toInteger(value(IntegerMin_)) >=10 ) seq{
    setValue (StartTime_, value(IntegerHour_)+":"+value(IntegerMin_)+" "+
        value(IntegerDay_)+"/"+ value(IntegerMonth_)+"/"+ value(IntegerYear_));
}
else seq{
    setValue (StartTime_, value(IntegerHour_)+":0"+value(IntegerMin_)+" "+
        value(IntegerDay_)+"/"+ value(IntegerMonth_)+"/"+ value(IntegerYear_));

}
//deadline defined
new(Case.deadline(Deadline_) in Case_);
new (Case.attr1(R2R, Case_,Deadline_));
new(Integer(IntegerDay2_) in Deadline_);
setValue(IntegerDay2_,toInteger(value(IntegerDay_)));
new(DateTime.day(R2_2,Deadline_,IntegerDay2_));
new(Integer(IntegerMonth2_) in Deadline_);
setValue(IntegerMonth2_,toInteger(value(IntegerMonth_)));
new(DateTime.month(R3_2,Deadline_,IntegerMonth2_));
new(Integer(IntegerYear2_) in Deadline_);
setValue(IntegerYear2_,toInteger(value(IntegerYear_)));
new(DateTime.year(R4_2,Deadline_,IntegerYear2_));
new(Integer(IntegerHour2_)in Deadline_);
setValue(IntegerHour2_,toInteger(value(IntegerHour_)));
new(Integer(IntegerMin2_) in Deadline_);

if((toInteger(value(IntegerMin_))+15) < 60) seq{
    setValue(IntegerMin2_,toInteger(value(IntegerMin_))+15);
    setValue(IntegerHour2_,toInteger(value(IntegerHour_)));
}
else if (toInteger(value(IntegerHour_))+1 < 24 ||
    ((toInteger(value(IntegerHour_))+1==24 &&
    ((toInteger(value(IntegerMin_))+15)-60)==0 ))) seq{
        setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
        setValue(IntegerHour2_,toInteger(value(IntegerHour_))+1);
}//assumption that all months have 31 days
else if((toInteger(value(IntegerDay_))+1)<31) seq {
    setValue(IntegerDay2_,toInteger(value(IntegerDay_))+1);
    setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
    setValue(IntegerHour2_,1);
}
else if((toInteger(value(IntegerMonth_))+1)<13) seq{
    setValue(IntegerMonth2_,toInteger(value(IntegerMonth_))+1);
    setValue(IntegerDay2_,1);
    setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
    setValue(IntegerHour2_,1);
}
else seq{
    setValue(IntegerYear2_,toInteger(value(IntegerYear_))+1);
    setValue(IntegerMonth2_,1);
    setValue(IntegerDay2_,1);
    setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
    setValue(IntegerHour2_,1);
}
new(DateTime.hour(R5_2,Deadline_,IntegerHour2_));
new(DateTime.minute(R6_2,Deadline_,IntegerMin2_));

if(toInteger(value(IntegerMin2_)) >=10 ) seq{
    setValue (Deadline_,   value(IntegerHour2_)+":"+value(IntegerMin2_)+" "
        +value(IntegerDay2_)+"/"+ value(IntegerMonth2_)+"/"+ value(IntegerYear2_));
}
else seq{
    setValue (Deadline_,   value(IntegerHour2_)+":0"+value(IntegerMin2_)+" "
        +value(IntegerDay2_)+"/"+ value(IntegerMonth2_)+"/"+ value(IntegerYear2_));
```

```
            }
                setAggregation(R2_,true);
                setAggregation(R3_,true);
                setAggregation(R4_,true);
                setAggregation(R5_,true);
                setAggregation(R6_,true);
                setAggregation(R2_2,true);
                setAggregation(R3_2,true);
                setAggregation(R4_2,true);
                setAggregation(R5_2,true);
                setAggregation(R6_2,true);

                println("Case added ");
        }
    }
}
```

New Medium Priority Delivery Case

```
    gtrule Rule_NewCaseMedPriority() =
    {
        precondition pattern lhs(Process_, Prescription_,Patient_,Customer_,
            Time_, Day_,Month_,Year_,Hour_,Minute_,M2_, State_) =
        {
            Process(Process_);
            ArtifactType(Prescription_);
            check(name(Prescription_)=="Prescription");

            Actor(Patient_);
            check(name(Patient_)=="Patient");

            Role(Customer_);
            check(name(Customer_)=="Customer");

            Clock(Clock_);
            Clock.Time(Time_);
            Clock.attr(R1,Clock_,Time_);
            find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
            M2(M2_);
            State(State_);
            check(name(State_)=="Type");
        }
    action {
            let
                Case_=undef,
                Val1_=undef,
                Val2_=undef,
                Val3_=undef,
                R1=undef,
                R2=undef,
                R3=undef,
                R4=undef,
                R5=undef,
                R6=undef,
                R7=undef,
                R8=undef,
                NewArtifact_=undef,
                R1_=undef,
                Person_=undef,
                RoleInstance_=undef,
                Priority_=undef,
                Free_=undef,
                StartTime_=undef,
                R2_=undef,
                IntegerDay_= undef,
                R3_=undef,
                IntegerMonth_= undef,
                R4_=undef,
                IntegerYear_= undef,
                R5_=undef,
                IntegerHour_= undef,
                IntegerMin_= undef,
                R6_=undef,
                Deadline_=undef,
                R2R=undef,
                R2_2=undef,
                IntegerDay2_= undef,
                R3_2=undef,
                IntegerMonth2_= undef,
                R4_2=undef,
                IntegerYear2_= undef,
                R5_2=undef,
                IntegerHour2_= undef,
                IntegerMin2_= undef,
                R6_2=undef,
                State_=undef,
                R9=undef,
                Store_=undef,
                R10=undef,
                NewStateInstance_=undef
            in seq {
                new(Case(Case_)in M2_);
                rename(Case_,name(Case_)+"Case");
                //create instanceOf relation with the Case:Process
                new(instanceOf(Case_,Process_));
                new(AttributeValue(Val1_)in Case_);
                new(AttributeValue(Val2_)in Case_);
                new(AttributeValue(Val3_)in Case_);
                //create instanceOf relation with the AttributeValue:Process
                new(instanceOf(Val1_,DSM.model.M1.Pharmacy.checked));
                //create instanceOf relation with the AttributeValue:Process
                new(instanceOf(Val2_,DSM.model.M1.Pharmacy.counsel));
                //create instanceOf relation with the AttributeValue:Process
                new(instanceOf(Val3_,DSM.model.M1.Pharmacy.type));
                new(Case.has(R1,Case_,Val1_));
                new(Case.has(R2,Case_,Val2_));
                new(Case.has(R3,Case_,Val3_));
                setValue(Val1_,"false");
                setValue(Val2_,"false");
                setValue(Val3_,"delivery");

                new(Case.priority(Priority_) in Case_);
                new(Case.attr2(R7,Case_,Priority_));
                setValue(Priority_,"1");


                new(instanceOf(NewStateInstance_,State_));
                new (Case.currentState(R2_,Case_,NewStateInstance_));

                new(Case.groupNo(Store_) in Case_);
```

```
new(Case.group(R10,Case_,Store_));
setValue(Store_,"1");

//New case contains a prescription
new(Artifact(NewArtifact_)in Case_);
new(instanceOf(NewArtifact_,Prescription_));
new (Case.contains(R1_,Case_,NewArtifact_));

//Add a person
new(Person(Person_)in DSM.model.M1);
new(instanceOf(Person_,Patient_));
rename(Person_,name(Person_)+"Person");
new (Person.free(Free_)in Person_);
setValue(Free_,"false");
new(Person.attr(R6,Person_,Free_));

//Add customer roleinstance
new(RoleInstance(RoleInstance_)in DSM.model.M1);
new(instanceOf(RoleInstance_,Customer_));

//add roleInstance to Case_
new(RoleInstance.presence(R4,RoleInstance_,Case_));
//assign Person_ to RoleInstance_
new (RoleInstance.assignedTo(R5,RoleInstance_, Person_));


