NOTE TO USERS

This reproduction is the best copy available.

Use Case and Task Models:
Formal Unification and Integrated Development Methodology

Daniel Sinnig

A Thesis
In the Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy at
Concordia University
Montreal, Quebec, Canada

September 2008

Your file Votre référence
ISBN: 978-0-494-63454-7
Our file Notre référence
ISBN: 978-0-494-63454-7

ABSTRACT

Use Case and Task Models:
Formal Unification and Integrated Development Methodology

Daniel Sinnig, Ph.D.
Concordia University, 2008

Use case models are the specification medium of choice for functional requirements, while task models are employed to capture User Interface (UI) requirements and design information. In current practice, both entities are treated independently and are often developed by different teams, which have their own philosophies and lifecycles. This lack of integration is problematic and often results in inconsistent functional and UI design specifications causing duplication of effort while increasing the maintenance overhead.

To address this shortcoming, we propose an integrated development methodology for use case and task models. Our methodology serves as a blueprint for practitioners to derive an iterative and incremental development process according to which the two artifacts are successively enhanced in a stepwise and integrated manner. With each step, it is verified that the resulting model is a valid refinement of its source model. For this purpose we define a suite of refinement relations for use case and/or task models and provide automated tool support.

The integrated development methodology is based on a formal framework, which defines a two-step mapping from a particular use case or task model notation to a common semantic domain. This two-step mapping results in a semantic framework that can be more easily validated, reused and extended. The intermediate semantic domains have been carefully chosen by taking into consideration the intrinsic characteristics of use case and task models. We selected *sets of partially ordered sets* (posets) and *nondeterministic finite state machines* (nFSMs) as semantic domains, supporting a true concurrent and interleaving model of concurrency, respectively.

During the course of our research, we also defined *DSRG-style use case models* and *Extended CTT task models* as improvements to their respective state-of-the-art counterparts. Each improvement has been carefully selected to ensure that the intent and nature of each model is preserved. In order to show that the set of posets semantics and the nFSM semantics coincide, we established a formal correspondence between the two semantics and prove that they are trace equivalent.

ACKNOWLEDGEMENTS

I am grateful to a number of individuals and organizations for their help and support:

My supervisors, Dr. Chalin and Dr. Khendek, for their guidance, sound judgment and direction throughout my doctoral research.

My committee members, Dr. Grogono and Dr. Ormandjieva, for their feedback and advice, especially during my research proposal. I have been deeply saddened by the passing of my former committee member Dr. Gohari.

NSERC (Natural Sciences and Engineering Council of Canada) for the financial support.

Homa Javahery, who spent hours brainstorming with me about my approach, and gave me a different perspective. I thank her for making my years as a graduate student so enjoyable and rewarding.

I thank the following colleagues and friends: Asif Dogar, Rajiv Abraham, Perry James, George Karabotsos, Stuart Thiel, Leveda Giannas, Stephen Barrett, Rozita Naghshin, Dr. Krishnan, Dr. Forbrig, Maik Wurdel, Jason Savard, Hanneke Beaulieu, Shirin Sadeghi, Thomas Kalinowski and Markus Stoy for their feedback.

Above all, I am grateful to my family. I thank my parents, my sister, and my dear girlfriend Jennifer for their active support throughout these past few years.

Table of Contents

List of Figures

List of Tables

1 Introduction

Software engineering (SE) is the systematic and disciplined application of sound engineering principles with the goal of achieving high quality software: software that is delivered on time, within budget and which meets the customer's needs [Leffingwell & Widrig 2003; Pressman 2005]. In modern SE processes, the development lifecycle is divided into a series of iterations [Larman 2004]. For each of the iterations, a set of *disciplines* are followed and associated activities are performed. The resulting artifacts are incrementally perfected until the software is ready to be shipped to the customer.

One of the main principles of software engineering is to gear the envisioned application to the needs of the stakeholders. This principle equally applies to the supported *functionality* as well as the conception of the *user interface* (UI). The former directly relates to stakeholders' core functional needs, while the latter relates to qualitative demands. Both aspects are not independent but are closely related. This becomes evident if we consider the fact that the UI is meant to grant users efficient, convenient and effective access to the functionality provided by the system. A good UI is designed according to the work practices of end users and serves as a facilitator between the user and the provided functionality.

1.1 Problem Statement

Despite the obvious interrelationship, UI design methods are not very well integrated with standard software engineering practices [Seffah et al. 2005]. Instead of having a single process, where UI design follows as a logical progression from a functional requirements specification, both entities are treated rather independently and are often developed by different teams, which have their own models and lifecycles [Kazman et al. 2003]. The apparent gap between software engineering and UI development has been noted by several authors [Juristo et al. 2001; Kazman et al. 2004; Sutcliffe 2005] and was partially addressed in our work [Sinnig et al. 2007] and the work of others [Clemmensen & Norbjerg 2003; Constantine et al. 2003; Kujala 2005]. However, to date, an integrated methodology does not exist. The following issues result directly from this lack of integration:

- Possible **conflicts during implementation** as general software development and UI design processes do not have the same reference specification and thus may produce inconsistent designs and implementations.

- **Duplication in effort** during development and maintenance due to redundancies and overlaps in the (independently) developed UI and software engineering models.

In this thesis we attempt to bridge the gap between software engineering and UI design by defining an integrated development methodology for **use case models** and **task models**. While the former are the predominant specification medium for functional requirements, the latter are commonly employed to capture UI requirements and design information. On one hand, use cases present user-system interaction by means of a main success scenario and a set of extensions. On the other hand, task models are hierarchically structured and decompose high-level user and system tasks into lower-level tasks until an atomic level has been reached.

Both artifacts share a common tenet and capture the interaction between actors and the system under development. An integrated development methodology for the two artifacts, however, does not yet exist.

1.2 Research Statement and Objectives

The primary objective of this thesis is to define an **integrated development methodology** for use case and task models. Such an integrated development methodology will serve as a blueprint for practitioners to derive an iterative and incremental development process according to which use case and task models are stepwise refined. We define a set of refinement relations for both artifacts and demonstrate that the deciding factor of whether a use case or task model is a valid refinement of a given source model depends on (1) the role of the involved artifacts in the software development lifecycle and (2) whether we compare a use case with a task model specification or artifacts of similar nature (e.g. two use case models or two task models).

In what follows, we discuss the key contributions of this thesis. Each contribution can be seen as a complementing milestone towards the main objective of this thesis, namely an integrated developing methodology for use case and task models.

Enhancement of Use Case and Task Models: We suggest and implement a set of enhancements to use case and task models. The enhancements are either (1) motivated by practical demand or (2) a result of cross-pollination between the two models. In the former case, we define a set of novel operators for task models. In the latter case, we take advantage of the fact that both models have individual strengths and weaknesses. We integrate the strengths of one model as an improvement of the other, while making sure that its very intent and nature is preserved.

Formal Framework for Use Case and Task Models: The theoretical basis for the envisioned integrated development methodology is a formal framework for use case and task models. Central to such a framework is the definition of syntax and semantics. We provide a formal syntax for use case models called *DSRG-style use case model*. It has been carefully chosen to provide a formal structure, while preserving the intuitive nature of use case specifications. In the case of task models we extend and completely formalize the concept of ConcurTaskTrees [Paternò & Santoro 2001] to define an *Extended ConcurTaskTrees* (ECTT) task modeling language.

To date, neither use case nor task models have formal *and* agreed upon semantics, not to mention a common semantics. Therefore, in this thesis we define a *common semantics* for DSRG-style use case models and ECTT task models. The semantic mapping is performed in two steps: First, the source models are mapped to respective intermediate semantic domains, followed by mappings to the common semantic domain, which is either based on *sets of partially ordered sets* (sets of posets) or *nondeterministic finite state machines* (nFSMs). We demonstrate that each semantic domain has individual strengths and weaknesses. While the former allows for the definition of non-interleaving semantics supporting true concurrency, the latter is limited to an interleaving model of concurrency but supports the distinction between choices made by the user and choices made by the system.

By using the trace set as a common denominator, we establish a formal correspondence between the two semantics and prove that they are trace equivalent; i.e. for a given use case or task model the set of traces in the set of posets semantics is equivalent to the set of traces in the nFSM semantics.

Tool Support and Validation: One fundamental principle of our integrated development methodology is that use case and task models are iteratively developed as a series of stepwise refinements. With each refining step it is important to ensure that the resulting artifact is a valid refinement of its respective source model. The verification of refinement should ideally be aided by supporting tools. Within this research we developed the tool *Use Case - Task Model - Verifier*. It accepts intermediate use case and task model specifications as input and performs the mapping to the semantic domain of nFSMs. It then uses a model-checking algorithm to verify refinement between both specifications (based on the desired refinement relation). In case of a verification failure, a violating counter-example is provided.

In order to validate the integrated development methodology, we partly formalize the requirements specifications of an "Invoice Management System" (IMS). We then use the *Use Case - Task Model - Verifier* to ensure the "IMS" task model is a valid refinement of the "IMS" use case model.

1.3 Thesis Organization

The organization of the remainder of this thesis is as follows:

In **Chapter 2** we discuss the background and the related work. In particular, we summarize key concepts in use case and task modeling and contrast their role in current development processes. We review common formalisms as potential candidates to serve as semantic domains and discuss concepts of refinement for event-based and state-based models.

In **Chapter 3** we present our integrated development methodology. We develop a set of key principles upon which the methodology rests and define several types of refinements for use case and/or task models.

In **Chapter 4** we provide an overall view of our framework for formalizing use case and task models. We introduce the "IMS" example, which will be used throughout the thesis, and define the syntactic portion of the framework: *DSRG-style use case models* and *ECTT task models*.

In **Chapter 5** we introduce Use Case Labeled Transition Systems (UC-LTS) and Generic Task Models (GTM) as intermediate semantic domains for use case and task models. We formally define the mappings from DSRG-style use case models and ECTT task models to their respective intermediate semantic domains.

In **Chapter 6** and **Chapter 7** we define the mappings into the semantic domains *sets of posets* and *nFSMs*. We provide necessary definitions, followed by the semantic mappings from UC-LTS and GTM to the respective semantic domains.

In **Chapter 8** we compare the two semantic domains in terms of their modeling capabilities and establish a formal correspondence between them. We show that for a given UC-LTS or GTM the set of posets semantics and nFSM semantics are trace equivalent.

In **Chapter 9** we formally define the different types of refinement between use case and/or task models identified in Chapter 3. The formalizations are given in the nFSM semantics and, if applicable, in the set of posets semantics. We also introduce the *Use Case - Task Model - Verifier* tool, which we developed to automate the verification of refinements.

Finally, **Chapter 10** summarizes our work, the main contributions, and presents future avenues for research in this area.

2 Background and Related Work

In this section we provide the necessary background information on use case and task models. For each model we present the main features and standard notations. We review how use case and task models are developed in popular development methodologies and discuss relevant related work with respect to the definition of formal semantics. We conclude this chapter by presenting concepts of refinement suitable for formally relating the use case and the task model.

2.1 Use Case Models

Use cases were introduced around 15 years ago by Jacobson [Jacobson 1992]. He defined a use case as a "specific way of using the system by using some part of the functionality." Use case modeling is making its way into mainstream practice as a key activity in the software development process (e.g. Rational Unified Process). There is accumulating evidence of significant benefits to customers and developers [Merrick & Barrow 2005].

The *use case model* captures the complete set of use cases for an application, where each use case specifies possible usage scenarios for a particular functionality offered by the system. A use case can be interrelated with other use cases in two ways: (1) The *<<include>>* relationship denotes the "invocation" of a use case by another one. More precisely, the behavior of the sub-use case is included at one or more specified locations defined in the base use case. The *<<extend>>* relationship factors out a so called "extending use case" which adds additional behavior to the base use case. The extension use case is invoked if the corresponding extension point has been reached within the base use case and the extension condition is fulfilled.

The use case model documents the majority of software and system requirements and as such, serves as a contract of the envisioned system behavior between stakeholders [Cockburn 2001]. Moreover, it is very useful in determining the boundary of the system. Only behavior defined by the use case model is part of the envisioned system. Interactions that are not specified are deemed outside of the system's boundary [Overgaard & Palmkvist 2004].

While some of the original concepts in use case modeling have evolved through the work of researchers and practitioners, the fundamental idea remains the same; that is, a use case describes the way a system is employed by its actors to achieve their goals [Armour & Miller 2001]. In other words, a use case captures the interaction between actors and the system under development. Actors represent users or entities (e.g. secondary systems) that interact with the system. By definition actors, either primary or secondary, are outside of the system boundary. The primary actor, typically a user, initiates the use case in order to accomplish a pre-set goal. Secondary actors play the role of supporting the execution of the use case and may participate in the interaction later [Gomaa 2005].

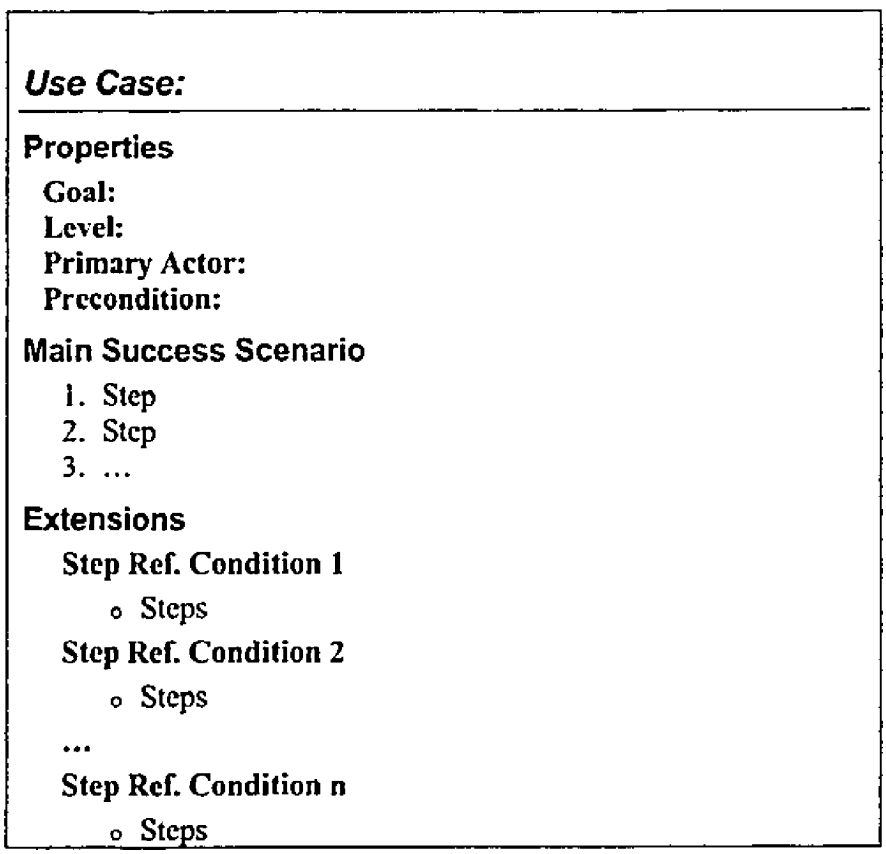


Figure 2.1. Use Case Template

Figure 2.1 depicts a template for a fully dressed use case as defined by Cockburn [Cockburn 2001]. Every use case starts with a header section containing various properties. The "primary actor" property identifies the actor who initiates the interaction specified by the use case. The "goal" property captures the very intent the primary actor has in mind when executing the use case. "Level" indicates the goal-level of the use case. While different goal-levels exist, -the most important ones are *summary*, *user goal* and

sub-function. The "precondition" property denotes a condition that must hold, in order to carry out the use case.

The core part of a use case is its main success scenario, which follows immediately after the header. It indicates the most common way in which the primary actor can reach his/her goal by using the system. A use case is completed by specifying the use case extensions. These extensions define alternative scenarios which may or may not lead to the fulfillment of the use case goal. They represent exceptional and alternative behavior (relative to the main success scenario) and are indispensable to capturing full system behavior. Each extension starts with a condition (relative to one or more steps of the main success scenario), which makes the extension relevant and causes the main success scenario to *branch* to the alternative scenario. The condition is followed by a sequence of action steps, which may lead to the fulfillment or the abandonment of the use-case goal and/or further extensions. From a requirements point of view, exhaustive modeling of use case extensions is an effective requirements elicitation device.

The main success scenario as well as each extension consists of a sequence of use-case steps. Different kinds of use-case steps exist, with the most frequently used ones being: *Atomic*, *choice*, *concurrent*, *goto* and *include*. *Atomic* steps are performed either by the primary actor or the system and do not contain any sub-steps. *Choice* steps provide the primary actor with the choice between several interactions. Each such interaction is (in turn) defined by a sequence of steps. *Concurrent* steps define a set of steps which may be performed in any order by the primary actor. *Goto* steps denote jumps to steps within the same use case. *Include* steps define the inclusion of a sub-use case.

2.2 Task Models

User-task modeling is by now a well understood technique supporting user-centered UI design [Paternò 2000]. In most UI development approaches, the task set is the primary input to the UI design stage. *Task models* capture the complete set of tasks that users perform using the application, as well as how the tasks are related to each other. The origin of most task-modeling approaches can be traced back to activity theory [Kuutti 1995], where a human operator carries out activities to change part of the environment (artifacts) in order to achieve a certain goal [Dittmar & Forbrig 2003]. Like use cases,

task specifications describe the user's interaction with the system. Their primary purpose is to systematically capture the way users achieve a goal when interacting with the system [Souchon et al. 2002]. More precisely, a task specification defines how the user makes use of the system to achieve a goal but also indicates how the system supports the involved (sub-)tasks.

Various notations for task models exist. Among the most popular ones are ConcurTaskTrees (CTT) [Paternò 2000], GOMS [Card et al. 1983], TaO Spec [Dittmar et al. July 2004], WebTaskModel (WTM) [Bomsdorf 2007], and HTA [Annett & Duncan 1967]. Even though all notations differ in terms of presentation, level of formality, and expressiveness, they share the following common tenet: Tasks are hierarchically decomposed into sub-tasks until an atomic level has been reached. Temporal operators are used to specify the execution order of tasks. Moreover tasks are typically further detailed with context-specific attributes such as 'preference', 'frequency' and context-independent properties such as pre- and post-conditions, manipulated artifacts, and the task type. The former attributes are specified relative to the user group and the used device, whereas the latter properties are universally valid.

Table 2.1. Temporal Operator Set of CTT

Operator	Syntax	Interpretation
Enabling	$t_1 \gg t_2$	Upon termination of t_1, t_2 becomes enabled.
Choice	$t_1 [] t_2$	Either t_1 or t_2 is executed. The execution of one task disables the other one.
Order Independence	$t_1 \boxplus t_2$	Execution of t_1 and t_2 in any order.
Concurrency	$t_1 \mid\mid\mid t_2$	Interleaved execution of t_1 and t_2 and their subtasks
Disabling	$t_1 [> t_2$	t_1 becomes deactivated as soon as the first action of t_2 is performed.
Suspend / Resume	$t_1 \mid > t_2$	At any time the execution of t_1 may be interrupted by t_2. After t_2 has finished its execution t_1 resumes.

Operator	Syntax	Interpretation
Iteration	t^*	t may be executed repetitively (0 or many times).
Optional Tasks	$[t]$	t may be executed or not.
Reference	$t \Downarrow$	Reference to a task named "t" defined elsewhere in the CTT task tree.

In what follows we describe in greater detail the task-modeling notation ConcurTaskTrees (CTT). CTT is a well accepted task-modeling language used for the systematic development of the user interface. Tasks are arranged hierarchically, with more complex tasks decomposed into simpler sub-tasks. CTT includes a set of binary and unary temporal operators. The former are used to temporally link sibling tasks, at the same level of decomposition, whereas the latter are used to identify optional and iterative tasks. A summary of CTT operators together with their intuitive interpretation is given in Table 2.1. We note that most binary operators (except for *suspend/resume*) have similar (yet not semantically identical) counterparts in LOTOS [Interactions 1987]. CTT is supported by a tool (CTTE) [Mori et al. 2002], which allows for creation and graphical visualization of CTT task specifications.

Figure 2.2. CTT Task Types

An example of such a CTT task specification is depicted in Figure 2.3. It shows a typical break down of a root task into sub-tasks. Tasks at the leaf level are called actions, since they are the tasks that are actually carried out by the user or the system. An exception to this rule is the task reference (denoted by the ⇩ symbol). These are not atomic tasks, but denote the invocation of the corresponding sub-task tree. CTT distinguishes between four different task types (Figure 2.2). Most important for UI design is to capture *interaction tasks*, *application tasks*, and *abstract tasks*[1]. While interaction tasks are performed by the

[1] *User tasks* represent activities that do not require the interaction with the UI (such as *thinking, problem solving*, etc...)

user (through the UI), application tasks are performed by the system and have an externally visible outcome to the user. Abstract tasks denote high-level tasks which can involve both interaction and application tasks.

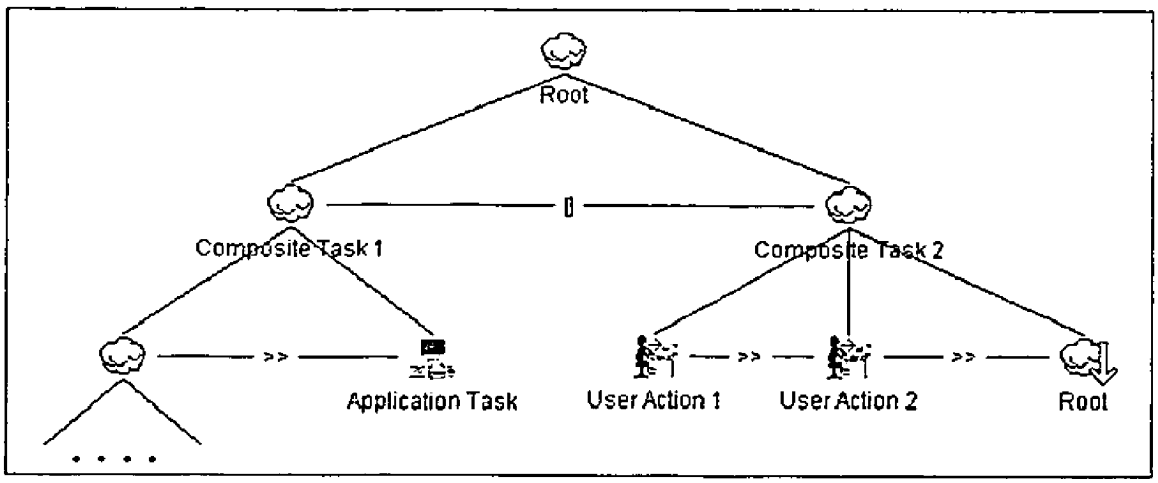


Figure 2.3. Sample CTT Task Specification

2.3 Related Work

In this thesis we propose an integrated methodology for the development of use case and task models. In this section, we review popular development methodologies driven by use case and task modeling. We then investigate alternative (formal) notations that have been suggested to capture the use case and the task model. Next, we provide an overview of formalisms supporting interleaving and non-interleaving models of concurrency, and discuss how they have been employed to define formal semantics. We conclude this section by summarizing the most important concepts of refinement that have been identified by the scientific community.

2.3.1 Development Methodologies Driven by Use Case and Task Models

To our knowledge, we are the first to attempt the definition of an *integrated* methodology for developing the use case and the task model, as the scientific literature treats both artifacts mainly as independent entities. Hence, in what follows, we review methodologies that are either driven by use case modeling, or driven by task modeling.

2.3.1.1 Development Methodologies Incorporating Use Cases

In use-case driven software development processes, including the Rational Unified Process (RUP) [Larman 2004], ICONIX [Rosenberg & Stephens 2006] and URDAD

[Solms 2005] the use cases defined for a system are the basis for the entire development process. For example, as portrayed by Figure 2.4 and according to RUP, the use case model drives all development work from initial gathering of requirements to design, implementation, deployment and testing.

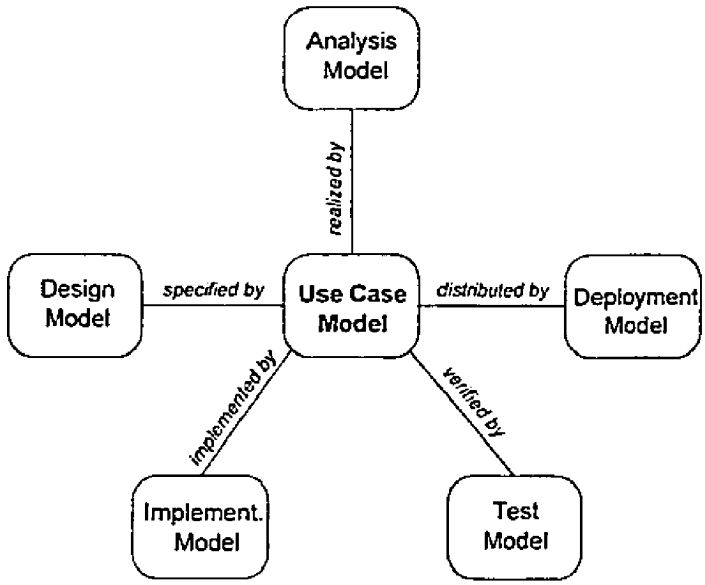


Figure 2.4. Role of the Use Case Model in the Rational Unified Process [adopted from Scott 2001]

In contrast to the obsolete Waterfall process model [Royce 1970] modern development processes promote an iterative lifecycle which dictates the successive enlargement and refinement of development artifacts through multiple iterations. With each iteration, a set of activities in various disciplines are performed such that the resulting artifacts are incrementally perfected. The development of the use case model is no exception to this rule.

In what follows, using the example of RUP, we discuss how the use case model is incrementally perfected through the development lifecycle. Even though RUP is an iterative process, development is divided into four sequential phases: Inception, Elaboration, Construction and Transition. In each phase (to more or less extend), the use case model is incrementally refined by either adding more details (which we call *structural refinement*) or filtering out low priority requirements in order to establish a requirements baseline (which we call *behavioral refinement*):

Inception: The intent of the inception phase is to establish a common vision for the objectives of the project and to determine its feasibility [Larman 2004]. For this purpose

only, approximately 10% of use cases (the ones that represent core functions) are written in detail. The remaining use cases are merely indentified (e.g. within a summary-level use case), but are not fully worked out.

Elaboration: The elaboration phase consists of multiple timeboxed iterations, during which architecturally significant parts of the system are incrementally built. The majority of the requirements are identified and specified. In line with this, the use case model is structurally refined (80-90% of all use cases are written in full detail) by adding additional information and by splitting previously atomic steps into sub-steps. Moreover, in an effort to stabilize requirements and to find a base line, low priority and superfluous features are eliminated from the use case model.

Construction and Transition: The focus of construction and transition is to complete and release the system. Most of the functional and non-functional requirements should have been already iteratively and incrementally stabilized. The degree of change of the requirements is much lower. Some minor use case writing (mostly structural refinement) does occur in which the remaining 10-20% of the use cases are completed and fully detailed.

2.3.1.2 Development Methodologies Incorporating Task Models

Several development methodologies for UI design exist. The majority of them are based on the task model with the most prominent examples being: MOBI-D [Puerta 1997], TERESA [Paternò et al. 2008], and TDD-CA [Wurdel et al. 2008]. MOBI-D defines an iterative development lifecycle. UI design begins with the elicitation of the user tasks. Next, the developer takes this description and builds task and domain models. The models are then integrated so that domain objects are related to the user tasks for which they are relevant. This forms the basis for the presentation model from which a UI design is derived.

TERESA is a development framework for developing UI designs for multiple user interface (MUI) applications. The framework defines a number of steps, starting with the specification of a task model for the MUI application [Marucci et al. 2003] and resulting in the derivation of a set of UI designs targeting multiple devices. TDD-CA is a task-based development methodology for cooperative applications, according to which

different "versions" of a task model are used throughout the development lifecycle. As depicted in Figure 2.5, it distinguishes between analysis, requirements and design task models.

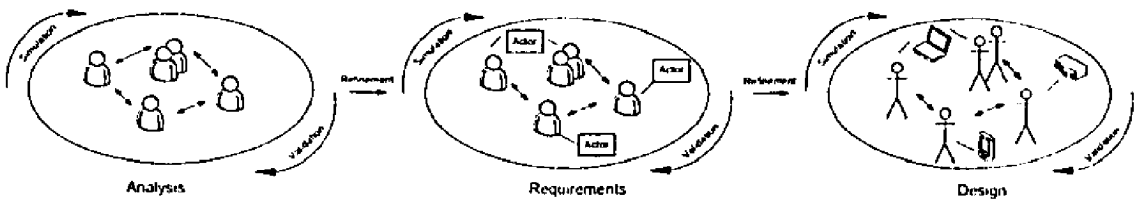


Figure 2.5. Development Lifecycle Proposed by TDD-CA [Wurdel et al. 2008]

Analysis: The purpose of analysis is to understand the user's behaviors, their collaborations and interactions. Consequently, the *analysis task model* captures the current work situation and highlights elementary domain processes as well as exposes bottlenecks and weaknesses of the problem domain. The focus is on the actual users while the envisioned interactive system is not yet taken into account.

Requirements: When moving to the requirements stage, the analysis information is further refined by taking into account the (functional) support of the envisioned interactive application (as captured in the use case model). Correspondingly, requirements level task models specify the envisioned way tasks are performed using the system under development. That is, tasks that were formerly performed by the user may now be taken over by the envisioned interactive system.

Design: During design, the various tasks of the requirements model are further refined, for example, tailoring the task set to a particular target device and taking into account its interaction capabilities.

2.3.2 Formal Notations for Use Case and Task Models

In Sections 2.1 and 2.2 we introduced structured prose and hierarchical task trees, respectively as standard notations for the use case and the task model. In this section we review alternative (more formal) notations that have been suggested by the scientific community.

2.3.2.1 Review of Formalisms

Most prominent alternative notations for the use case and task model are message *sequence charts* (MSCs), *statecharts* and *extended finite state machines* (EFSMs). Each formalism will be defined in the following.