//Starttime defined
new( Case.startTime(StartTime_) in Case_);
new(Case.attr3(R8,Case_,StartTime_));
new(Integer(IntegerDay_) in StartTime_);
setValue(IntegerDay_,value(Day_) );
new(DateTime.day(R2_,StartTime_,IntegerDay_));
new(Integer(IntegerMonth_) in StartTime_);
setValue(IntegerMonth_,value(Month_));
new(DateTime.month(R3_,StartTime_,IntegerMonth_));
new(Integer(IntegerYear_) in StartTime_);
setValue(IntegerYear_,value(Year_));
new(DateTime.year(R4_,StartTime_,IntegerYear_));
new(Integer(IntegerHour_) in StartTime_);
setValue(IntegerHour_,value(Hour_));
new(DateTime.hour(R5_,StartTime_,IntegerHour_));
new(Integer(IntegerMin_)in StartTime_);
setValue(IntegerMin_,value(Minute_));
new(DateTime.minute(R6_,StartTime_,IntegerMin_));

if(toInteger(value(IntegerMin_)) >=10 ) seq{
    setValue (StartTime_, value(IntegerHour_)+":"+value(IntegerMin_)+" "+
        value(IntegerDay_)+"/"+ value(IntegerMonth_)+"/"+ value(IntegerYear_));
}
else seq{
    setValue (StartTime_, value(IntegerHour_)+":0"+value(IntegerMin_)+" "
    + value(IntegerDay_)+"/"+ value(IntegerMonth_)+"/"+ value(IntegerYear_));
}
//deadline defined
new(Case.deadline(Deadline_) in Case_);
new (Case.attr1(R2R, Case_,Deadline_));
new(Integer(IntegerDay2_) in Deadline_);
setValue(IntegerDay2_,toInteger(value(IntegerDay_)));
new(DateTime.day(R2_2,Deadline_,IntegerDay2_));
new(Integer(IntegerMonth2_) in Deadline_);
setValue(IntegerMonth2_,toInteger(value(IntegerMonth_)));
new(DateTime.month(R3_2,Deadline_,IntegerMonth2_));
```

```
                new(Integer(IntegerYear2_) in Deadline_);
                setValue(IntegerYear2_,toInteger(value(IntegerYear_)));
                new(DateTime.year(R4_2,Deadline_,IntegerYear2_));
                new(Integer(IntegerHour2_)in Deadline_);
                setValue(IntegerHour2_,toInteger(value(IntegerHour_)));
                new(Integer(IntegerMin2_) in Deadline_);

                if((toInteger(value(IntegerMin_))+15) < 60) seq{
                    setValue(IntegerMin2_,toInteger(value(IntegerMin_))+15);
                    setValue(IntegerHour2_,toInteger(value(IntegerHour_)));
                }
                else if (toInteger(value(IntegerHour_))+1 < 24 ||
                    ((toInteger(value(IntegerHour_))+1==24 &&
                    ((toInteger(value(IntegerMin_))+15)-60)==0 ))) seq{
                        setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
                        setValue(IntegerHour2_,toInteger(value(IntegerHour_))+1);
                }//assumption that all months have 31 days
                else if((toInteger(value(IntegerDay_))+1)<31) seq {
                    setValue(IntegerDay2_,toInteger(value(IntegerDay_))+1);
                    setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
                    setValue(IntegerHour2_,1);
                }
                else if((toInteger(value(IntegerMonth_))+1)<13) seq{
                    setValue(IntegerMonth2_,toInteger(value(IntegerMonth_))+1);
                    setValue(IntegerDay2_,1);
                    setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
                    setValue(IntegerHour2_,1);
                }
                else seq{
                    setValue(IntegerYear2_,toInteger(value(IntegerYear_))+1);
                    setValue(IntegerMonth2_,1);
                    setValue(IntegerDay2_,1);
                    setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
                    setValue(IntegerHour2_,1);
                }
                new(DateTime.hour(R5_2,Deadline_,IntegerHour2_));
                new(DateTime.minute(R6_2,Deadline_,IntegerMin2_));

                if(toInteger(value(IntegerMin2_)) >=10 ) seq{
                    setValue (Deadline_,   value(IntegerHour2_)+":"+value(IntegerMin2_)+" "
                        +value(IntegerDay2_)+"/"+ value(IntegerMonth2_)+"/"+ value(IntegerYear2_));
                }
                else seq{
                    setValue (Deadline_,   value(IntegerHour2_)+":0"+value(IntegerMin2_)+" "
                        +value(IntegerDay2_)+"/"+ value(IntegerMonth2_)+"/"+ value(IntegerYear2_));
                }
                    println("Case added ");
            }
        }
}
```

New Walk-in High Priority Case

```
gtrule Rule_NewCaseHighPriority() =
{
    precondition pattern lhs(Process_, Prescription_,Patient_,Customer_,Time_,
         Day_,Month_,Year_,Hour_,Minute_, M2_, State_) =
    {
```

```
    Process(Process_);
    ArtifactType(Prescription_);
    check(name(Prescription_)=="Prescription");

    Actor(Patient_);
    check(name(Patient_)=="Patient");

    Role(Customer_);
    check(name(Customer_)=="Customer");

    Clock(Clock_);
    Clock.Time(Time_);
    Clock.attr(R1,Clock_,Time_);
    find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
    M2(M2_);
    State(State_);
    check(name(State_)=="Type");
}
action {
    let
    Case_=undef,
    Val1_=undef,
    Val2_=undef,
    Val3_=undef,
    R1=undef,
    R2=undef,
    R3=undef,
    R4=undef,
    R5=undef,
    R6=undef,
    R7=undef,
    R8=undef,
    NewArtifact_=undef,
    R1_=undef,
    Person_=undef,
    RoleInstance_=undef,
    Priority_=undef,
    Free_=undef,
    StartTime_=undef,
    R2_=undef,
    IntegerDay_= undef,
    R3_=undef,
    IntegerMonth_= undef,
    R4_=undef,
    IntegerYear_= undef,
    R5_=undef,
    IntegerHour_= undef,
    IntegerMin_= undef,
    R6_=undef,
    Deadline_=undef,
    R2R=undef,
    R2_2=undef,
    IntegerDay2_= undef,
    R3_2=undef,
    IntegerMonth2_= undef,
    R4_2=undef,
    IntegerYear2_= undef,
    R5_2=undef,
    IntegerHour2_= undef,
    IntegerMin2_= undef,
    R6_2=undef,
    State_=undef,
    R9=undef,
    Store_=undef,
    R10=undef,
    NewStateInstance_=undef

in seq {
    new(Case(Case_)in M2_);
    rename(Case_,name(Case_)+"Case");
    //create instanceOf relation with the Case:Process
    new(instanceOf(Case_,Process_));
    new(AttributeValue(Val1_)in Case_);
    new(AttributeValue(Val2_)in Case_);
    new(AttributeValue(Val3_)in Case_);
    //create instanceOf relation with the AttributeValue:Process
    new(instanceOf(Val1_,DSM.model.M1.Pharmacy.checked));
    //create instanceOf relation with the AttributeValue:Process
    new(instanceOf(Val2_,DSM.model.M1.Pharmacy.counsel));
    //create instanceOf relation with the AttributeValue:Process
    new(instanceOf(Val3_,DSM.model.M1.Pharmacy.type));
    new(Case.has(R1,Case_,Val1_));
    new(Case.has(R2,Case_,Val2_));
    new(Case.has(R3,Case_,Val3_));
    setValue(Val1_,"false");
    setValue(Val2_,"false");
    setValue(Val3_,"walkin");
    new(Case.priority(Priority_) in Case_);
    new(Case.attr2(R7,Case_,Priority_));
    setValue(Priority_,"3");

    new(instanceOf(NewStateInstance_,State_));
    new (Case.currentState(R2_,Case_,NewStateInstance_));

    new(Case.groupNo(Store_) in Case_);
    new(Case.group(R10,Case_,Store_));
    setValue(Store_,"2");

    //New case contains a prescription
    new(Artifact(NewArtifact_)in Case_);
    new(instanceOf(NewArtifact_,Prescription_));
    new (Case.contains(R1_,Case_,NewArtifact_));