Message Sequence Charts [ITU-T 1999] are a graphical notation for capturing patterns of interaction between processes or objects. Interaction scenarios are primarily described in terms of message passing between involved instances. The MSC notation consists of two core components: The basic MSC and the high-level MSC. Basic MSCs represent one particular scenario. They consist of a finite set of instances, which represent objects or processes of the system. The instances communicate with each other in the form of one-to-one message passing. Additionally, basic MSCs may be augmented with conditions, inline expressions, actions, timers, etc. High-level MSC are used to compose several basic MSCs and their embedded scenarios in order to derive a more complete behavioral specification of the system. More precisely, a high-level MSC can be seen as "a graphical overview of the relation between the MSCs contained" [Mauw & Reniers 1994]. The Unified Modeling Language [UML 2004] defines UML Sequence Diagrams and Interaction Overview Diagrams as counterparts for basic MSCs and high-level MSCs.

Statecharts [Harel 1986] and **EFSMs** [Wagner et al. 2006] belong to the family of state transition systems. Both consist of a set of states (with one state being the initial state), transitions and possibly actions. While both models differ in terms of expressiveness, complexity and semantics, they share the following common tenet: States are connected by transitions. A transition indicates a state change, which may be guarded by conditions and may trigger the execution of actions. Given an input, the machine traverses a set of states according to the transition function.

2.3.2.2 Formal Notations for Use Case Models

In the literature MSCs and UML Sequence Diagrams have been identified as suitable models to capture the interactions specified in use cases. Li [2000] describes a translation method that details how a "normalized" textual use case is translated into UML Sequence Diagrams. Each use case step is represented by a corresponding message. The

composition of the sequence diagrams is guided by the control constructs and sequencing information in the use case. Sinha et al. [2007] present an approach which generates EFSM models from textual use case specifications. Use cases are assumed to be UML 2.0 compliant and to possess a formal structure. Each use case step belongs to a predefined category (e.g. output statement, query statement, update statement, etc.). The resulting EFSM is behaviorally equivalent to the use case in the sense that it allows the same set of scenarios.

In pursuit of the overall goal of automatically generating test cases, Fröhlich and Link [2000] present a transformation algorithm that derives a statechart model from a given set of textual use cases. Each use case is assumed to correspond to the structure suggested by Cockburn [2001]. In their approach, use case steps that are performed by the system are represented by actions, whereas steps performed by the primary actor are modeled as events, causing state transitions. Use case extensions are defined through sub-states of the state representing the corresponding step in the main success scenario. Nested extensions (e.g. extensions of extensions) however are not supported. Inclusion of sub-use cases is modeled through nesting of state charts; i.e. the statechart representing the included use case is inserted as a "composite" state into the statechart representing the super-ordinate use case.

In the approach by Mizouni et al. [2007] use case automata (UCA) are used to represent the behavioral aspects of individual use cases. Since each use case is only a partial specification of the system, composition of several use cases is needed to obtain a complete system specification. For this purpose the authors formally define a set of composition operators for UCAs. The composition itself is guided by evaluating so-called UCA expressions and makes use of a technique called "label matching".

2.3.2.3 Alternative Notations for Task Models

In contrast to the use case model, the standard notation for the task model, viz. hierarchical task trees, is a formal representation, amendable for analysis and sophisticated tool support. Consequently, fewer approaches (relative to the use case model) exist that attempt to formalize task models using an alternative formalism. One noteworthy approach is the effort by Lu et al. [2003], who attempt to integrate object-

oriented analysis and design with task modeling. The presented approach generates UML diagrams (including sequence diagrams) from a given task model specification. In the case of sequence diagrams, the obtained result includes only the interactions between the user and the perceivable system objects, with atomic tasks being roughly equivalent to a message sent between objects.

Klug & Kangasharju [2005] propose a formalization for task models where a task is not regarded as an atomic entity (like in CTT) but has a complex lifecycle, modeled by a so-called task state machine. In [van den Bergh 2007] entire task expressions are translated into state charts. As a result, a generic state machine is created for leaf tasks as well as for complex task expressions which are translated according to the used temporal operators. In [Bomsdorf 2007] a more elaborate life cycle for tasks is defined. The work is focused on the development of web application and considers external events related to web technology (e.g. session timeouts and user aborts).

2.3.3 Semantic Domains

A cornerstone for defining an integrated development methodology is a formal semantics for the use case and the task model. Both models belong to the family of scenario-based notations. For such scenario-based specifications, the behavioral aspects of a system represent the important features to describe. Among the approaches which define semantics for scenario-based notations, there are two main methods: Those with *interleaving semantics* and those with *truly concurrent semantics*. Examples of the former include process algebras (e.g. CSP [Hoare 1985], CCS [Milner 1980], ACP [Baeten & Weijland 1990]) and labeled transition systems (LTS) [Brinksma et al. 1987]. Examples of the latter are partially ordered sets (posets) [Pratt 1986], trace theory [Mazurkiewicz 1995], Petri net theory [Petri 1962], and prime event structures [Winskel 1980].

In what follows, for each method we peruse two of the most popular formalisms (*process algebra* and *transition systems* for interleaving semantics and *partially ordered sets* and *Petri nets* for true concurrent semantics) and discuss how they have been used for the definition of formal semantics.

2.3.3.1 Formalisms with Interleaving Semantics

A formalism that has been widely used to define *interleaving* semantics of scenario-based notations is process algebras. In this approach, the behavior of a system is modeled by a set of (possibly concurrently running) processes. The formalism itself is presented as a formal calculus with associated "deduction/transformation" rules for reasoning about algebraic specifications. Several process algebraic theories exist, which can be distinguished by their alphabet of elementary actions, the set of operators and the equivalence relation used. Among others, the most influential ones are: Calculus of Communicating Systems (CCS) [Milner 1980], Communicating Sequential Processes (CSP) [Hoare 1985] Algebra of Communicating Processes (ACP) [Baeten & Weijland 1990], and Language of Temporal Ordering Specification (LOTOS) [Bolognesi & Brinksma 1987].

The International Telecommunication Union (ITU) has published a recommendation for the formal semantics of basic Message Sequence Charts (MSCs) based on the Algebra of Communicating Processes (ACP) [Baeten & Weijland 1990; ITU-T 1996]. This work is a continuation of preliminary research first established by Mauw and Reniers [1994]. In more recent work, Rui also suggest a process algebraic semantics for use case models, with the overall goal of formalizing use case refactoring [2007]. In their approach, scenarios are represented as basic MSCs—as suggested by [Regnell et al. 1996]. In Rui's proposal, he assigns meaning to a particular use case scenario (episode) by partially adapting the ITU MSC semantics. In addition, semantics are defined for related scenarios of the same use case as well as for related use cases. The following use case relations are formally defined: *includes*, *extends*, *generalization*, *proceeds*, *similar*, and *equivalence*.

Paternò and Santoro define formal semantics for a subset of the task model notation CTT (ConcurTaskTrees) based on LOTOS [2003]. Tasks and sub-tasks from the CTT task model are mapped in a one-to-one fashion to LOTOS process specifications. Temporal relations between tasks are mapped to their corresponding LOTOS counterparts.

Transition Systems: Transition systems as defined by Keller [1976] are an abstract relational model (mainly) based on two primitive notions: Those of a state and those of a transition [De Nicola 1987]. Popular instances of transition systems are *Labeled*

Transition Systems (LTS) and *Nondeterministic Finite State Machines* (nFSM). Both formalisms define graph structures in which nodes represent *states* and edges (transitions) represent *state changes*. State changes are triggered by actions, which may or may *not* be observable. In contrast to LTSs, the state space of nFSMs is restricted to finite sets only. Additionally nFSMs carry the concept of a final (accepting) state, which typically, is not part of a conventional LTS definition.

In the literature, transition systems have been identified as a popular fundamental model to define interleaving semantics. Most notable is the definition of structural operational semantics for most of the process algebras discussed in the previous sub-section. By means of transition rules, a given process expression (e.g. choice composition) is mapped to a corresponding operation defined on LTSs. In [Sinnig DSV-IS 2007], we proposed a preliminary common nFSM formalization for the use case and the task model for the purpose of formally verifying that a given task model is consistent with a respective use case specification. Eshuis presents a translation from UML activity diagrams to finite state machines [2006]. LTS semantics for UML statecharts is proposed by the work of [Eshuis & Wieringa 2000]. In both approaches the authors employ model-checking techniques to verify properties such as data integrity, absence of conflicts and consistency to related UML models such as class diagrams.

Finally we note that transition systems form the operational basis for model-checking tools such as FDR [Roscoe 2005], LTS Analyzer [Magee & Kramer 1999] and the CADP tool set [Garavel et al. 2007], which analyze a transition system for properties of interest (such as deadlock or livelock freedom) or mutually compare two given specifications. For the purpose of the latter, various equivalence and refinement relations have been defined. A discussion of the most popular ones can be found in Section 2.3.4.

2.3.3.2 Formalisms with Non-Interleaving Semantics

In this sub-section we discuss two formalisms to define non-interleaving (true concurrent) semantics, namely partially ordered sets and petri nets.

A **partially ordered set (poset)** is a set of elements which are placed into order by a partial-order relation. The order can be partial and hence not all elements need to be

mutually comparable (relatable). A partial order is defined as a binary relation, which is reflexive, antisymmetric and transitive.

In [Zheng 2004] a non-interleaving semantics for timed MSC 2000 [ITU-T 1999] based on timed labelled partial order sets (lposets) are proposed. A timed lposet additionally contains a labeling function which assigns a label to each event. The labels serve as an indicator for the corresponding event type. Possible event types are: *message input*, *message output*, *internal action*, *start timer*, *stop timer* and *timeout*. Furthermore Zheng defines two functions which attach timing constraints to events in order to specify the time range within which an event could occur and to define delays between two events. The semantic mapping is performed by associating an MSC with a set of timed lposets which entail the possible execution scenarios of the MSC.

Partial order semantics for un-timed MSC have been proposed by Alur et al. [1996] and Katoen and Lambert [1998]. On the one hand, the approach of Alur et. al. proposes semantics for a limited subset of MSCs, which merely allows message events as possible MSC events types. On the other hand the approach to MSC semantics of Katoen and Lambert is more complete and maps MSCs to a set of partial order multi sets (pomsets). Preliminary results towards the definition of a common semantic model for use case and CTT task models were reported in [Sinnig et al. 2006; Sinnig et al. 2007]. In [Sinnig et al. 2007] we defined an extensible semantic framework, in which use cases and CTT task specifications are first mapped into intermediate semantic domains, which in turn are mapped to the semantic domain of sets of posets.

Petri nets were introduced in 1962 by C. A. Petri [Petri 1962] as a true concurrent model to capture the behavioral dynamics of distributed systems. Formally, Petri nets are directed, connected, bipartite graphs in which each node is either a *place* or a *transition*. Places may contain tokens and denote the local state whereas transitions denote actions. State transitions are denoted by a movement of tokens from one place to another, triggered by a firing transition. In basic Petri nets, tokens are indistinguishable and the firing of transitions is assumed to be instantaneous (zero time units). Popular extensions to basic Petri nets are "Colored Petri nets" [Cost et al. 2000] and "Times Petri nets"

[Zuberek 1980]. While the former uses a typed form of tokens, the latter allows the association of time delays with the transitions.

In the research community, Petri nets are an attractive tool to defining non-interleaving semantics for event and scenario-based notations. In [Kryvyy & Matvyeyeva 2007] MSC'2000 compliant message sequence charts are translated into a set of Petri nets. The various Petri nets are then, based on the pre-condition statements of each MSC, sequentially composed in order to synthesize the system into one consolidated Petri net. Fernandes et al. [2007] present an approach to translate use cases into Colored Petri net models. It is assumed that each use case is represented by a UML sequence diagram. The translation is performed in two steps in a top down manner: (1) The use case model is mapped into a global Petri net which contains placeholders for each individual use case. (2) Each placeholder is replaced by a sub-Petri net capturing the various scenarios of the use case.

Cheung et al. [2006] propose a synthesis methodology for use cases based on labeled Petri nets. The overall goal of their approach is to prove certain properties of the system, such as deadlock freedom or capacity overflow. Both properties play an important role in the domain of manufacturing systems, where a set of concurrent and asynchronous processes needs to be synchronized and coordinated. Probably the most comprehensive approach has been defined by Somé [2007]. In his work, the author proposes execution semantics for use cases by defining a set of mapping rules from well-formed use cases to basic Petri nets. A use case is deemed well-formed if it syntactically corresponds to the pre-defined meta-model and satisfies a set of consistency and well-formedness rules. The mapping to Petri nets is defined over the various components of the use case (e.g. use case step, extension, control flow construct, etc.).

2.3.4 Refinement

In the next chapter we present a development methodology according to which use case model and task model specifications are successively refined into more detailed specifications. With each refinement step, it is important to verify that the more detailed specification is a valid refinement of its base specification. In general, refinement of specifications can be classified into event-based and state-based approaches. The former

specify the behavior of the system in terms of temporal ordering of events [Butler 1992], whereas the latter specify system behavior in terms of state transformations using pre- and post-conditions.

Within the context of use case and task modeling, the important aspects to describe are the sets of allowed interaction sequences of the envisioned system. Taking this into consideration together with the fact that a scenario can be seen as a sequence of events, event-based approaches seem to be more applicable than state-based approaches. In what follows, we will review event-based refinement in detail and provide a brief overview of state-based refinement.

2.3.4.1 Refinement of Event-based Models

In event-based approaches, system behavior is characterized by the set of possible interaction scenarios. A scenario consists of finite or infinite sequence of events that cause changes in the system state. The state space itself, however (at least in purely event-based approaches), is not observed [Butler 1992]. Different types of events exist representing different types of user or system actions. Typical examples are user input, system output, sending/receiving a message, and performing an internal action.

Different notions of refinement for event-based models have been proposed in the literature. Khendek et. al [2001] propose a refinement relation between basic MSCs. It ensures that a scenario, described in the source MSC specification, is also available in the refined specification. In much the same vein, a scenario that is forbidden in the source specification must never occur (or be derivable from) in the refined specification [Li 2000]. In other words, the behavior of the source MSC must be preserved in the target MSC. Events defined in the source MSC should also occur in the target MSC, and the relative order of these events needs to be preserved. The order of newly introduced events is not restricted.

Various equivalences have been defined for labeled transition systems (LTSs). Some are finer than others, which means that the criteria, that need to be fulfilled to call two LTSs equivalent, are stronger. In the case of *trace equivalence* [Bergstra 2001], two systems are deemed equivalent if they have the same set of traces. A finer notion of equivalence is obtained if refusal properties are also taken into account. Two systems are deemed *testing*

equivalent [Brinksma et al. 1987] if, in addition to trace equivalence, they have the same blocking properties. That is, whenever one system may refuse an interaction, the other system may also refuse the same interaction. If additionally, we take into account the internal (non-observable) states of the system, we obtain an even finer equivalence, namely, the *strong bisimulation* equivalence. It requires that if one LTS can perform a transition labeled 'a' then the other LTS must also be able to perform a transition labeled 'a', in such a way that the resulting states are again related. A formal definition can be found in [Park 1981].

In addition to the equivalences, various refinement preorders (reflexive and transitive relations) have been proposed in the literature. Amongst others the most popular ones are *reduction* [Brinksma et al. 1987] and *extension* [Brinksma et al. 1987; De Nicola 1987]. The *reduction* preorder defines a refining specification as a proper reduction of a specification, if it results from the latter by resolving choices that were left open. In this case, the refining specification may have less traces. In the case of the *extension* preorder, two specifications are compared for refinement by taking into account that one specification may contain behavioral information, which is not present in the other specification. In particular, a specification S_1 *extends* a specification S_2, if S_1 may perform any sequence of interaction that S_2 may perform. In addition, S_1 can only refuse an event 'a' after a sequence of events accepted by S_2, if S_2 can do the same. We note that the extension preorder "induces" the testing equivalence. That is, if S_1 extends S_2 and S_2 extends S_1 then S_1 and S_2 are testing equivalent [Khendek & Bochmann 1995].

2.3.4.2 Refinement for State-based Models

In state-based approaches, the behavior of a program is specified in terms of state transformations. State changes are performed by executing statements. Refinement of state-based models is concerned with the question of whether an implementation satisfies a given specification. In [Dijkstra 1976] and [Back 1988; Morgan 1988] specifications are denoted by pre- and post-condition. Weakest precondition transformers [Dijkstra 1976] and refinement calculus [Back 1988] are used to prove that the implementation satisfies the specification. Other approaches to state-based refinement are defined in line with the Vienna Definition Language (VDM) [Jones 1990] and Z [Spivey 1989], where

specifications are based on predicate calculus and set theory. Noteworthy is also the approach by [Abadi & Lamport 1991], which introduces refinement mapping as a means to prove that a lower-level specification correctly refines a higher-level specification.

2.3.5 Conclusion

In the remainder of this thesis we define an integrated development methodology for the use case and the task model. At the syntactic level individual use cases are represented in a format similar to Cockburn's "fully dressed" style [2001] according to which a use case is divided into a header section, a main success scenario and a set of extensions. For the task model syntax we use a notation similar to the hierarchical task-tree notation popularized by CTT [Paternò 2000]. Tasks are hierarchically decomposed into sub-tasks until an atomic level has been reached. Temporal operators are then used to specify the execution order of tasks.

Similar to RUP and TDD-CA we assume that both artifacts are developed iteratively. With each iteration, additional information is added to the model and/or superfluous features are eliminated. In contrast to TDD-CA, we use task models only at the requirements and design stage. We do not take into account the use of task models during analysis to capture behavioral characteristics of the user. In fact, we believe that for this purpose, other techniques such as personas [Pruitt & Grudin 2003] and user profiles [Mayhew 1999] are more suitable.

A cornerstone of the integrated development methodology is the definition of a common formal semantics for the use case and the task model. The semantic framework defined in this thesis is inspired by the lposet approach proposed in [Zheng 2004] and is a continuation of preliminary work first reported in [Sinnig et al. 2006; Sinnig et al. 2007]. Similar to the research presented by Mizouni [2007], we employ labeled graph structures as an intermediate notation for use cases. Analogous to [Sinnig et al. 2007] we use the nFSM formalism to define interleaving semantics for use case and task models. The integrated development methodology is supported by a set of refinement relations. For this purpose we adopt (with a few modifications) *trace equivalence* and the *reduction preorder*. The proof of refinement is supported by our tool *Use Case - Task Model –*

Verifier, which uses a model-checking algorithm similar to the one used by FDR [Roscoe 1994].

3 Integrated Development Methodology

In the previous chapter, the main characteristics of use case models and task models were presented. In what follows, we will analyze and compare both kinds of artifacts in order to integrate them in a development methodology where use cases are employed to document functional requirements and task models are used to describe UI requirements or designs. We will then define several types of refinements for relating these two artifacts.

3.1 Development Methodology for Use Case and Task Models

Use case and task models are both scenario-based and as such capture sets of usage scenarios of the system. On one hand, a use case describes system functionality by means of a main success scenario and extensions. On the other hand, a task specification captures user-system interactions within a hierarchical task tree. At a certain level of abstraction, both models can be used to capture the same information. In our integrated development methodology, however, use case models are employed to document functional requirements whereas task models are used to describe UI requirements and/or designs. We identify two main differences that are pertinent to their purpose of application:

1. In use case models, requirements are captured at a higher level of abstraction whereas task models are more detailed. Hence, the atomic actions of a task specification are often lower-level UI details that are irrelevant (actually contraindicated [Cockburn 2001]) in the context of a use case.

2. Task models concentrate on aspects that are relevant for UI design and as such, their specified usage scenarios are strictly depicted as input-output relations between the user and the system. System interactions that are hidden from the end user (e.g. involvement of secondary actors or internal computations) as specified in use case models are *not* captured.

Ideally, the functional requirements captured in use cases are independent of a particular user interface. On the contrary, the requirements and design information captured in task

models takes into account the specifics of a particular type of user interface. In other words, the use case model captures the bare functional requirements of the system, which are then "instantiated" to a particular type of user interface by means of a task model specification. If the application supports multiple UIs (e.g. Web UI, GUI, Mobile, etc.) then one use case is refined by several task models; one for each "type" of user interface.

The integrated development methodology for use case and task models rests upon the following four guidelines:

(1) Use case models are used for the purpose of capturing functional requirements, whereas task models are specifications of UI interaction requirements and design details.

(2) The development of the use case model precedes the development of the task model. That is, task models are developed based on an underlying functional requirements specification.

(3) While use case models are exclusively used at the requirements stage, task models may be used at the requirements and/or at the design stage. When the task model is used as a requirements artifact, a detailed specification of the UI is considered part of the contract between stakeholders about the envisioned interactive application, whereas when exclusively used as a design document, the development of the UI is left up to the designer.

(4) Use case models may be developed into further detailed use case models or task models through a series of refinement steps. With each refinement step it is important to verify that the refining model is a valid refinement of its source specification.

Figure 3.1 portrays all possible refinement relations between use case models and task models. At the *requirements stage* a source use case model (m_{A1}) may be further refined by a more detailed use case model (m_{A2}) or refined by a task model (m_{A3}), which takes into account UI specific interaction requirements. Furthermore a task model (m_{A3}) at the requirements stage may be further refined by a more elaborate requirements-level task model (m_{A4}). When moving from the *requirements to the design stage* a use case model

(m_{B1}) or a (requirements-level) task model (m_{A4}) may be refined by design-level task models (m_{A4} and m_{B1}, respectively). At the *design stage*, a task model may be further refined by a more elaborate task model, which takes into account additional UI specific interaction details.

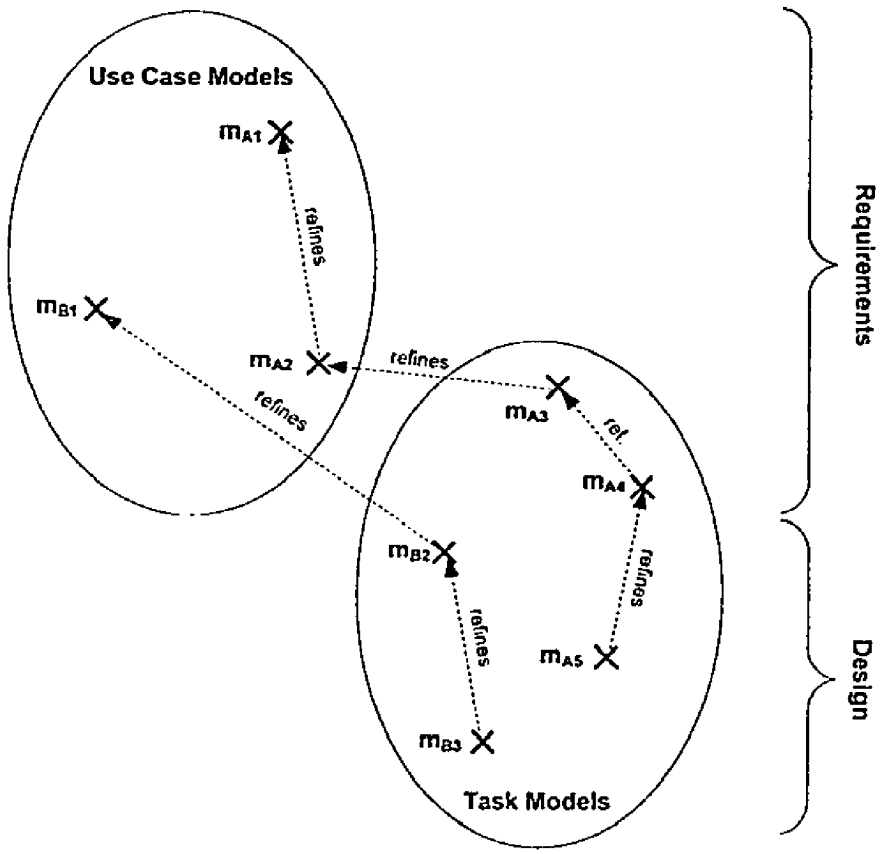


Figure 3.1. Use Case and Task Models in SW Development

The interpretation of what constitutes a valid refinement depends on the artifacts involved, as well as on their purpose in the software lifecycle. This will be discussed in the next section.

3.2 Refinement between Models

Generally, refinement between models can take two different forms: (1) Structural refinement, which consists of breaking previously atomic use case steps or tasks into sub-steps and sub-tasks respectively; and/or (2) Behavioral refinement, which restricts the set of possible scenarios.

Structural Refinement. The refined model may contain more information than its base model. This can be achieved by further refining the atomic units (atomic use case steps or leaf tasks) of the super-ordinate model. It is important to ensure that structural refinement preserves the "type" of the refined use case step or leaf tasks respectively. For example, a use case step performed by the system may only be refined to a set of use case steps of the same type. The same applies to the refinement of tasks in the task model.

Behavioral Refinement. The refined model must not allow more scenarios than the original specification. In some cases, the refined model may even further restrict the set of allowed scenarios. Whether this is possible or not depends on the models involved and their purpose in the software lifecycle. A summary of valid behavioral refinements is provided in Table 3.1. As depicted, each case requires its specific notion of refinement.

Table 3.1. Valid Behavioral Refinements

Case	Source Model	Refining Model	Allowed Behavioral Refinement
Requirements vs. Requirements			
1	Use Case Model	Use Case Model	Restriction of User Choices
2	Use Case Model	Task Model	Restriction of User Choices
3	Task Model	Task Model	Restriction of User Choices
Requirements vs. Design			
4	Task Model	Task Model	Scenario Equivalence
5	Use Case Model	Task Model	Scenario Equivalence
Design vs. Design			
6	Task Model	Task Model	Scenario Equivalence

In the following sub-sections each of the listed refinement cases and their rationale are discussed in detail.

3.2.1 Use Case Model – Use Case Model Refinement

At the requirements level, a base-use case model may be refined by a more detailed use case model. In such a case, the refinement is deemed valid if the refining use case model

does not allow more scenarios than its base model *and* if all system choices that happen non-deterministically from the primary actor's perspective are preserved.

Use Case: Login (Iteration 1)
Goal: Customer logs into the system.
Level: Sub-function
Primary Actor: Customer
Main scenario

1. Primary Actor provides username / password for authentication. ($m_{A1}.prUP$)
2. System ensures that provided username and password are valid. ($m_{A1}.vaUP$)
3. System informs the primary actor that login was successful. ($m_{A1}.inSU$)
4. *The use case ends successfully.*

Extensions

1a. Primary Actor decides to use fingerprint authentication.

1a1. Primary Actor provides his/her fingerprint for authentication. ($m_{A1}.prFP$)
1a2. System ensures that the provided fingerprint is valid. ($m_{A1}.vaFP$)
1a3. *The use case resumes at step 3.*

1a2a. The provided fingerprint is invalid:

1a2a1. The system informs the customer that the provided fingerprint is not in the database. ($m_{A1}.inFI$)
1a2a2. *The use case ends unsuccessfully.*

2a. The provided username / password combination cannot be validated:

2a1. The system informs the customer that the provided username and/or password could not be validated. ($m_{A1}.inCI$)
2a2. *The use case ends unsuccessfully.*

Figure 3.2. "Login" Use Case (m_{A1})

To illustrate this refinement principle, let us consider the "Login" use case given in Figure 3.2. In order to login, the primary actor has the choice between providing a username and a password (*step 1*) or using his fingerprint (*extension 1a*) as authentication method. In both cases, the system validates the login information and either informs the primary actor of the success or failure of the login process.

Table 3.2. Scenarios Allowed by "Login" Use Case (m_{A1})

Number	Description
1	Primary actor logs in *successfully* using a *username and password.*
2	Primary actor *fails* to login using a *username and password.*
3	Primary actor *successfully* logs in using *fingerprint authentication.*
4	Primary actor *fails* to login using *fingerprint authentication.*

Table 3.2 depicts that the use case allows four different scenarios. The scenarios differ in terms of which authentication method is used and whether the login was successful or unsuccessful. While the choice of authentication method is made by the primary actor, the decision of whether the login is successful or unsuccessful is made by the system. In other words, the choice between login success or failure depends on internal system details (e.g. access rights, availability of authentication server, etc.), which are unknown to the user. Since the user does not participate in the decision making he/she views the choice as non-deterministic.

Within our integrated development methodology, use cases may be further refined by more specific use cases. It is important that each refining use case of m_{A1} should not allow more scenarios than the ones depicted in Table 3.2, but may only implement a subset of the scenarios as long as both possible system responses about the outcome of the login validation are preserved. The preservation of system choices is important as a restriction may lead to incomplete requirements which do not sufficiently capture exceptional and error cases (e.g. login failure).

A valid refinement of the use case is given in Figure 3.3. Clearly, the refining use case only implements a subset of the scenarios allowed by the base use case. The primary actor is no longer given the choice to select between textual and fingerprint authentication. Instead, only the textual login option is available. In a real project setting, this restriction, in terms of functionality, may be the result of filtering the requirements in order to establish a base line. Possibly, the feature of allowing fingerprint authentication has been attributed with a low priority or was "stamped" as too risky and hence was dropped in the refining use case. Note that during requirements filtering it is not allowed

to limit the system's response on the outcome of the authentication. Clearly, the login procedure may always result in a success case or a failure case. This distinction needs to be preserved in all refining models.

In addition to behavioral refinement, the use case in Figure 3.3 also structurally refines its base-use case. In particular, *step 2* (validation of login information) contains more details. First the system establishes a connection to an external authentication server and then has the authentication server validate the login information. The step is now associated with two extensions, which take place when the connection to the authentication server cannot be established (*2a*) or when the authentication fails due to an invalid username / password combination (*2b*). Both extensions lead to the premature (unsuccessful) termination of the use case.

Use Case: Login (Iteration 2)
Goal: Customer logs into the system.
Level: Sub-function
Primary Actor: Customer
Main scenario

1. Primary Actor provides username / password for authentication. ($m_{A2}.prUP$)
2. System connects to authentication server and has authentication server validate the provided login data. ($m_{A2}.cvUP$)
3. System informs the primary actor that login was successful. ($m_{A2}.inSU$)
4. *The use case ends successfully.*

Extensions

2a. Connection to authentication server failed:

2a1. The system informs the customer that the connection to the authentication server could not be established. ($m_{A2}.inCF$)

2a2. *The use case ends unsuccessfully.*

2b. The provided username and/or password is invalid:

2b1. The system informs the customer that the provided username and password are invalid. ($m_{A2}.inCI$)

2b2. *The use case ends unsuccessfully.*

Figure 3.3. Refined "Login" Use Case (m_{A2})

3.2.2 Use Case Model – (Req.) Task Model Refinement

A task model may behaviorally refine a given use case model by restricting the number of allowed scenarios. Within our integrated methodology task models capture UI requirements and design details, which are developed based on a given use case model.