    //Add a person
    new(Person(Person_)in DSM.model.M1);
    new(instanceOf(Person_,Patient_));
    rename(Person_,name(Person_)+"Person");
    new (Person.free(Free_)in Person_);
    setValue(Free_,"false");
    new(Person.attr(R6,Person_,Free_));

    //Add customer roleinstance
    new(RoleInstance(RoleInstance_)in DSM.model.M1);
    new(instanceOf(RoleInstance_,Customer_));

    //add roleInstance to Case_
    new(RoleInstance.presence(R4,RoleInstance_,Case_));
    //assign Person_ to RoleInstance_
    new (RoleInstance.assignedTo(R5,RoleInstance_, Person_));


    //Starttime defined
    new( Case.startTime(StartTime_) in Case_);
    new(Case.attr3(R8,Case_,StartTime_));
```

```
new(Integer(IntegerDay_) in StartTime_);
setValue(IntegerDay_,value(Day_) );
new(DateTime.day(R2_,StartTime_,IntegerDay_));
new(Integer(IntegerMonth_) in StartTime_);
setValue(IntegerMonth_,value(Month_));
new(DateTime.month(R3_,StartTime_,IntegerMonth_));
new(Integer(IntegerYear_) in StartTime_);
setValue(IntegerYear_,value(Year_));
new(DateTime.year(R4_,StartTime_,IntegerYear_));
new(Integer(IntegerHour_) in StartTime_);
setValue(IntegerHour_,value(Hour_));
new(DateTime.hour(R5_,StartTime_,IntegerHour_));
new(Integer(IntegerMin_)in StartTime_);
setValue(IntegerMin_,value(Minute_));
new(DateTime.minute(R6_,StartTime_,IntegerMin_));

if(toInteger(value(IntegerMin_)) >=10 ) seq{
    setValue (StartTime_, value(IntegerHour_)+":"+value(IntegerMin_)+" "+
        value(IntegerDay_)+"/"+ value(IntegerMonth_)+"/"+ value(IntegerYear_));
}
else seq{
    setValue (StartTime_, value(IntegerHour_)+":0"+value(IntegerMin_)+" "+
        value(IntegerDay_)+"/"+ value(IntegerMonth_)+"/"+ value(IntegerYear_));
}

//deadline defined
new(Case.deadline(Deadline_) in Case_);
new (Case.attr1(R2R, Case_,Deadline_));
new(Integer(IntegerDay2_) in Deadline_);
setValue(IntegerDay2_,toInteger(value(IntegerDay_)));
new(DateTime.day(R2_2,Deadline_,IntegerDay2_));
new(Integer(IntegerMonth2_) in Deadline_);
setValue(IntegerMonth2_,toInteger(value(IntegerMonth_)));
new(DateTime.month(R3_2,Deadline_,IntegerMonth2_));
new(Integer(IntegerYear2_) in Deadline_);
setValue(IntegerYear2_,toInteger(value(IntegerYear_)));
new(DateTime.year(R4_2,Deadline_,IntegerYear2_));
new(Integer(IntegerHour2_)in Deadline_);
setValue(IntegerHour2_,toInteger(value(IntegerHour_)));
new(Integer(IntegerMin2_) in Deadline_);
if((toInteger(value(IntegerMin_))+15) < 60) seq{
    setValue(IntegerMin2_,toInteger(value(IntegerMin_))+15);
    setValue(IntegerHour2_,toInteger(value(IntegerHour_)));
}
else if (toInteger(value(IntegerHour_))+1 < 24 ||
    ((toInteger(value(IntegerHour_))+1==24 &&
        ((toInteger(value(IntegerMin_))+15)-60)==0 ))) seq{
            setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
            setValue(IntegerHour2_,toInteger(value(IntegerHour_))+1);

}//assumption that all months have 31 days
else if((toInteger(value(IntegerDay_))+1)<31) seq {
    setValue(IntegerDay2_,toInteger(value(IntegerDay_))+1);
    setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
    setValue(IntegerHour2_,1);
}
else if((toInteger(value(IntegerMonth_))+1)<13) seq{
    setValue(IntegerMonth2_,toInteger(value(IntegerMonth_))+1);
    setValue(IntegerDay2_,1);
    setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
    setValue(IntegerHour2_,1);
}
```

```
        else seq{
            setValue(IntegerYear2_,toInteger(value(IntegerYear_))+1);
            setValue(IntegerMonth2_,1);
            setValue(IntegerDay2_,1);
            setValue(IntegerMin2_,((toInteger(value(IntegerMin_))+15)-60));
            setValue(IntegerHour2_,1);
        }
        new(DateTime.hour(R5_2,Deadline_,IntegerHour2_));
        new(DateTime.minute(R6_2,Deadline_,IntegerMin2_));

        if(toInteger(value(IntegerMin2_)) >=10 ) seq{
            setValue (Deadline_,   value(IntegerHour2_)+":"+value(IntegerMin2_)+" "
                +value(IntegerDay2_)+"/"+ value(IntegerMonth2_)+"/"+ value(IntegerYear2_));
        }
        else seq{
            setValue (Deadline_,   value(IntegerHour2_)+":0"+value(IntegerMin2_)+" "
                +value(IntegerDay2_)+"/"+ value(IntegerMonth2_)+"/"+ value(IntegerYear2_));
        }
        println("Case added ");
        }
    }
}
```

A.3.2 Type Prescription Rule

```
gtrule Rule_TypePrescription() =
{
    precondition pattern lhs(Case_,RoleInstance_,Role_,Person_,
        TypedPrescription_, StateInstance_, NextState_,R6_) =
    {
        Case(Case_);
        Role(Role_);
        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);
        RoleInstance(RoleInstance_);
        ArtifactType(TypedPrescription_);

        Case.currentState(R6_,Case_,StateInstance_);
        typeOf(State_,StateInstance_);
        check(name(State_)=="Type");

        State.next(R1,State_,NextState_);

        check(name(TypedPrescription_)=="TypedPrescription");
        find ETassigned (Case_,RoleInstance_,Role_,Person_);
        neg find TypedPrescriptionExist(Case_);
        find PrescriptionExist(Case_);
    }
    action {
        let
            NewArtifact_=undef,
            NewStateInstance_=undef,
            R1_=undef,
            R2_=undef

        in seq {
            // create Class inside the Case
            new(Artifact(NewArtifact_)in Case_);
            //create instanceOf relation with the Artifact:ArtifactType
            new(instanceOf(NewArtifact_,TypedPrescription_));
            //create relation between Case and Artifact
            new (Case.contains(R1_,Case_,NewArtifact_));

            delete(StateInstance_);
            delete(R6_);
            // create Class inside the Case
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            //create instanceOf relation with the Artifact:ArtifactType
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
            println("Typed Prescription added ");
        }
        println ("Type Prescription for "+name(Case_)+" Case");
    }
}
```

A.3.3 Print Label Rule

```
gtrule Rule_PrintPrescription() =
{
    precondition pattern lhs(Case_, Label_, StateInstance_, NextState_,R6_) =
    {
        ArtifactType(Label_);
        check(name(Label_)=="Label");
        Case(Case_);
        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);

        typeOf(State_,StateInstance_);
        check(name(State_)=="Print");

        find TypedPrescriptionExist(Case_);
        neg find LabelExist(Case_);

    }
    action {

    let
        NewArtifact_=undef,
        R1_=undef,
        NewStateInstance_=undef,
        R2_=undef
    in seq {
            // create Class inside the Case
            new(Artifact(NewArtifact_)in Case_);
            //create instanceOf relation with the Artifact:ArtifactType
            new(instanceOf(NewArtifact_,Label_));
            //create relation between Case and Artifact
            new (Case.contains(R1_,Case_,NewArtifact_));

            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
            println("Label printed");
        }
        println ("Print Prescription Label for "+name(Case_)+" Case");
    }
}
```

A.3.4 Fill Prescription Rule

```
gtrule Rule_FillPrescription() =
{

    precondition pattern lhs(Case_,RoleInstance_,Role_,Person_,
        FilledPrescription_, StateInstance_, NextState_,R6_) =
    {
        Case(Case_);
        ArtifactType(FilledPrescription_);
        check(name(FilledPrescription_)=="FilledPrescription");
        find FTassigned (Case_,RoleInstance_,Role_,Person_);
        neg find FilledPrescriptionExist(Case_);
        find LabelExist(Case_);