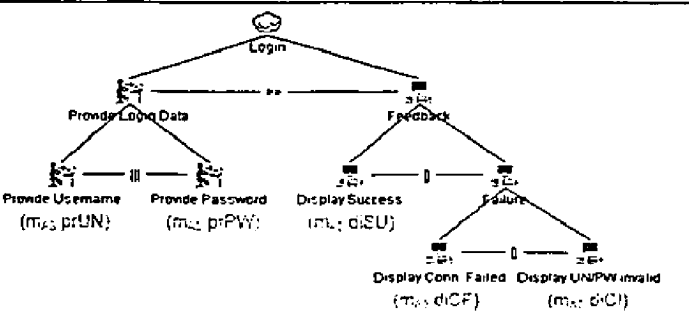
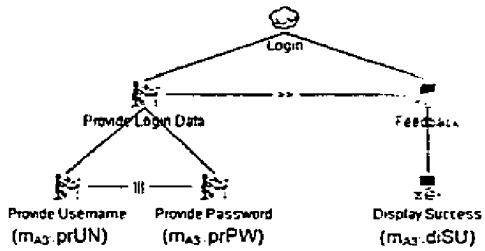
Hence, task models are geared to a particular user interface and as such must obey to its limitations. For example, a voice-activated user interface will most likely support less functionality than a fully-fledged graphical user interface.

As an example, let us reconsider the "Login" use case of Figure 3.3 and the two task models depicted by Table 3.3. Both task models specify the interactions required for logging in using a graphical user interface (GUI). While the task model on the left hand side implements the same set of scenarios as the use case, the task model on the right hand side only implements the case of a successful login. Login failure is not supported. If we apply the before-mentioned refinement criteria, we note that only the task model on the left-hand side is a valid refinement. The task model on the right-hand side is an invalid refinement, limiting the non-deterministic system feedback only to success messages. As a consequence, every case of login failure (for whatever reason) would lead to a UI stalemate, as the UI would have been designed to *not* provide failure feedback.

While the "Login" task model on the left hand side and the "Login" use case are behaviorally equivalent, the task model further refines the use case from a structural point of view. In particular, *step 1* of the use case has been decomposed into two separate user tasks, namely "Provide Username" and "Provide Password". As indicated by the concurrency operator (|||), they may be performed in any order. The subdivision of the use case step can be justified if we take into consideration that task models are more fine grained than use case models, since they take into account UI specific details. As such we can argue that providing the username and providing the password are two separate tasks, each requiring specific support for the UI (e.g. providing the username may be supported by an ordinary textbox, whereas providing the password should be supported by a "blinded" textbox, which does not reveal the entered password).

Finally we note that the "Login" task model does not implement any tasks representing internal system operations (such as *step 3* and *step 4* in the use case). This is due to the fact that in a task model, the system is viewed at a level of abstraction which focuses on input-output interactions and omits internal system operations. These internal system operations are irrelevant for UI design.

Table 3.3. Valid and Invalid Refinement of the "Login" Use Case (m_{A2})

Valid Task Model (m_{A3})	Invalid Task Model ($m_{A3'}$)
	

3.2.3 (Req.) Task Model – (Req.) Task Model Refinement

According to the second principle of our integrated development methodology (discussed in the previous section) it is assumed that task models are developed based on a given use case model. In the case of task model – task model refinement not only the original task model, but also all refining task models are (indirectly) based on an underlying use case specification. As a result, in order to ensure transitivity of refinement (i.e. every refining task model is also a valid refinement of the base-use case model), we require that only user choices may be restricted at the requirements level. Limiting a system choice to allow fewer alternatives is an invalid refinement.

A valid refinement of the "Login" task model (m_{A3}) (Table 3.3) is portrayed in Figure 3.4. The refining task model implements a subset of the scenarios offered by its base-task model. While the system choice between displaying a success message and displaying a failure message has been preserved, the previously arbitrary order of providing the username and the password has been restricted to the sequential order. Within the context of a real project, it is easy to envision that the behavioral refinement (i.e. sequential order of providing the username and the password) was necessary to address a newly-introduced security requirement, according to which the username and the password should never be entered and displayed on the same page; a common practice used by many banking sites to lower the probability of identity theft (e.g. Desjardins – Caisse Populaire [Desjardins 2008]).

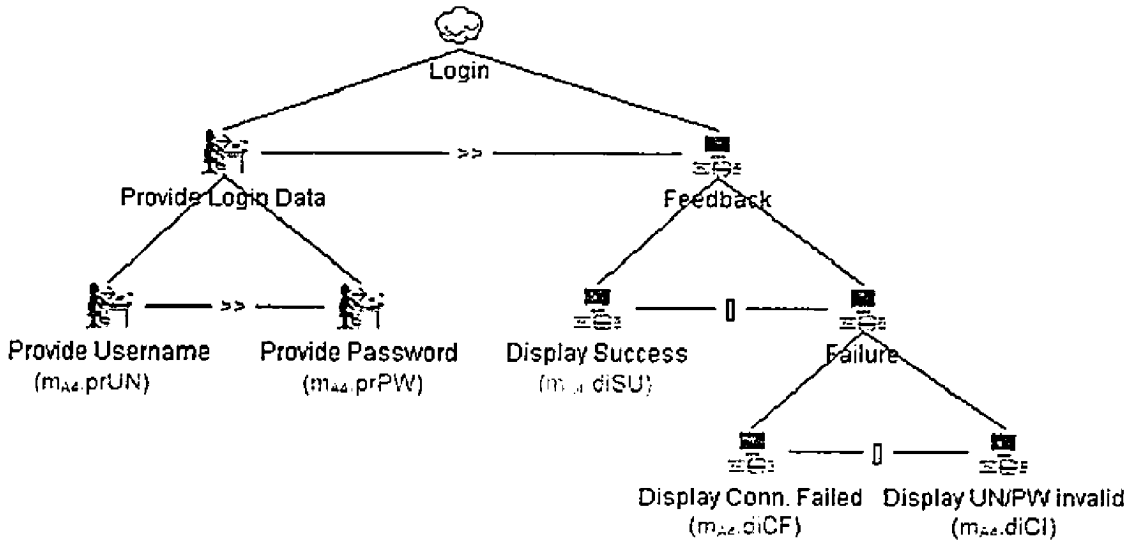


Figure 3.4. Refined Requirements Level Task Model (m_{A4})

3.2.4 (Req.) Task Model – (Design) Task Model Refinement

The artifacts gathered during requirements specification are part of the contract between stakeholders about the envisioned application. When moving from a requirements model to a design model, it is important to ensure that the refining model truly implements the requirements. As a consequence, the refining model may only add information in terms of structural refinement, but must not restrict or extend the number of possible scenarios. If a requirements-level task model is refined by a design-level task model we require that each task of the requirements model be present in the design level task model and that the execution orders of all "implemented" requirements-level tasks be preserved.

In Figure 3.5 a design-level refinement of the task model given in Figure 3.4 is presented. Clearly, both task models allow the same set of scenarios. The only difference between them is that in the design-level task model, the tasks of entering the username and password have been further detailed. In particular the tasks have been refined with details of *how* the user enters and possibly modifies the authentication coordinates.

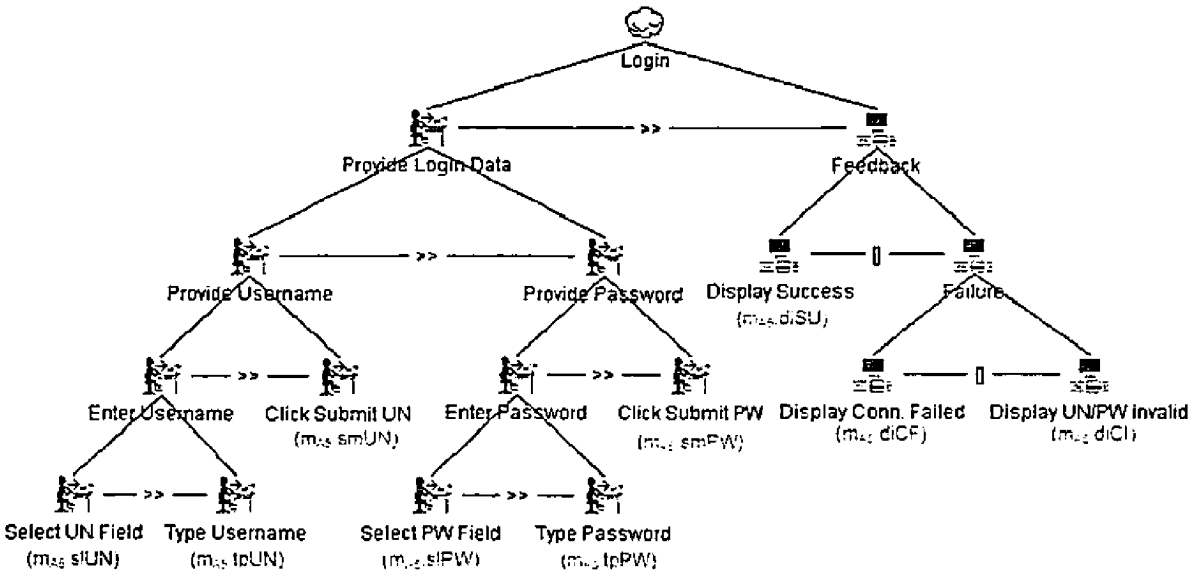


Figure 3.5. Design Level Task Model (m_{A5})

3.2.5 Use Case Model – (Design) Task Model Refinement

As indicated in Table 3.1 a use case or task model at the requirements level may be further refined by a task model at the design level. When moving from a use case model to a design-level task model we have to ensure that the task model adopts the entirety of the functional requirements specified in the use case and integrates them into the UI design stage. With the exception of internal system steps (see Section 3.2.2 for a detailed discussion), each use case step needs to be represented by a corresponding task in the refining task model. Moreover, similar to the task model - task model refinement presented in the previous section (3.2.4), the task model should remain "behaviorally equivalent" to the use case model and hence allow the same set of scenarios. That is, each scenario of the refined model is also a valid scenario of the refining model and vice versa.

In order to illustrate the refinement, let us consider the summary-level use case depicted in Figure 3.6. The use case captures the functional requirements for using an automated teller machine (ATM). After successful authentication the customer may repetitively choose and perform one of the main functionalities provided (*Withdraw Money*, *Print Account Statement*, *Make a Deposit*) or quit the system.

Use Case: Use Automated Teller Machine
Goal: Customer uses an ATM Machine.
Level: Summary
Primary Actor: Customer
Precondition: Customer is logged into the System.
Main scenario

1. Primary Actor provides authentication using his/her ATM card and PIN. ($m_{B1}.prAU$)
2. System authenticates the primary actor. ($m_{B1}.vaAU$)
3. System prompts the users to choose a desired functionality. ($m_{B1}.prCF$)
4. Primary Actor *chooses one of the following*:
   - 4a. Primary Actor selects to Withdraw Money. ($m_{B1}.slWM$)
   - 4b. Primary Actor selects to Print his/her Account Statement. ($m_{B1}.slPS$)
   - 4c. Primary Actor selects to Make a Deposit. ($m_{B1}.slMD$)
5. *The use case resumes at step 4.*

Extensions

2a. The primary actor could not be authenticated:
- 2a1. The system informs the primary actor that the authentication failed. ($m_{B1}.inAF$)
- 2a2. *The use case ends unsuccessfully.*

4a. Primary actor decides to cancel the use case:
- 4a1. Primary actor indicates that he/she wishes to cancel the use case. ($m_{B1}.slAB$)
- 4a2. *The use case ends.*

Figure 3.6. "Use Automated Teller Machine" Use Case (m_{B1})

Based on the use case the design-level task model depicted in Figure 3.7 has been developed. Clearly it is behaviorally equivalent to its base-use case. On the one hand, all the system choices (i.e. the choice between a success and failure login) are preserved. On the other hand, all user choices (selection between the three functionalities or to quit the system) are also present in the task model. In addition, step 1 (providing authentication) of the use case has been structurally refined into two separate tasks which distinguish between providing the ATM card and providing the PIN. The internal system step of authenticating the user is not present in the task model, as it is irrelevant for UI design.

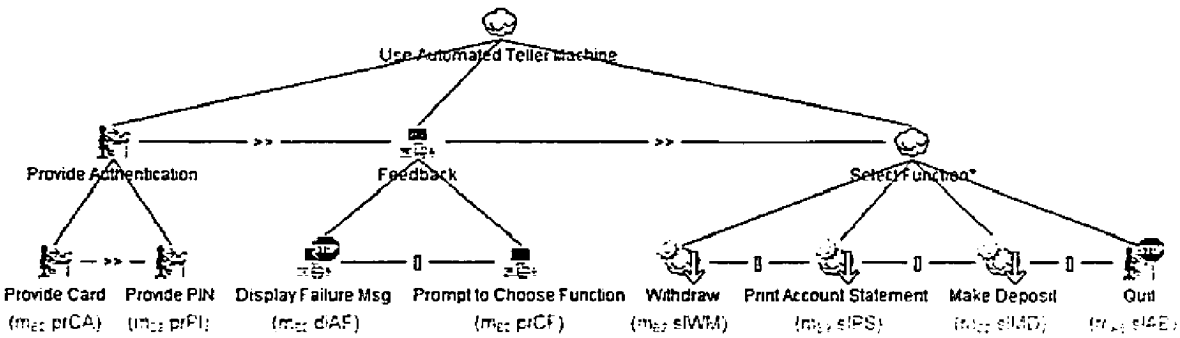


Figure 3.7. Design Level Task Model (m_{B2})

We conclude this refinement case by presenting a task model which is not a valid refinement of the ATM use case. As shown in Figure 3.8, the task model only allows the user to choose between withdrawing cash and making a deposit. The option of printing account statements is no longer supported (possibly because the UI is not equipped with a statement printing device). While the restriction of user choices would have been legitimate at the requirements level, it is an invalid refinement at the design level, as the requirements contract is not fulfilled by the design of the UI.

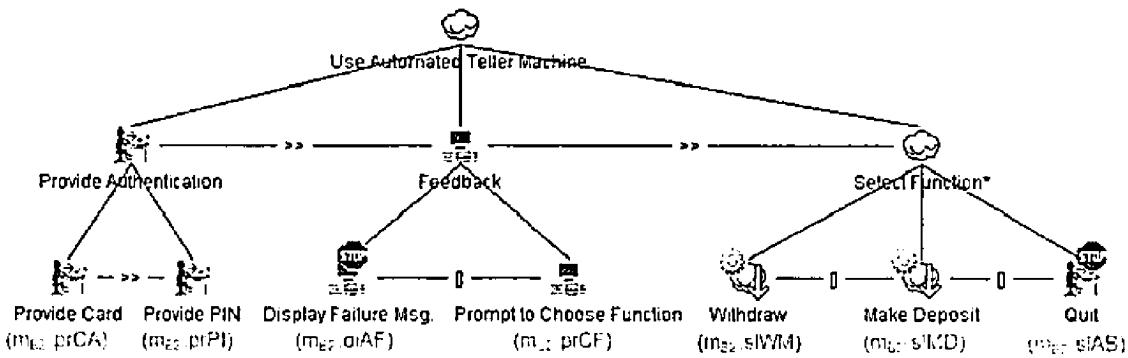


Figure 3.8. Invalid Design Level Task Model ($m_{B2'}$)

3.2.6 (Design) Task Model – (Design) Task Model Refinement

Within our methodology it is assumed that any task model is directly or indirectly based on an underlying use case model. Moreover, a design-level task model must implement all allowed behaviors of the corresponding requirements specification. Therefore, if, at the design stage, a task model is further refined by another task model, refinement may only take place at the structural but *not* at the behavioral level.

A valid refinement of the design-task model depicted in Figure 3.8 is given in Figure 3.9. The task model merely structurally refines the interaction tasks "Provide Card" and

"Provide PIN" by taking into account additional interaction details pertinent to a particular ATM. For example, in order to provide the ATM card it is required to first activate the screen and then to "slide" the card through the card reading device. The "Provide PIN" task has been further detailed with an additional "Submit" sub-task.

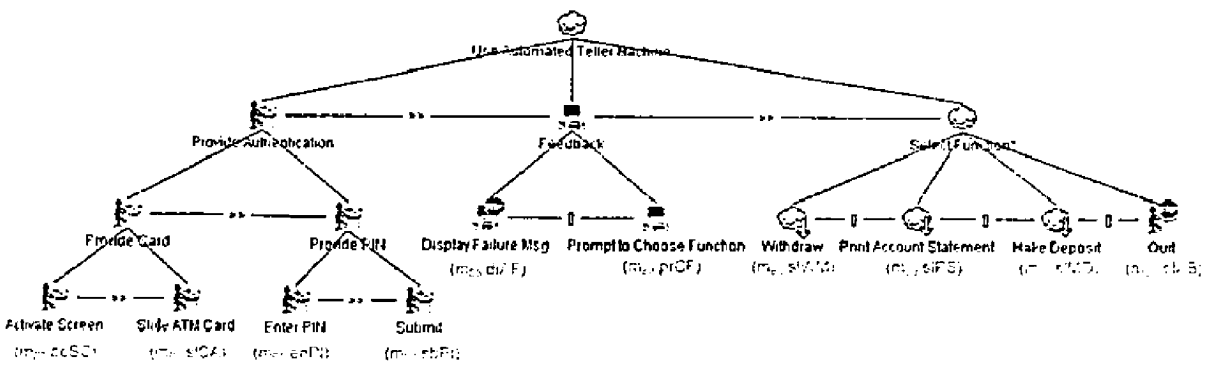


Figure 3.9. Design Level Task Model (m_{B3})

3.3 Summary

In this chapter we introduced an integrated development methodology for the use case and the task model. The methodology clarifies the application scope of each model, their relations and allowed modifications. As a key principle we established that the use case model documents functional requirements and the task models capture desired interactions of the user interface. While use case models are exclusively used at the requirements stage, task models may be used at the requirements and/or at the design stage. Both artifacts are developed iteratively through a series of refinements under the assumption that the development of the use case model precedes the development of the task model.

In order to ensure that the refined model is a valid refinement of its base model we discussed six different refinement relations for the use case and task model. Each refinement relation takes into account the nature of the involved artifacts as well as their purpose (requirements or design document) in the software development lifecycle. In the subsequent chapters, we will present a framework for defining a common formal semantics for use case and task models. By taking the formal semantics as a reference point, we formalize and automate the verification (by means of a tool) of the refinement relations.

4 Formal Framework for Use Case and Task Models

In this chapter we provide a general overview of our framework for formalizing use case and task models. We introduce a running example which will be used throughout the remainder of this thesis and define the syntactic portion of the framework. The semantics are presented in the subsequent chapters.

4.1 Overview

By formal framework we mean a structure of related concepts for formally representing and comparing use case and task models. The foundation of the framework is based on mathematical concepts and is independent of any software application and/or technology domain. We devised the framework to meet the following principal goals:

Formal Semantics: Primary goal of the framework is to define a common semantic base within which use case and task models can both be formally defined.

Refinement: The framework should include the facilities required to specify refinement relations between use case and task models.

Extensibility: The framework should not be tied to a specific use case or task modeling notation but should be applicable and extensible for a variety of notations and breeds.

Tools Support: The semantic framework should be amenable to tool support. This will partly serve as a means of validating the framework. Tools could, e.g., offer automated verification of refinement between models.

In what follows we introduce our formal framework by discussing how each of the aforementioned requirements are addressed. We start with the definition of formal semantics. Figure 4.1 illustrates how our framework promotes a two-step mapping from a particular use case or task model notation to the common semantic domain which is either based on *sets of partially ordered sets* (sets of posets) or *nondeterministic finite state machines* (nFSMs).

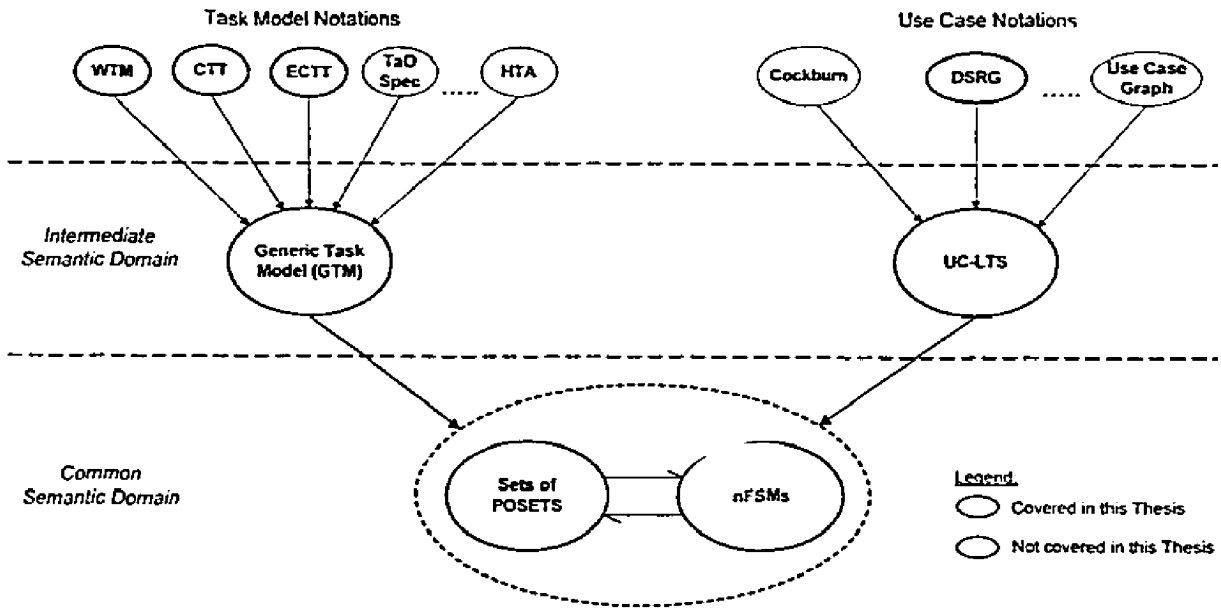


Figure 4.1. Two-Step Semantic Mapping

The main reason behind using a two-step mapping, rather than a direct mapping, is to provide a semantic framework that can be validated, reused and extended more easily. The intermediate semantic domains have been carefully chosen by taking into consideration the intrinsic characteristics of use case and task models, respectively. As a result, the mappings to the intermediate semantic domains are straightforward and intuitive: use cases are mapped to a so called Use Case Labeled Transition System (UC-LTS); task models are mapped into what we call a Generic Task Model (GTM). Since the second level mappings are more involved, the intermediate semantic domains have been chosen so as to be as simple as possible, containing only the necessary core constructs. As a consequence of this two-step semantic definition, we believe that our framework can be easily extended to incorporate new task model or use case notations by simply defining a new mapping to the intermediate semantic domain.

Both of the common semantic models will serve as a reference for the definition of a set of refinement relations between use case and task models. In this vein, we will introduce our tool called the *Use Case and Task Model Verifier*. It parses and type-checks GTM and UC-LTS specifications. Using the nFSM semantics as a reference point, it maps a given set of specifications to the corresponding nFSM representations and allows the developer to execute a suite of fully automated refinement checks.

4.2 Running Example

Throughout this thesis, we use the example of an "Invoice Management System" (IMS) to illustrate the introduced concepts. As depicted in the use case diagram of Figure 4.2, the system features the following user-goal level functionalities: "Order Product", "Cancel Existing Order", and "Review Orders". The use case "Order Product" requires interaction with a secondary actor; the payment authorization system.

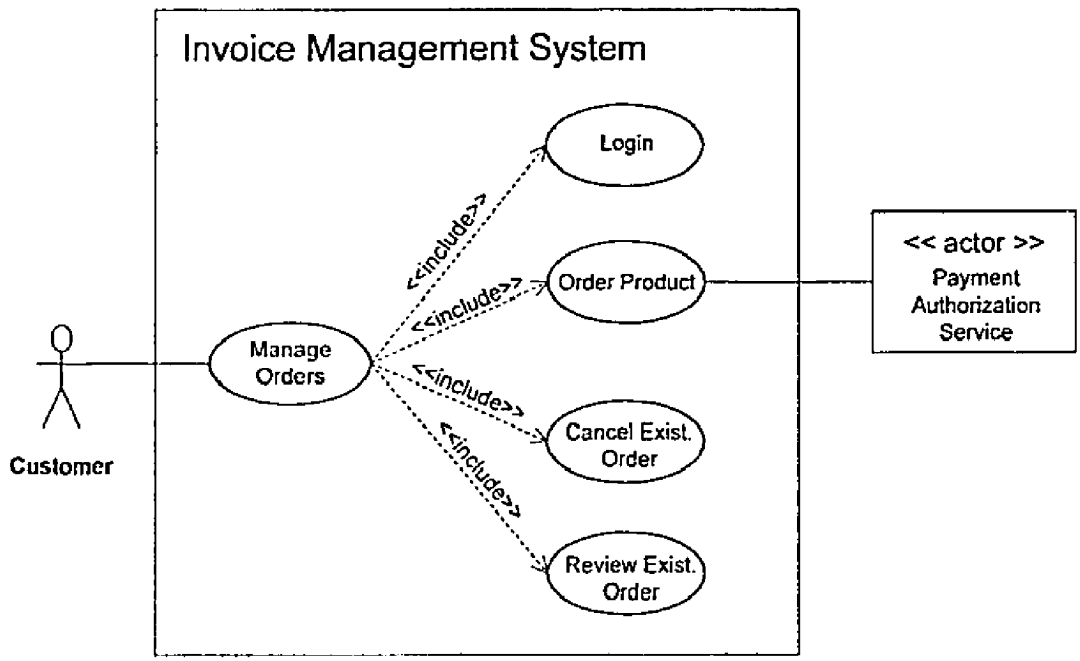


Figure 4.2. Use Case Diagram of "IMS" System

The four user-goal level use cases are invoked and temporally related by the summary-level use case "Manage Orders" (given in Figure 4.3). All functionalities require user authentication, i.e. successful termination of the sub-function use case "Login". (An unsuccessful termination of the "Login" use case is handled by *extension 1a* and leads to the termination of the overall use case.) As indicated by the *step 3* and *extensions 2a* each user-goal level use case may be invoked for an arbitrary number of times until the user decides to quit the application.

Use case: Manage Orders

Properties

Goal: Primary Actor manages his/her orders
Level: Summary
Primary Actor: Customer
Precondition: None

Main Success Scenario

1. Primary Actor logs into the Invoice Management System
2. Primary Actor *chooses one of the following*
   - Order a Product
   - Cancel an Existing Order
   - Review an Existing Order
3. *Use case resumes at step 2*

Extensions

1a. Login terminated unsuccessfully:

1a1. *Use case ends unsuccessfully*

2a. Primary actor decides to quit the Invoice Management System:

2a1. Primary actor indicates to quit the Invoice Management System.

2a2. *Use case ends successfully*

Figure 4.3. "Manage Order" Use Case

A subset of the domain model for the "IMS" is portrayed in Figure 4.4. It captures relevant concepts of the problem domain and their interrelations. The diagram shows that products and their availability (in terms of quantities) are recorded by a central inventory. Orders can only be placed for a single product. A customer however, can place multiple orders.

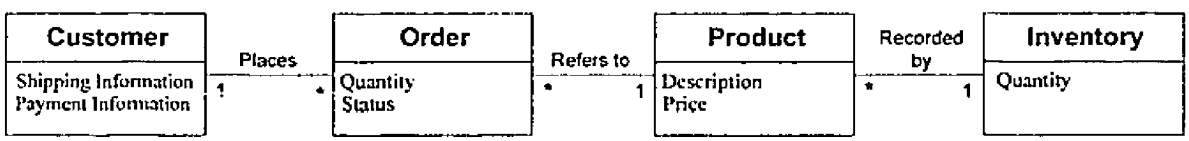


Figure 4.4. Domain Model for "Invoice Management System"

In the next section we provide the syntax for two particular use case and task model notations, namely DSRG-style use case models and Extended Concurrent Task Trees (ECTT). For both notations we provide motivational and illustrative examples which are taken from the "IMS" case study.

4.3 Syntax for Use Case Models

4.3.1 Motivation

Different notations for expressing use cases possessing different degrees of formality have been suggested. The extremes range from purely textual constructs written in prose [Cockburn 2001] to entirely formal specifications written in Z [Butler et al. 1993], as Abstract State Machines (ASM) [Grieskamp et al. 2001; Barnett et al. 2003], or as graph structures [Mizouni 2007]. While the use of prose makes use case modeling an attractive tool for facilitating communication among stakeholders, its informal nature makes it prone to ambiguities and thus leaves little room for tool support.

In this thesis, we adopt an intermediate solution which enforces a formal structure but also preserves the intuitive nature of use cases. I.e., we provide support for formalizing the sequencing of use case steps and their types, but the respective actions, as well as the associated conditions are specified informally. The property section of the use case, except for the discrete goal-level property is specified using narrative language.