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);
    }
    action {
        let
            NewArtifact_=undef,
            R1_=undef,
            NewStateInstance_=undef,
            R2_=undef

        in seq {
            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));

            new(Artifact(NewArtifact_)in Case_);
            new(instanceOf(NewArtifact_,FilledPrescription_));
            new (Case.contains(R1_,Case_,NewArtifact_));

            println("Filled Prescription added ");
        }
        println ("Fill Prescription for "+name(Case_)+" Case");
    }
}
```

A.3.5 Check Rules

Unsuccessful Check Type 1

```
gtrule Rule_UnsucessfullCheck() =
{
    precondition pattern lhs(Case_, Artifact_, StateInstance_, NextState_,R6_) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);
        find RequiresChecked(Case_,AttributeValue_);
        find DPassigned (Case_,RoleInstance_,Role_,Person_);

        find FilledPrescriptionExist(Case_,Artifact_,ArtifactType_);

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);
    }
    action {
        //removes the filledprescription
        let
            NewStateInstance_=undef,
            R2_=undef
        in seq{
            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
            delete(Artifact_);
            println("Prescription was unsuccessfully checked");
            println(name(Case_)+" Case requires prescription to be refilled to be redone");
        }
    }
}
```

Unsuccessful Check Type 2

```
gtrule Rule_UnsucessfullCheck2() =
    {
            precondition pattern lhs(Case_, Artifact_, Artifact2_,
                 Artifact3_, StateInstance_, NextState_,R6_) =
            {
           Case(Case_);
             AttributeValue(AttributeValue_);
             find RequiresChecked(Case_,AttributeValue_);
           find DPassigned (Case_,RoleInstance_,Role_,Person_);

             find FilledPrescriptionExist(Case_,Artifact_,ArtifactType_);
             find TypedPrescriptionExist(Case_,Artifact2_,ArtifactType2_);
             find LabelExist(Case_,Artifact3_,ArtifactType3_);

             State(State_);
            State(NextState_);
            StateInstance(StateInstance_);

            Case.currentState(R6_,Case_,StateInstance_);
            State.next(R7,State_,NextState_);
            }
            action {
                let
                    NewStateInstance_=undef,
                    R2_=undef
                in seq{
                    delete(StateInstance_);
                    delete(R6_);
                    new(StateInstance(NewStateInstance_)in DSM.model.M1);
                    new(instanceOf(NewStateInstance_,NextState_));
                    new (Case.currentState(R2_,Case_,NewStateInstance_));

                    //removes the filledprescription, typedprescription and label
                    delete(Artifact_);
                    delete(Artifact2_);
                    delete(Artifact3_);
                    println("Prescription was unsuccessfully checked (back to type state)");
                    println(name(Case_)+" Case requires prescription to be completely redone");
                }
            }
     }
```

Successful Check Default

```
gtrule Rule_SucessfullCheck() =
{
    precondition pattern lhs(Case_,AttributeValue_,Bag_,Artifact_,
        Artifact2_, StateInstance_, NextState_,R6_ ) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);
        ArtifactType(Bag_);
        check(name(Bag_)=="Bag");
        find RequiresChecked(Case_,AttributeValue_);
        find DPassigned (Case_,RoleInstance_,Role_,Person_);
        find FilledPrescriptionExist(Case_,Artifact_,ArtifactType_);
        find LabelExist (Case_,Artifact2_, ArtifactType2_);

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);
    }
    action {
        let
            NewArtifact_=undef,
            R1_=undef,
            NewStateInstance_=undef,
            R2_=undef

        in seq {
            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));

            new(Artifact(NewArtifact_)in Case_);
            new(instanceOf(NewArtifact_,Bag_));
            new (Artifact(Artifact_) in NewArtifact_);
            new (Artifact(Artifact2_) in NewArtifact_);
            new (Case.contains(R1_,Case_,NewArtifact_));

            setValue(AttributeValue_,"true");

            println("Prescription is checked");
            println(name(Case_)+" Case requires prescription to be checked");
        }
    }
}
```

Successful Check on Time Rule

```
gtrule Rule_SucessfullCheckOntime() =
{

    precondition pattern lhs(Case_,AttributeValue_,Artifact_,
        Artifact2_, StateInstance_, NextState_,R6_) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);
        find RequiresChecked(Case_,AttributeValue_);
        find DPassigned (Case_,RoleInstance_,Role_,Person_);

        find FilledPrescriptionExist(Case_,Artifact_,ArtifactType_);
        find LabelExist (Case_,Artifact2_, ArtifactType2_);

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);
        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);

        find getEndTime(Deadline_, Day_,Month_,Year_,Hour_,Minute_);
        find getTime(Time_, Day1_,Month1_,Year1_,Hour1_,Minute1_);

        check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_)))
            <=((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));

    }
    action {

     let
            NewStateInstance_=undef,
            R2_=undef
            in seq{
            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
            setValue(AttributeValue_,"true");
    }
        println("Prescription is checked");
        println(name(Case_)+" Case requires prescription to be checked");

    }
}
```

Successful Check Less Than 5 Minutes Late

```
gtrule Rule_SucessfullCheckLessthan5() =
{
    precondition pattern lhs(Case_,AttributeValue_,Artifact_,
        Artifact2_, StateInstance_, NextState_,R6_ ) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);

        find RequiresChecked(Case_,AttributeValue_);
        find DPassigned (Case_,RoleInstance_,Role_,Person_);

        find FilledPrescriptionExist(Case_,Artifact_,ArtifactType_);
        find LabelExist (Case_,Artifact2_, ArtifactType2_);

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);

        find getEndTime(Deadline_, Day_,Month_,Year_,Hour_,Minute_);
        find getTime(Time_, Day1_,Month1_,Year1_,Hour1_,Minute1_);

        check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_))) >
            ((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));
        check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_))-5) <=
            ((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));

    }
    action {

        let
            NewStateInstance_=undef,
            R2_=undef
            in seq{
            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
    }

    setValue(AttributeValue_,"true");
    println("Prescription is checked");
    println(name(Case_)+" Case requires prescription to be checked");
    }
}
```

Successful Check More Than 5 Minutes Late

```
gtrule Rule_SucessfullCheckMorethan5() =
{
    precondition pattern lhs(Case_,AttributeValue_,Artifact_,Artifact2_,
         StateInstance_, NextState_,R6_ ) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);

        find RequiresChecked(Case_,AttributeValue_);
        find DPassigned (Case_,RoleInstance_,Role_,Person_);

        find FilledPrescriptionExist(Case_,Artifact_,ArtifactType_);
        find LabelExist (Case_,Artifact2_, ArtifactType2_);

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);

        find getEndTime(Deadline_, Day_,Month_,Year_,Hour_,Minute_);
        find getTime(Time_, Day1_,Month1_,Year1_,Hour1_,Minute1_);

        check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_))) >
            ((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));
        check ((60*toInteger(value(Hour1_)) + toInteger(value(Minute1_))-5) >
            ((60*toInteger(value(Hour_))) +toInteger(value(Minute_))));

    }
    action {

        let
```

```
            NewStateInstance_=undef,
            R2_=undef
            in seq{

            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
            setValue(AttributeValue_,"true");
    }
    println("Prescription is checked");
    println(name(Case_)+" Case requires prescription to be checked");
    }
}
```

A.3.6 Counsel Rules

Default Counsel Rule

```
gtrule Rule_Counsel() =
{
    precondition pattern lhs(Case_,AttributeValue_,
    StateInstance_, NextState_,R6_) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);
        AttributeValue(AttributeValue2_);
        find RequiresCounsel(Case_, AttributeValue_);
        neg find RequiresChecked(Case_,AttributeValue2_);

        Case.has(R4,Case_,AttributeValue_);

        find DPassigned (Case_,RoleInstance_,Role_,Person_);
        find CAssigned (Case_,RoleInstance2_,Role2_,Person2_);

        Clock(Clock_);
        Clock.Time(Time_);
        Case.deadline(Deadline_);
        Clock.attr(R1,Clock_,Time_);
        find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
        Case.attr1(R2, Case_,Deadline_);
        find getEndTime(Deadline_, Day2_,Month2_,Year2_,Hour2_,Minute2_);

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);
    }
    action {
        let
            NewStateInstance_=undef,
            R2_=undef
        in seq {

            delete(StateInstance_);
            delete(R6_);
            // create Class inside the Case
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            //create instanceOf relation with the Artifact:ArtifactType
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
            setValue(AttributeValue_,"true");

            println("Counselling done ");

        }
        println(name(Case_)+" Case requires counselling");
    }

}
```

Counsel Late

```
gtrule Rule_CounselLate() =
{
    precondition pattern lhs(Case_,AttributeValue_,
        StateInstance_, NextState_,R6_) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);
        AttributeValue(AttributeValue2_);
        find RequiresCounsel(Case_, AttributeValue_);
        neg find RequiresChecked(Case_,AttributeValue2_);
        Case.has(R4,Case_,AttributeValue_);
        find DPassigned (Case_,RoleInstance_,Role_,Person_);
        find CAssigned (Case_,RoleInstance2_,Role2_,Person2_);

        Clock(Clock_);
        Clock.Time(Time_);
        Case.deadline(Deadline_);
        Clock.attr(R1,Clock_,Time_);
        find getTime(Time_, Day_,Month_,Year_,Hour_,Minute_);
        Case.attr1(R2, Case_,Deadline_);
        find getEndTime(Deadline_, Day2_,Month2_,Year2_,Hour2_,Minute2_);

        check ((60*toInteger(value(Hour_)) + toInteger(value(Minute_)))
          < (60*toInteger(value(Hour2_)) +toInteger(value(Minute2_))));

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);
    }
    action {
        let
            NewStateInstance_=undef,
            R2_=undef
        in seq {
            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
            setValue(AttributeValue_,"true");
            println("Counselling done ");
        }
        println(name(Case_)+" Case requires counselling");
    }

}
```

Counsel High Priority Case

```
gtrule Rule_CounselHigh() =
{
    precondition pattern lhs(Case_,AttributeValue_,
        StateInstance_, NextState_,R6_) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);
        AttributeValue(AttributeValue2_);
        find RequiresCounsel(Case_, AttributeValue_);
        neg find RequiresChecked(Case_,AttributeValue2_);
        Case.has(R4,Case_,AttributeValue_);
        find DPassigned (Case_,RoleInstance_,Role_,Person_);
        find CAssigned (Case_,RoleInstance2_,Role2_,Person2_);


        Case.priority(Priority_);
        Case.attr2(R1,Case_,Priority_);

        check (value(Priority_)=="3");

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);
    }
    action {
        let
            NewStateInstance_=undef,
            R2_=undef
        in seq {
            delete(StateInstance_);
            delete(R6_);

            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            //create instanceOf relation with the Artifact:ArtifactType
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
            setValue(AttributeValue_,"true");

            println("Counselling done ");
        }
        println(name(Case_)+" Case requires counselling");
    }

}
```

Counsel Medium Priority Case

```
gtrule Rule_CounselMed() =
{
    precondition pattern lhs(Case_,AttributeValue_,
        StateInstance_, NextState_,R6_) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);
        Case.has(R4,Case_,AttributeValue_);
        AttributeValue(AttributeValue2_);
        find RequiresCounsel(Case_, AttributeValue_);
        neg find RequiresChecked(Case_,AttributeValue2_);
        find DPassigned (Case_,RoleInstance_,Role_,Person_);
        find CAssigned (Case_,RoleInstance2_,Role2_,Person2_);


        Case.priority(Priority_);
        Case.attr2(R1,Case_,Priority_);

        check (value(Priority_)=="1");

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);
    }
    action {
        let
            NewStateInstance_=undef,
            R2_=undef
        in seq {

            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));

            setValue(AttributeValue_,"true");

        }
        println("Counselling done ");

        println(name(Case_)+" Case requires counselling");
    }

}
```

Counsel Low Priority Case

```
gtrule Rule_CounselLow() =
{
    precondition pattern lhs(Case_,AttributeValue_, StateInstance_, NextState_,R6_) =
    {
        Case(Case_);
        AttributeValue(AttributeValue_);
        AttributeValue(AttributeValue2_);
        find RequiresCounsel(Case_, AttributeValue_);
        neg find RequiresChecked(Case_,AttributeValue2_);
        Case.has(R4,Case_,AttributeValue_);
        find DPassigned (Case_,RoleInstance_,Role_,Person_);
        find CAssigned (Case_,RoleInstance2_,Role2_,Person2_);
        Case.priority(Priority_);
        Case.attr2(R1,Case_,Priority_);
        check (value(Priority_)=="-1");
        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);
        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);
    }
    action {
    let
        NewStateInstance_=undef,
        R2_=undef
        in seq {
            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
             new (Case.currentState(R2_,Case_,NewStateInstance_));

            setValue(AttributeValue_,"true");
            println("Counselling done ");
            println(name(Case_)+" Case requires counselling");
        }

    }
}
```

A.3.7 Give Payment Rule

```
gtrule Rule_GivePayment() =
{

    precondition pattern lhs(Case_,Payment_, StateInstance_, NextState_,R6_) =
    {
        Case(Case_);

        ArtifactType(Payment_);
        check(name(Payment_)=="Payment");

        find PCAssigned (Case_,RoleInstance_,Role_,Person_);
        find CAssigned (Case_,RoleInstance2_,Role2_,Person2_);
        neg find PaymentExist (Case_) ;

        State(State_);
        State(NextState_);
        StateInstance(StateInstance_);

        Case.currentState(R6_,Case_,StateInstance_);
        State.next(R7,State_,NextState_);
    }
    action {
        let
            NewArtifact_=undef,
            R1_=undef,
            NewStateInstance_=undef,
            R2_=undef

        in seq {
            delete(StateInstance_);
            delete(R6_);
            new(StateInstance(NewStateInstance_)in DSM.model.M1);
            new(instanceOf(NewStateInstance_,NextState_));
            new (Case.currentState(R2_,Case_,NewStateInstance_));
            new(Artifact(NewArtifact_)in Case_);
            new(instanceOf(NewArtifact_,Payment_));
            new (Case.contains(R1_,Case_,NewArtifact_));

            println("Payment received");
            println(name(Case_)+" Case received payment from customer");
        }
    }
}
```

A.4 Patterns

A.4.1 “Exist” Patterns

```
pattern Trigger1Exist (Case_)={

    Case(Case_);

    Escalation(Level_);
    Case.escalations(R1,Case_, Level_);
    check (name(Level_)=="level1");

}

pattern Trigger2Exist (Case_)={

Case(Case_);
```

```
Escalation(Level_);
Case.escalations(R1,Case_, Level_);
check (name(Level_)=="level2");

}
pattern Trigger3Exist (Case_)={

Case(Case_);

Escalation(Level_);
Case.escalations(R1,Case_, Level_);
check (name(Level_)=="level3");

}
//Case contains an instance of Payment:ArtifactType
pattern PaymentExist (Case_)={
Artifact(Artifact_);
ArtifactType(ArtifactType_);
Case(Case_);
Case.contains(Rel, Case_, Artifact_);
typeOf(ArtifactType_,Artifact_);
check (name(ArtifactType_)=="Payment");
}



//Case contains an instance of Label:ArtifactType
pattern LabelExist (Case_)={
Artifact(Artifact_);
ArtifactType(ArtifactType_);
Case(Case_);
Case.contains(Rel, Case_, Artifact_);
typeOf(ArtifactType_,Artifact_);
check (name(ArtifactType_)=="Label");
}

//Case contains an instance of Label:ArtifactType
pattern LabelExist (Case_,Artifact_, ArtifactType_)={
    Artifact(Artifact_);
    ArtifactType(ArtifactType_);
    Case(Case_);
    Case.contains(Rel, Case_, Artifact_);
    typeOf(ArtifactType_,Artifact_);
    check (name(ArtifactType_)=="Label");
}
//Case contains an instance of TypedPrescription:ArtifactType
pattern TypedPrescriptionExist(Case_)={
    Artifact(Artifact_);
    ArtifactType(ArtifactType_);
    Case(Case_);
    Case.contains(Rel, Case_, Artifact_);
    typeOf(ArtifactType_,Artifact_);
    check (name(ArtifactType_)=="TypedPrescription");
}