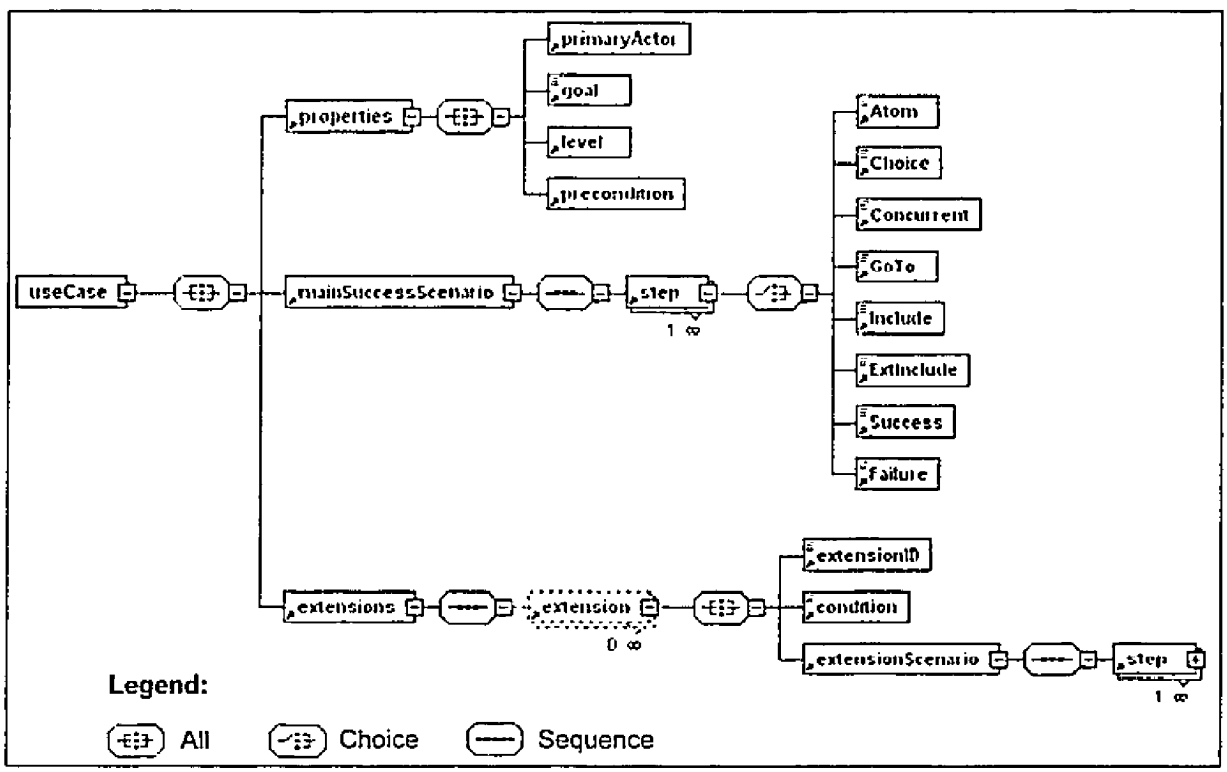


Figure 4.5. DSRG-style Use Case Structure

We define a *DSRG-style use case model* as a set of interrelated use cases, with one use case being the root use case. As shown in Figure 4.5 a use case is defined to have a property section, a main success scenario and a set of extensions. Central to the definition of the main success scenario and extensions is the concept of a *step*, which can be of eight different kinds (explained in Table 4.1). Additionally, an extension contains a condition that states under which circumstances the extension may occur. An arbitrary number of extensions may be associated with *atomic*, *choice* and *concurrent steps*. *Extended inclusions* have exactly one extension, which is taken, if the execution of the included use case terminates unsuccessfully. The remaining kinds of steps cannot be associated with an extension.

Table 4.1. DSRG Use Case Step Kinds

Use Case Step Kind	Description
Atom	Denotes an atomic use case step. Similar to CTT task models it contains type information which defines whether the step is carried out by the user or performed by the system. The latter case is further distinguished as: steps observable to the user and internal system actions (whose effects are invisible to the user). Each atomic use case step contains a *label*, which (using prose) defines an action performed by an actor or the system.
Choice	Denotes a complex use case step, which provides the primary actor with a list of alternative step sequences, among which one alternative must be chosen.
Concurrent	Denotes a set of primitive[2] use case steps, which may be performed in any order by the primary actor.
Goto	Branch to another step.
Include	Inclusion of a sub-use case.
ExtInclude	Extended inclusion of a sub-use case. If the included use case terminates unsuccessfully the associated extension will be taken (e.g. inclusion of "Login" use case in Figure 4.3).
Success	Successful termination of the scenario.
Failure	Unsuccessful termination of the scenario.

[2] A primitive use case step is an atomic step, which must not be associated with an extension or be the target of a *Goto* step.

An example of a DSRG-style use case is given in Table 4.2. The use case captures the interactions for the "Order Product" functionality of the "Invoice Management System". For the sake of brevity, only the main success scenario and the extensions are shown in full, the header section of the use case is partly omitted. The complete use case along with the remaining use cases of the "IMS" example can be found in Appendix G. The main success scenario of the use case describes the situation in which the primary actor directly accomplishes his/her goal of ordering a product. The extensions specify alternative scenarios which may (4a) or may not (4b, 5a, 7a, 8a) lead to the abandonment of the use case goal. For the sake of readability the use case is presented in a user friendly-narrative form[3]. However, in order to illustrate the relation to the formal DSRG syntax (discussed in the next section) each step is further attributed with information about its *kind*, *type* and a *label* which serves as a short-hand for the narrative action description.

[3] For this purpose we have developed an XSLT style sheet which is able to generate a readable representation from a DSRG-style use case model encoded in XML.

Table 4.2. Order Product Use Case (Narrative Style)

Use case: Order Product

Properties

Goal: Primary actor places an order for a specific product.
Level: User-goal
...

Main Success Scenario

1. Primary Actor indicates that he/she wishes to Order a Product. (Atom. interaction. 'trOP')
2. Primary Actor specifies the desired product category. (Atom. interaction. 'spCA')
3. System displays search results that match the Primary Actor's supplied criteria. (Atom. application. 'diSR')
4. Primary Actor selects a product and identifies the desired quantity. (Atom. interaction. 'slPQ')
5. System validates that the product is available in the requested quantity. (Atom. internal. 'vaPQ')
6. System displays the purchase summary. (Atom. application. 'diPS')
7. Primary Actor *chooses one of the following* (Choice)

   7A.1. Primary Actor elects to pay by credit card and submits account information. (Atom. interaction. 'paCC')

   OR

   7B.1 Primary Actor elects to pay by debit card and submits account information. (Atom. interaction. 'paDB')
   7B.2 The Primary Actor provides his/her PIN number. (Atom. interaction. 'prPI')
8. System interacts with the Payment authorization system to carry out the payment. (Atom. internal. 'vaPA')
9. System informs Primary Actor that order is confirmed. (Atom. application. 'inCO'
10. *Use case ends successfully* (Success)

Extensions

4a. Primary Actor is not satisfied with the search results:

4a1. Primary Actor indicates to do another product search. (Atom. interaction. 'inPS')

4a2. *Use case resumes at step 2.* (Goto)

4b. Primary Actor decides to cancel the use case:

4b1. Primary Actor indicates to cancel the use case. (Atom. interaction. 'inCA')

4b2. *Use case ends unsuccessfully.* (Failure)

5a. The desired product is not available in sufficient quantities:

5a1. System informs Primary Actor that product unavailable in desired quantity. (Atom. application.'inIQ')
5a2. *Use case ends unsuccessfully.* (Failure)

7a. Primary actor decides to cancel the use case:

7a1. Primary Actor indicates to cancel the use case. (Atom. interaction. 'inCA')

7a2. *Use case ends unsuccessfully.* (Failure)

8a. The payment was not authorized:

8a1. System informs Primary Actor that payment was not authorized. (Atom. application. 'inPF')
8a2. *Use case resumes at step 7.* (Goto)

4.3.2 Formal Definition of Abstract Syntax

Part of the formal theory presented in this thesis has been written in Isabelle/HOL [Nipkow et al. 2008]. An overview of the basic Isabelle theory constructs is given in Appendix B. Expressing parts of our formal system in Isabelle allows us to formally

prove key properties (such as well-formedness and closure properties), and hence, help validate the semantics. In this section we present a formalization of the syntax of DSRG-style use case models. We also provide a set of well-formedness rules and a formal statement of the "Order Product" use case introduced previously.

Table 4.3. DSRG-style Use Case Syntax Formalized in Isabelle

```
theory uc
imports Main
begin

(*For the sake of brevity the definitions for the following types are not shown.

  UCName, GoalProperty, ActorProperty, PreconditionProperty, StepID,
  ExtensionID, Label, Condition

  Each of these types is defined as a synonym for the type "string".
*)

datatype GoalLevelProperty = SUMMARY | USERGOAL | SUBFUNCTION
datatype StepType =  APPLICATION | INTERACTION | INTERNAL

record UCProperties = Goal :: GoalProperty
                      PrimaryActor :: ActorProperty
                      GoalLevel :: GoalLevelProperty
                      Precondition :: PreconditionProperty

types PrimStep = Label

datatype Step = Atom StepID StepType Label "ExtensionID set"|
                Choice StepID "(Step list) list"  "ExtensionID set" |
                Concurrent StepID "PrimStep set" "ExtensionID set" |
                Goto StepID StepID |
                Include StepID UCName |
                ExtInclude StepID UCName ExtensionID |
                Success StepID |
                Failure StepID

record Extension = ID :: ExtensionID
                   Condition :: Condition
                   ExtensionScenario :: "Step list"

record UseCase = Name :: UCName
                 Properties :: UCProperties
                 MainSuccessScenario :: "Step list"
                 Extensions :: "Extension set"
```

Table 4.3 depicts that, analogously to the informal definition discussed in the previous section, each use case is defined as a record consisting of a use case name, a set of properties, a main success scenario, and a set of extensions. The main success scenario consists of a list of use case steps among which we distinguish between eight different step kinds (datatype *Step*). A use case extension is defined as a record consisting of an identifier, a condition, and a list of use case steps. The latter denote an alternative flow, relative to the main success scenario (or denotes a super-ordinated extension).

[4] Instead of a 'list' it would be semantically more accurate to use a 'set'. However Isabelle/HOL does not support using 'sets' within recursively defined datatypes.

The building blocks of each use case are *atomic* use case steps, which define the actual interactions of the primary actor and corresponding system responses. We distinguish between three different step types: Steps of type *interaction* are performed by the primary actor, whereas steps of types *application* and *internal* are carried out by the system, with the difference that the former have an externally visible effect (to the primary actor) while the effects of the latter are invisible. Finally, we note that according to the informal definition of Table 4.1, a concurrent use case step denotes a set of primitive steps performed by the primary actor, which cannot be associated with any extension (or be the target of a *Goto* step). In the formal definition, such primitive steps are denoted by the *PrimStep* definition. Each *PrimStep* merely consists of a *label*. The type of a *PrimStep* is assumed to be *interaction* (see mapping to intermediate semantic domain in Chapter 5.1).

We can now define a DSRG-style use case model as a collection of use cases with one designated use case being the root use case. In what follows, for the sake of enhanced readability, we will express some of the definitions in mathematical notation, instead of using Isabelle/HOL syntax. The nomenclature for the use symbols is given in Appendix A.

Definition 4.1 (DSRG-style Use Case Model). A *DSRG-style Use Case Model* D is a tuple $D = (n_0, \mathcal{U})$ where,

$n_0 \in UCNAME$ is the name of the root use case.

$\mathcal{U} \in UCNAME \rightarrow USECASE$ is a map of use case definitions (with a finite domain) such that $n_0 \in dom(\mathcal{U})$. If $(n, uc) \in \mathcal{U}$ then we shall write $[n := uc]_{\mathcal{U}}$, sometimes omitting the subscript, when it is clear from the context.

In order to illustrate the definition of a DSRG-style use case, let us reconsider the previously depicted "Order a Product" use case. Table 4.4 portrays its formalization as an Isabelle constant of type $USECASE$ (defined in Table 4.3). Formalizations of the entire IMS use case model can be found in Appendix G.

Table 4.4. Formalized Syntax of "Order Product" Use Case

```
constdefs
OrderProductUC :: UseCase
 "OrderProductUC == (|
   Name = ''Order Product'',
   Properties = (|
    Goal = ''Primary actor places an order for a specific product'',
    PrimaryActor = ''Customer'',
    GoalLevel = USERGOAL,
    Precondition = ''Primary Actor is logged into the system.''
   |),
   MainSuccessScenario = [
      Atom ''OP.s1'' INTERACTION ''trOP'' {},
      Atom ''OP.s2'' INTERACTION ''spCA'' {},
      Atom ''OP.s3'' APPLICATION ''diSR'' {},
      Atom ''OP.s4'' INTERACTION ''slPQ'' {''OP.e1'', ''OP.e2''},
      Atom ''OP.s5'' INTERNAL    ''vaPQ'' {''OP.e3''},
      Atom ''OP.s6'' APPLICATION ''diPS'' {},
      Choice ''OP.s7'' [
       [Atom ''OP.s7A1'' INTERACTION ''paCC'' {} ],
       [
        Atom ''OP.s7B1'' INTERACTION ''paDB'' {},
        Atom ''OP.s7B2'' INTERACTION ''prPI'' {}
       ] ] {''OP.e4''},
      Atom ''OP.s8'' INTERNAL    ''vaPA'' {''OP.e5''},
      Atom ''OP.s9'' APPLICATION ''inCO'' {},
      Success ''OP.s10''
   ],
   Extensions = {
     (*Extension 4a*)
    (|
     ID = ''OP.e1'',
     Condition = ''Primary Actor is not satisfied with search results'',
     ExtensionScenario =
      [ Atom ''OP.s4a1'' INTERACTION ''inPS'' {},
        Goto ''OP.s4a2'' ''OP.s2'']
    |),
     (*Extension 4b*)
    (|
     ID = ''OP.e2'',
     Condition = ''Primary Actor decides to cancel the use case'',
     ExtensionScenario =
      [ Atom ''OP.s4b1'' INTERACTION ''inCA'' {},
        Failure ''OP.s4b2'']
    |),
     (*Extension 5a*)
    (|
     ID = ''OP.e3'',
     Condition = ''The product is unavailable in sufficient quantities'',
     ExtensionScenario =
      [ Atom ''OP.s5a1'' APPLICATION ''inIQ'' {},
        Failure ''OP.s5a2'' ]
    |),
     (*Extension 7a*)
    (|
     ID = ''OP.e4'',
     Condition = ''Primary Actor decides to cancel the use case'',
     ExtensionScenario =
      [ Atom ''OP.s7a1'' INTERACTION ''inCA'' {},
        Failure ''OP.s7a2'' ]
    |),
     (*Extension 8a*)
    (|
     ID = ''OP.e5'',
     Condition = ''The payment was not authorized'',
     ExtensionScenario =
      [ Atom ''OP.s8a1'' APPLICATION ''inPF'' {},
        Goto ''OP.s8a2'' ''OP.s7'' ]
    |)
   }
|)"
```

We conclude this section by defining a set of well-formedness conditions that need to be satisfied for any DSRG-style use case model.

Definition 4.2 (Well-formedness of an DSRG-style Use Case Model). A DSRG-style use case model $D = (n_0, \mathcal{U})$ is well-formed if and only if:

1. All use case steps and extension IDs are unique (within the scope of a use case model).
2. For every step $Goto\ (id, id_{jump})$ or extension reference (id_{ex}), there exists a corresponding use case step or use case extension within the same use case, respectively.
3. For every $Include(id, n)$ or $ExtInclude(id, n, id_{ex})$ we require that $n \in dom(\mathcal{U})$ and that there are no circular inclusions. Furthermore, there must not be circles consisting of only *internal* system steps.
4. The last element of every use case step sequence is either $Goto$, $Success$, or $Failure$.
5. Atomic use case steps of type *application* do not have extensions. Instead, the extension is to be associated with an *internal* system step, which precedes the (externally visible) *application* step.
6. The list of step sequences within a $Choice$ step contains at least one element.
7. The condition statements for extensions of $ExtInclude$ steps are empty.
8. Each defined step sequence within the main success scenario, an extension, or $Choice$ step contains at least one step.

4.4 Syntax for Task Models

In this section we define the syntax of the *ECTT* task modeling notation, an extended version of CTT [Paternò & Santoro 2001]. We start by motivating why such an extension of CTT is needed.