//Case contains an instance of TypedPrescription:ArtifactType
pattern TypedPrescriptionExist(Case_,Artifact_, ArtifactType_)={
    Artifact(Artifact_);
    ArtifactType(ArtifactType_);
    Case(Case_);
    Case.contains(Rel, Case_, Artifact_);
    typeOf(ArtifactType_,Artifact_);
    check (name(ArtifactType_)=="TypedPrescription");
}


//Case contains an instance of Prescription:ArtifactType
pattern PrescriptionExist(Case_)={
    Artifact(Artifact_);
    ArtifactType(ArtifactType_);
    Case(Case_);
    Case.contains(Rel, Case_, Artifact_);
    typeOf(ArtifactType_,Artifact_);
    check (name(ArtifactType_)=="Prescription");
}


//Case contains an instance of FilledPrescription:ArtifactType
pattern FilledPrescriptionExist(Case_)={
    Artifact(Artifact_);
    ArtifactType(ArtifactType_);
    Case(Case_);
    Case.contains(Rel, Case_, Artifact_);
    typeOf(ArtifactType_,Artifact_);
    check (name(ArtifactType_)=="FilledPrescription");
}
//Case contains an instance of FilledPrescription:ArtifactType
pattern FilledPrescriptionExist(Case_,Artifact_, ArtifactType_)={
    Artifact(Artifact_);
    ArtifactType(ArtifactType_);
    Case(Case_);
    Case.contains(Rel, Case_, Artifact_);
    typeOf(ArtifactType_,Artifact_);
    check (name(ArtifactType_)=="FilledPrescription");
}
```

A.4.2 “Assigned” Patterns

```
pattern isAssign(RoleInstance_)={
    RoleInstance(RoleInstance_);
    Person(Person_);
    Case(Case_);
    RoleInstance.presence(R2,RoleInstance_,Case_);
    RoleInstance.assignedTo(R1, RoleInstance_,Person_);
}
//DispensingPharmacist Assigned
pattern DPassigned (Case_,RoleInstance_,Role_,Person_) =
{
    Person(Person_);
    Case(Case_);
    Role(Role_);
    check (name(Role_)=="DispensingPharmacist");
    RoleInstance(RoleInstance_);
    typeOf(Role_, RoleInstance_);
    RoleInstance.presence(Rel2, RoleInstance_, Case_);
    RoleInstance.assignedTo(Rel,RoleInstance_,Person_);
}
```

```
//EntryTechnician Assigned
pattern ETassigned (Case_,RoleInstance_,Role_,Person_) =
{
    Person(Person_);
    Case(Case_);
    Role(Role_);
    check (name(Role_)=="EntryTechnician");
    RoleInstance(RoleInstance_);
    typeOf(Role_, RoleInstance_);
    RoleInstance.presence(Rel2, RoleInstance_, Case_);
    RoleInstance.assignedTo(Rel,RoleInstance_,Person_);
}

//FillingTechnician Assigned
pattern FTassigned (Case_,RoleInstance_,Role_,Person_) =
{
    Person(Person_);
    Case(Case_);
    Role(Role_);
    check (name(Role_)=="FillingTechnician");
    RoleInstance(RoleInstance_);
    typeOf(Role_, RoleInstance_);
    RoleInstance.presence(Rel2, RoleInstance_, Case_);
    RoleInstance.assignedTo(Rel,RoleInstance_,Person_);
}
//Customer Assigned
pattern CAssigned (Case_,RoleInstance_,Role_,Person_) =
{
    Person(Person_);
    Case(Case_);
    Role(Role_);
    check (name(Role_)=="Customer");
    RoleInstance(RoleInstance_);
    typeOf(Role_, RoleInstance_);
    RoleInstance.presence(Rel2, RoleInstance_, Case_);
    RoleInstance.assignedTo(Rel,RoleInstance_,Person_);
}

//Customer Assigned
pattern CAssigned (Case_) =
{
Person(Person_);
Case(Case_);
Role(Role_);
check (name(Role_)=="Customer");
RoleInstance(RoleInstance_);
typeOf(Role_, RoleInstance_);
RoleInstance.presence(Rel2, RoleInstance_, Case_);
RoleInstance.assignedTo(Rel,RoleInstance_,Person_);
}
//PharmacyCashier Assigned
pattern PCAssigned (Case_,RoleInstance_,Role_,Person_) =
{
    Person(Person_);
    Case(Case_);
    Role(Role_);
    check (name(Role_)=="PharmacyCashier");
    RoleInstance(RoleInstance_);
    typeOf(Role_, RoleInstance_);
    RoleInstance.presence(Rel2, RoleInstance_, Case_);
    RoleInstance.assignedTo(Rel,RoleInstance_,Person_);
}
```

```
pattern checkIfAssigned2 (Case_,RoleInstance_,Role_,Person_) =
{
    Person(Person_);
    Case(Case_);
    Role(Role_);
    RoleInstance(RoleInstance_);
    typeOf(Role_, RoleInstance_);
    RoleInstance.presence(Rel2, RoleInstance_, Case_);
    RoleInstance.assignedTo(Rel,RoleInstance_,Person_);

}
```

A.4.3 “Free Person” Patterns

```
pattern freePerson(Person_, Free_)={
    Person(Person_);
    Person.free(Free_);
    Person.attr(P1,Person_,Free_);
    check (toBoolean(value(Free_)) == true);
}



pattern freePerson(Actor_,Person_, Free_, Case_)={
    Case(Case_);
    Person(Person_);
    Person.free(Free_);
    Person.attr(P1,Person_,Free_);
    Actor(Actor_);
    typeOf(Actor_, Person_);
    check (toBoolean(value(Free_)) == true);
    Person.groupNo(Store_);
    Person.group(R2_,Person_,Store_);
    Case.groupNo(Store2_);
    Case.group(R3_,Case_,Store2_);
    check (value(Store_)== value(Store2_));
}

pattern freePersonCapability(Actor_,Role_,Person_, Free_, Case_)={
    Case(Case_);
    Person(Person_);
    Person.free(Free_);
    Person.attr(P1,Person_,Free_);
    Actor(Actor_);
    Role(Role_);
    Capability(Capability_);
    Role.required(R1,Role_,Capability_);
    Person.actual(R2,Person_,Capability_);
    check (toBoolean(value(Free_)) == true);
    Person.groupNo(Store_);
    Person.group(R2_,Person_,Store_);
    Case.groupNo(Store2_);
    Case.group(R3_,Case_,Store2_);
    check (value(Store_)== value(Store2_));
}
```

```
or{
    find  freePerson(Actor_,Person_, Free_, Case_);
}


pattern freePersonPolicy(Actor_,Person_, Free_, Case_)={
    Case(Case_);
    Person(Person_);
    Person.free(Free_);
    Person.attr(P1,Person_,Free_);
    Actor(Actor_);

    typeOf(Actor_, Person_);
    check (toBoolean(value(Free_)) == true);
    Person.groupNo(Store_);
    Person.group(R2_,Person_,Store_);
    Case.groupNo(Store2_);
    Case.group(R3_,Case_,Store2_);
    check (value(Store_)== value(Store2_));
}
or
{
    Case(Case_);
    Actor(Actor2_);
    Actor(Actor_);
    Actor.super(R3_,Actor2_,Actor_);

    find  freePersonPolicy(Actor2_,Person2_, Free2_, Case_);
}

pattern freePersonCapabilityPolicy(Actor_,Role_,Person_, Free_, Case_)={
    Case(Case_);
    Person(Person_);
    Person.free(Free_);
    Person.attr(P1,Person_,Free_);
    Actor(Actor_);
    Role(Role_);
    Capability(Capability_);
    Role.required(R1,Role_,Capability_);
    Person.actual(R2,Person_,Capability_);
    check (toBoolean(value(Free_)) == true);
    Person.groupNo(Store_);
    Person.group(R2_,Person_,Store_);
    Case.groupNo(Store2_);
    Case.group(R3_,Case_,Store2_);
    check (value(Store_)== value(Store2_));
}
or{
    find  freePersonPolicy(Actor_,Person_, Free_, Case_);
}
```

A.4.4 “Requires” Patterns

```
pattern RequiresState(State_,Role_)={
    State(State_);
    Role(Role_);
    State.requires(R1_,State_,Role_);
}

pattern RequiresCounsel(Case_, AttributeValue_)={
    Case(Case_);
    AttributeValue(AttributeValue_);
    Case.has(R1,Case_,AttributeValue_);
    AttributeDeclaration(AttributeDeclaration_);
    typeOf(AttributeDeclaration_,AttributeValue_);
    check (name(AttributeDeclaration_) == "counsel");
    check(value(AttributeValue_)=="false");
}

//check if Case requires to be checked
pattern RequiresChecked(Case_, AttributeValue_)={
    Case(Case_);
    AttributeValue(AttributeValue_);
    Case.has(R1,Case_,AttributeValue_);
    AttributeDeclaration(AttributeDeclaration_);
    typeOf(AttributeDeclaration_,AttributeValue_);
    check (name(AttributeDeclaration_) == "checked");
    check(value(AttributeValue_)=="false");
}
```

A.4.5 RoleInstance Presence on Case

```
pattern RoleInstance(Case_, Role_)={
    Case(Case_);
    Role(Role_);
    RoleInstance(RoleInstance_);
    typeOf(Role_,RoleInstance_);
    RoleInstance.presence(R1,RoleInstance_,Case_);
}
```

A.4.6 Time Patterns

```
pattern getStartTime(Time_, Day_,Month_,Year_,Hour_,Minute_)={
    Case.startTime(Time_);
    Integer(Day_);
    Integer(Month_);
    Integer(Year_);
    Integer(Hour_);
    Integer(Minute_);

    DateTime.day(R2_,Time_,Day_);
    DateTime.month(R3_,Time_,Month_);
    DateTime.year(R4_,Time_,Year_);
    DateTime.hour(R5_,Time_,Hour_);
    DateTime.minute(R6_,Time_,Minute_);
}

pattern getTime(Time_, Day_,Month_,Year_,Hour_,Minute_)={
    Clock.Time(Time_);
    Integer(Day_);
    Integer(Month_);
    Integer(Year_);
    Integer(Hour_);
    Integer(Minute_);

    DateTime.day(R2_,Time_,Day_);
    DateTime.month(R3_,Time_,Month_);
    DateTime.year(R4_,Time_,Year_);
    DateTime.hour(R5_,Time_,Hour_);
    DateTime.minute(R6_,Time_,Minute_);
}

pattern getEndTime(Time_, Day_,Month_,Year_,Hour_,Minute_)={
    Case.deadline(Time_);
    Integer(Day_);
    Integer(Month_);
    Integer(Year_);
    Integer(Hour_);
    Integer(Minute_);

    DateTime.day(R2_,Time_,Day_);
    DateTime.month(R3_,Time_,Month_);
    DateTime.year(R4_,Time_,Year_);
    DateTime.hour(R5_,Time_,Hour_);
    DateTime.minute(R6_,Time_,Minute_);
}
```

Bibliography

[1] M. A. Adibi, M. Zandieh, and M. Amiri. Multi-objective scheduling of dynamic job shop using variable neighborhood search. *Expert System Application*, 37:282–287, January 2010.

[2] Adobe, BEA, Oracle, Active Endpoints, IBM, and SAP. Web service human task (WS-HumanTask). *URL: http://incubator.apache.org/hise/WS-HumanTask_v1.pdf*, (1.0), June 2007 (Accessed Oct 1 2008).

[3] Alessandra Agostini and Giorgio De Michelis. Improving flexibility of workflow management systems. In Wil van der Aalst, Jörg Desel, and Andreas Oberweis, editors, *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 289–342. Springer Berlin Heidelberg, 2000.

[4] Ariba, International Business Machines Corporation, and Microsoft. Web services description language (WSDL) 1.1. *URL: http://www.w3.org/TR/wsdl*, 2001 (Accessed December 1 2008).

[5] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, September 2003.

[6] Björn Axenath, Ekkart Kindler, and Vladimir Rubin. The aspects of business processes: An open and formalism independent ontology. Technical Report tr-ri-05-256, Computer Science Department, University of Paderborn, 2005.

[7] Moulinath Banerjee. A note on the exponential distribution. *URL: `http://www.stat.lsa.umich.edu/~moulib/note-exponential.pdf`*, 2007 (Accessed Feburary 27 2011).

[8] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation.* Prentice Hall, third edition, 2000.

[9] Jean Bezivin, Frederic Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.*, Vancouver, British Columbia, Canada, October 2004. ACM Inc.

[10] BOC-Group. ADONIS:Community Edition. *URL: `http://www.adonis-community.com/`*, 2010 (Accessed September 14 2010).

[11] Jeremy T. Bradley. Semi-markov PEPA: Modelling with generally distributed actions. *International Simulation Journal*, 6(3-4):43–51, 2005.

[12] Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Guiseppe Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Trans. Database Syst.*, 24(3):405–451, 1999.

[13] Jayeeta Chanda, Ananya Kanjilal, Sabnam Sengupta, and Swapan Bhattacharya. Traceability of requirements and consistency verification of UML usecase, activity and class diagram: A formal approach. In *International Conference on Methods and Models in Computer Science*, pages 1–4. IEEE, 2009.

[14] GAMS Development Corporation. General Algebraic Modeling System (GAMS). *URL: `http://www.gams.com/`*, 2010 (Accessed October 1 2010).

[15] Shoppers Drug Mart Corporation. Shoppers Drug Mart. *URL: `http://www.shoppersdrugmart.ca/`*, 2011 (Accessed April 8 2011).

[16] Peter Cowling and Marcus Johansson. Using real time information for effective dynamic scheduling. *European Journal of Operational Research*, 139(2):230–244, June 2002.

[17] Joel M. Crichlow. *An Introduction to Distributed and Parallel Computing*. Prentice Hall, 1997.

[18] Ralph Depke, Reiko Heckel, and Jochen Küster. Agent-oriented modeling with graph transformation. In *Agent-Oriented Software Engineering: First International Workshop, AOSE 2000 Limerick, Ireland, June 10, 2000 Revised Papers*, volume 1957 of *LNCS*, pages 105–119, Berlin, Heidelberg, 2001. Springer-Verlag.

[19] Ralph Depke, Reiko Heckel, and Jochen Malte Küster. Formal agent-oriented modeling with UML and graph transformation. In *Sci. Comput. Program.*, volume 44, pages 229–252, Amsterdam, The Netherlands, 2002. Elsevier North-Holland, Inc.

[20] Marlon Dumas, Wil M. van der Aalst, and Arthur H.M.ter Hofstede, editors. *Process-Aware Information Systems : Bridging People and Software through Process Technology.* Wiley-Interscience, Hoboken, NJ, 2005.

[21] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[22] Brian Sidney Everitt. *The Cambridge Dictionary of Statistics.* Cambridge Univ. Press, Cambridge [u.a.], third edition, 2006.

[23] David Ferraiolo and Richard Kuhn. Role-based access control. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[24] Martin Fowler. *UML Distilled.* Addison-Wesley, Boston, Massachusetts, thrid edition, 2004.

[25] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pages 21–35, London, UK, 1997. Springer-Verlag.

[26] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[27] Stephen Gilmore and Jane Hillston. The PEPA workbench: A tool to support a process algebra-based approach to performance modelling. In *In Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, number 794 in Lecture Notes in Computer Science*, pages 353–368. Springer-Verlag, 1994.

[28] Cesar Gonazalez-Perez and Brian Henderson-Sellers. *Metamodelling for Software Engineering.* John Wiley & Sons Ltd., 2008.

[29] LASER Process Working Group. Little-JIL 1.5 Language Report. Technical report, Laboratory for Advanced Software Engineering Research, University of Massachusetts, Amherst, 1997-2006.

[30] Reiko Heckel and Paolo Torrini. Stochastic modelling and simulation of mobile systems. In Gregor Engels, Claus Lewerentz, Wilhelm Schäfer, Andy Schürr, and Bernhard Westfechtel, editors, *Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science*, pages 87–101. Springer Berlin / Heidelberg, 2010.

[31] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph grammars - a constructive approach. In *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation"*, volume 2 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 118–126, 1995.

[32] Peter Heimann, Gregor Joeris, Carl-Arndt Krapp, and Bernhard Westfechtel. DYNAMITE: dynamic task nets for software process management. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 331–341, Washington, DC, USA, 1996. IEEE Computer Society.

[33] IBM. *Image and Workflow Library FlowMark Design Guidelines*. Number 2.3. IBM Redbooks, 1998.

[34] IBM and SAP. WS-BPEL extension for people BPEL4People. Technical report, July 2005 (Accessed date Oct 1 2008).

[35] WebFinance Inc. Business dictionary. *URL: http://www.businessdictionary.com/definition/process-management.html*, 2011 (Accessed February 3 2011).

[36] Michael Jackson. *Problem frames: analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[37] Stefan Jurack, Leen Lambers, Katharina Mehner, Gabriele Taentzer, and Gerd Wierse. Object flow definition for refined activity diagrams. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering*, volume 5503 of *Lecture Notes in Computer Science*, pages 49–63. Springer Berlin / Heidelberg, 2009.

[38] Ahmet Karatas, Halit Oguztüzün, and Ali Dogru. Global constraints on feature models. In David Cohen, editor, *Principles and Practice of*

Constraint Programming CP 2010, volume 6308 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010.

[39] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling*. John Wiley & Sons Inc., New Jersey, 2008.

[40] Sandy Kemsley and Steve Russell. Getting started with BPM part iv: The nature of work: Structured versus unstructured. *URL: `http://www.bpm.com/`*, 2011 (Accessed March 25 2011).

[41] Manuel Koch, L.V. Mancini, and Francesco Parisi-Presicce. Graph-based specification of access control policies. *Journal of Computer and System Sciences*, 71(1):1 – 33, 2005.

[42] Kleopatra Konstanteli, Tommaso Cucinotta, and Theodora Varvarigou. Optimum allocation of distributed service workflows with probabilistic real-time guarantees. *Service Oriented Computing and Applications*, pages 1–15, 2010. 10.1007/s11761-010-0068-1.

[43] OptXware Research & Development LLC. The VIATRA - I model transformation framework pattern language specification. *URL: `http://www.eclipse.org/gmt/VIATRA2/doc/ViatraSpecification.pdf`*, 2006 (Accessed August 10 2010).

[44] OptXware Research & Development LLC. The VIATRA -I model transformation framework users' guide. *URL: `http://www.eclipse.org/gmt/VIATRA2/doc/viatratut.pdf`*, 2007 (Accessed August 10 2010).

[45] Microsoft. Developing models for software design. *Visual Studio*, 2010.

[46] Michael zur Muehlen. *Workflow-based Process Controlling: Foundation,Design, and Application of Workflow-driven Process Information Systems*. Logos Verlag Berlin, 2002.

[47] Object Managment Group (OMG). Business process modeling notation (BPMN). Technical Report Version 1.2, January 2009.

[48] Tom Pender. *UML^{TM} Bible*. Wiley Publishing, Inc., Indianapolis, IN, 2003.

[49] Andras Pfeiffer, Botond Kadar, and Laszlo Monostori. Stability-oriented evaluation of rescheduling strategies, by using simulation. *Computers in Industry*, 58(7):630–643, September 2007.

[50] Jenny Preece. *Human-Computer Interaction*. Addison-Wesley, New York, 1994.

[51] Corrado Priami. Stochastic π-calculus. *The Computer Journal*, 38:578–589, 1995.

[52] Corrado Priami. Stochastic pi-calculus with general distributions. In *in Proc. of the 4th Workshop on Process Algebras and Performance Modelling (PAPM '96), CLUT*, pages 41–57, 1996.

[53] Ruedee Rangsaritratsamee, William G. Ferrell, and Mary Beth Kurz. Dynamic rescheduling that simultaneously considers efficiency and stability. *Computers & Industrial Engineering*, 46(1):1 – 15, 2004.

[54] Mohammad S. Raunak and Leon J. Osterweil. Effective resource allocation for process simulation: A position paper. In *6th International Workshop on Software Process Simulation and Modeling (ProSim)*, St. Louis, MO, USA, May 2005.

[55] J. Romn Ubeda and R. Allan. Stochastic simulation and monte carlo methods applied to the assessment of hydro-thermal generating system operation. In *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research*, volume 2, pages 1–23. Springer Berlin / Heidelberg.

[56] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: Towards a unified standard. In *In Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63, 2000.

[57] Ravi S. Sandhu. Role-based access control. In *Advances in Computers*, volume 46, pages 237–286. Academic Press, 1998.

[58] Sunil K. Sarin. Workflow and data management in inconcert. volume 0, page 497. IEEE Computer Society, Los Alamitos, CA, USA, 1996.

[59] Andy Schürr, Andreas Winter, and Albert Zündorf. Graph grammar engineering with PROGRES. In Wilhelm Schäfer and Pere Botella, editors, *Software Engineering ESEC '95*, volume 989 of *Lecture Notes in Computer Science*, pages 219–234. Springer Berlin / Heidelberg, 1995.

[60] Daniel G. Shimshak, Dru Gropp Damico, and Hope D. Burden. A priority queuing model of a hospital pharmacy unit. *European Journal of Operational Research*, 7(4):350–354, 1981.

[61] Charles W. Spry and Mark A. Lawley. Evaluating hospital pharmacy staffing and work scheduling using simulation. In *WSC '05: Proceedings of the 37th conference on Winter simulation*, pages 2256–2263. IEEE, 2005.

[62] MathWave Technologies. Mathwave: Data analysis & simulation. *URL: http://www.mathwave.com/*, 2011 (Accessed February 3 2011).

[63] OASIS . OASIS web services business process execution language (WSBPEL) tc. *URL: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel*, 2011 (Accessed March 3 2011).

[64] Adobe, BEA, Oracle, Active Endpoints, IBM, and SAP. WS-BPEL Extension for People (BPEL4People),. *URL: http://public.dhe.ibm.com/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf*, (Version 1.0), June 2007 (Accessed Oct 2 2008).

[65] Object Managment Group (OMG). BPMN elements and attributes. *URL: http://www.omg.org/bpmn/Documents/BPMN_Elements_and_Attributes.pdf*, 2009 (Accessed March 25 2011).

[66] Object Managment Group (OMG). OMG's MetaObject Facility (MOF). *URL: http://www.omg.org/mof/*, 2011 (Accessed March 15 2011).

[67] Object Managment Group (OMG). UML resource page. *URL:* ***http://www.uml.org/***, 2011 (Accessed March 5 2011).

[68] Paolo Torrini, Reiko Heckel, and István Ráth. Stochastic simulation of graph transformation systems. In *Fundamental Approaches to Software Engineering (FASE)*, volume 6013 of *Lecture Notes in Computer Science*, pages 154–157. Springer Berlin / Heidelberg, 2010.

[69] Illse Truter. Dispensing service research-pilot project. *Pharmaciae-Official publication of the South African Pharmacy Council*, 14(1):20–23, April 2006.

[70] Wil M. P. van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, 2002.

[71] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics). *Software and Systems Modeling*, V2(3):187–210, October 2003.

[72] VIATRA2. VIsual Automated model TRAnsformations framework. *URL:* ***http://wiki.eclipse.org/VIATRA2***, (Accessed August 5 2010).

[73] Hans Van Vliet. *Software Engineering: Principles and Practice.* John Wiley & Sons ltd, Etobicoke, Ontario, 2003.

[74] Markus Völter and Thomas Stahl. *Model-Driven Software Development.* Wiley & Sons, 1 edition, May 2006.

[75] Thomas von der Maβen and Horst Lichter. Determining the variation degree of feature models. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 82–88. Springer Berlin / Heidelberg, 2005.

[76] Eric W. Weisstein. Central limit theorem. *URL:* ***http://mathworld.wolfram.com/CentralLimitTheorem.html***, 2011 (Accessed March 10 2011). From MathWorld–A Wolfram Web Resource.

[77] Junchao Xiao, Leon J. Osterweil, Qing Wang, and Mingshu Li. Dynamic resource scheduling in disruption-prone software development environments. In David S. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 6013 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2010.