4.4.1 Motivation for ECTT

In [Sinnig et al. 2007] we reported that current task modeling notations provide insufficient support to serve as specifications for user interfaces of moderate to large sizes. The two main shortcomings are as follows:

- **Error and Failure Cases:** Exhaustive modeling of alternatives and error scenarios (as is done in use case modeling through the use of extensions) is indispensable to capturing a full behavioral specification of a user interface. The *current CTT operator set is not sufficient* to effectively describe task specifications that take into account failure and error cases. For example, in contrast to use-case modeling, CTT does not have an operator defining the premature abortion of a scenario (whether it is due to human or system error). Error handling with the traditional operator set results in an explosion of complexity which diminishes the readability of the task model [Bastide & Basnyat 2006].
- **Monolithic Structure of Task Model:** Traditionally task models capture the behavioral aspects of the user interface within a single monolithic task tree. A monolithic task tree is suitable for applications of small sizes but becomes unmanageable (in terms of visualization, comprehension and modification) for applications of even moderate size. Even the newest version of CTT only supports references to tasks within the same task tree. A modular construction of the task model out of sub-models is not possible.

In order to address the above-mentioned shortcomings we define ECTT as an extension to CTT. The extensions relative the CTT are two-fold. First, we introduce new temporal operators (*stop* and *resume*) allowing the specification and treatment of error cases. Second, similar to the <<include>> relationship in use case models, we define concepts in support of modularization which allow for the arbitrary inclusion sub-ordinate task definitions. In what follows, before formally defining the syntax of ECTT, we provide an intuitive definition of the newly introduced concepts.

Stop and Resume

Intuitively the unary operators *stop* and *resume*[5] denote the deactivation and reactivation of the respective operand task (see Table 4.5). As such their interplay is similar to the *throw* and corresponding *catch* of an exception of programming languages like Java. A task which "throws" a *stop* exception cannot enable any tasks. *stop* denotes an exceptional case, which, "untreated", leaves the super-ordinate task incomplete and thus

[5] The unary *resume* operator is not to be confused with the binary standard CTT operator *suspend / resume* (|>).

inevitably leads to the premature termination of a scenario. *Resume* is used to "catch" a *stop* exception and as such counteracts and limits the effects of *stop*. After *resume*, the execution of the affected task returns back to "normal"; i.e. its execution will enable respective subsequent tasks.

Table 4.5. Additional Operators of ECTT

Operator	Syntax	Interpretation
Stop	$stop(t)$	The execution of task t and consequently the execution of t's super-ordinate task tree cannot enable any task.
Resume	$resume(t)$	Counteracts the effect of *stop* such that the execution of t will enable subsequent tasks regardless whether t's definition entails *stop* or not.

Modular Set-up

Each ECTT task model consists of a set of atomic tasks and task definitions, where each task definition denotes a high-level task. High-level tasks are further decomposed into so-called task expressions, which are compositions of lower-level task definitions or task references. Task references denote the inclusion of already existing task definitions. In contrast to CTT (which only allows the inclusion of tasks within the same task-tree hierarchy, an ECTT task definition allows the inclusion of any task definition which belongs to the ECTT task model, regardless of whether it is part of the same task hierarchy or not.

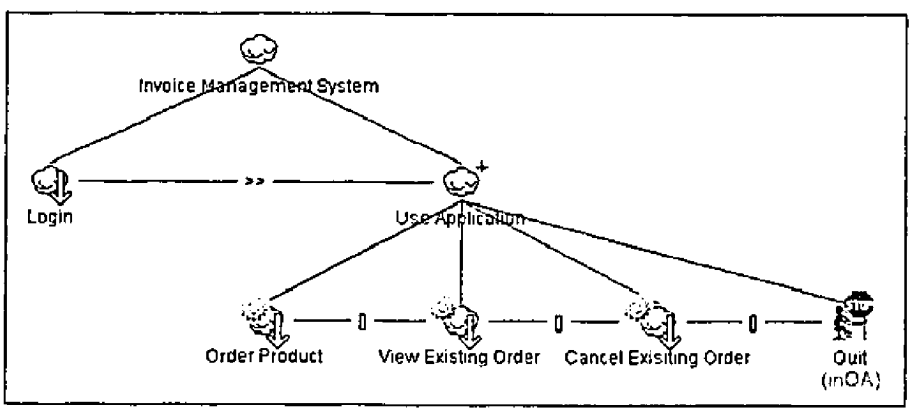


Figure 4.6. Root Task Definition of the Invoice Management System

In order to illustrate the new operators and concepts, let us consider the *requirements-level* task model of the "Invoice Management System". The task model has been developed for a user interface running on a kiosk machine. As visualized by Figure 4.6, only high-level tasks and their interrelation are shown. In particular, we can learn that "Login" is necessary to enable the application. Once logged in, the user may repetitively perform any of the following tasks "Order Product", "View Existing Order", "Cancel Existing Order", or "Quit". The first three tasks are attributed with the resume ($\gg$) operator, denoting that regardless of whether the respective task terminates successfully or is aborted prematurely the overall "Use Application" task remains enabled. In contrast to this, the "Quit" task is attributed with the *stop* operator denoting that after its execution the overall "Use Application" task may no longer be performed iteratively which inevitably leads to the termination of the scenario.

The "Order Product" task definition is visualized in Figure 4.7. (The task definitions "Login", "View Existing Order", and "Cancel Existing Order" are given in Appendix G) If we compare the "Order Product" task definition with the "Order Product" use case (Section 4.3) we note that the task specification has more UI details. For example use case *step 2* ("Primary Actor specifies the desired product category.") has been refined by two sequential tasks ("Select Category" and "Submit Criteria"). Moreover, the task model only implements a subset of the functionality of the use case. From a purely functionality point of view (specified in the use case) the system supports both payment by credit card and payment by debit card. The capabilities of the UI, however only allow the user to pay by credit card. According to the integrated development methodology (discussed in the previous chapter) this is a valid restriction at the requirements level. Finally it is noticeable that the task model does not specify a corresponding task for use case step 8 ("System interacts with the Payment authorization system to carry out the payment."). This step denotes an internal system interaction which is irrelevant for UI design.

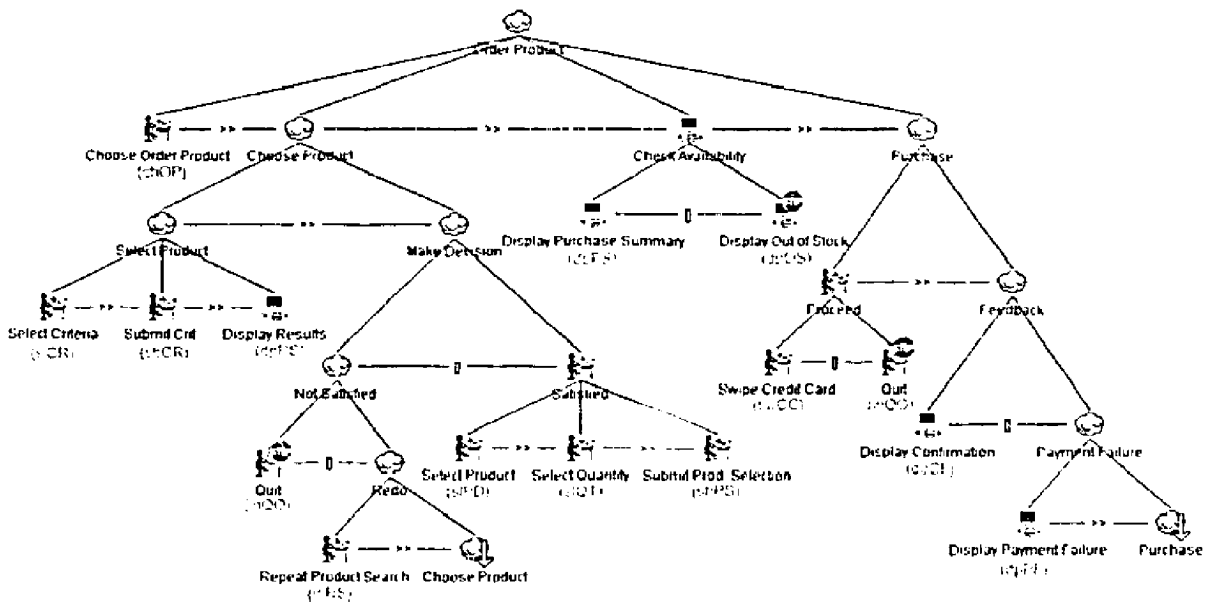


Figure 4.7. Visualization of the "Order Product" Task Definition

4.4.2 Formal Definition of Abstract Syntax

In the previous section, we motivated the need for ECTT and its new operator set. In this section, we define an ECTT abstract syntax. We start with the definition of an ECTT task model. The nomenclature for the used symbols is given in Appendix A.

Definition 4.3 (ECTT Task Model). An *ECTT Task Model* C is a triple $C = (n_0, \mathcal{D}, \tau)$ where,

$n_0 \in TASKNAME$ is the name of the main task definition of the ECTT task model,

$\mathcal{D} \in TASKNAME \rightarrow TASKEXPR$ is partial map of task definitions. Note:

- $n_0 \in dom(\mathcal{D})$.
- If $(n, o) \in \mathcal{D}$ then we shall write $[n := o]_{\mathcal{D}}$, sometimes omitting the subscript, when it is clear from the context.
- We say that a task name n denotes an **atomic task** if $n \notin dom(\mathcal{D})$ or a **task reference** if $n \in dom(\mathcal{D})$.

$\tau \in TASKNAME \rightarrow \{abstract, interaction, application\}$ is a typing function that associates a task type with each task name in C. More precisely, we define the domain of τ as follows:

$$dom(\tau) = dom(\mathcal{D}) \cup \{n \in TASKNAME \mid n \text{ occurs in } \mathcal{D}(n') \text{ for } n' \in dom(\mathcal{D})\}$$

Note that in contrast to CTT, the various task definitions ($\mathcal{D}$) need not be *connected* by some task-subtask hierarchy. This allows for a more modular setup, enabling the UI designer to work on multiple task hierarchies concurrently and eventually connect them using references. The creation of a single monolithic task tree (as required by CTT) is not necessary.

Each task definition consists of a task name and an ECTT task expression. The latter is defined as follows:

Definition 4.4 (ECTT Task Expression). Let v, φ be ECTT task expressions and n be a task name, then the set of all *ECTT Task Expressions*, $TASKEXPR$, is the smallest set closed under the following rules:

(1) $n \in TASKNAME$ is a task expression.

(2) $v \gg \varphi, v\ [\ ]\ \varphi, v \mid\mid\mid \varphi, v \boxplus \varphi, v[> \varphi, v \mid > \varphi, [v], v^*, v^+, stop(v), resume(v)$ are also ECTT task expressions.

In order to illustrate the before-mentioned definitions let us reconsider the "IMS" task model partially visualized by Figure 4.6 and Figure 4.7. The corresponding formalization as an ECTT task model is depicted in Table 4.6. For the sake of conciseness only a subset (i.e. the root task definition and the "Order Product" task definition) of the model is given. A complete formalization can be found in Appendix G. Leaf task names are abbreviated by the label displayed underneath the respective task symbols.

Table 4.6. Partial ECTT Formalization of the IMS Task Model

C_{IMS} = ("Invoice Management System", $\mathcal{D}, \tau$), with

$\mathcal{D}$ = {

"Invoice Management System" := "Login" ≫ ("Use Application")$^+$

"Login" := ⋯

"Use Application" := *resume* ("Order Product")[ ]
resume ("View Existing Order") [ ]
resume("Cancel Existing Order") [ ]
stop ("inQA")

"Order Product" := "chOP" ≫ "Choose Product" ≫ "Check Availability" ≫ "Purchase"

"Choose Product" := "Search Product" ≫ "Make Decision"

"Search Product" := "slCR" ≫ "sbCR" ≫ "dpRS"

"Make Decision" := "Not Satisfied"[ ] "Satisfied"

"Not Satisfied" := *stop*("inQO") [ ] "Redo"

"Redo" := "inRS" ≫ "Choose Product"

"Satisfied" := "slPD" ≫ "slQT" ≫ "sbPS"

"Check Availability" := "dpPS"[ ] *stop*("dpOS")

"Purchase" := "Proceed" ≫ "Feedback"

"Proceed" := "swCC" [ ] *stop*("inQO")

"Feedback" := "dpCF"[ ] "Payment Failure"

"Payment Failure" := "dpPF" ≫ "Purchase"

"View Existing Order" := ⋯

"Cancel Existing Order" := ⋯

$$\tau(t) = \begin{cases} abstract, & if\ t \in \{\text{"InvoiceManagementSystem", " Login", "Use Application", "View Existing Order", "Cancel Existing Order", "Order Product", "Choose Product", "Make Decision", "Not Satisfied", "Redo", "Purchase", "Feedback", Payment Failure, ...}\} \\ interaction, & if\ t \in \{\text{"inQA", "chOP", "slCR", "sbCR", "inQO", "inRS", "Satisfied", "slPD", "slQT", "sbPS", "Proceed", "swCC", ...}\} \\ application, & if\ t \in \{\text{"dpRS", "CheckAvailability", "dpPS", "dpOS", "dpCF", "dpPF", ...}\} \end{cases}$$

Similar to DSRG-style use case modes we also define a set of well-formedness rules for ECTT task models.

Definition 4.5 (Well-formedness of an ECTT Task Model). An ECTT Task Model $C = (n_0, \mathcal{D}, \tau)$ is well-formed, if and only if:

- C contains at most "tail recursive" task definitions $[n := o]$ of the forms defined in Table 4.7.
- The task type of atomic tasks is either *interaction* or *application*.
- Direct and indirect operands of |||, |>, [> are of type *interaction*.

In order to ensure that a recursive task reference is *tail* recursive, the usage of recursive references needs to be restricted. Table 4.7 depicts the supported forms of recursion. Every well-formed ECTT task model can be transformed into a behaviorally equivalent task model without tail recursion. This transformation is defined in the next section.

4.4.3 Removal of Tail Recursion

Since a well-formed task model contains at most a very restrictive form of tail-recursion, for which we have identified iterative forms, the process of eliminating task references is rather straightforward. Table 4.7 depicts the supported forms of recursion and their non-recursive counterparts. From a modeling point of view template (2b) is most interesting. It represents the case in which recursion is used to implement a form of conditional iteration. Task n is only executed iteratively if subtask φ is performed. If, instead of φ, subtask v is performed n ends without starting another iteration. The templates (1) and (2a) are of less practical relevance as the task modeler probably would chose the non-iterative versions in the first place.

Table 4.7. Supported Forms of Tail Recursive Task Definitions

Category	Expression Template	Non-recursive Equivalent	Description
Enabling	(1) $n := [v] \gg n$	v^*	A recursive reference may be used on the right hand side of the *enabling* operator, if the enabling task (left hand side) is defined as *optional* and does not contain a recursive reference. In this case the expression can be simplified to the iterative execution of the enabling task.
Choice	(2a) $n := v \,[\,]\, n$	v	A recursive reference may be used (2a) on the left and right hand side of the *choice* operator. Furthermore (2b) it may be used as a subtask of the left or

Category	Expression Template	Non-recursive Equivalent	Description
	(2b) $n := v \,[\,]\, (\varphi \gg n)$	$\varphi^* \gg v$	right operand if it is used on the right hand side of an *enabling* operator. In the former case, the expression can be simplified to be the non-recursive part of the choice, whereas in the latter case it can be rewritten to the sequential composition of the iteration of the enabling task and the non-recursive operand of the *choice* operator.
Suspend / Resume	$n := \varphi \mid > v$, where φ is one of the form (1), (2a), (2b).	$\varphi' \mid > v$, where φ' is the non-recursive form of φ.	A recursive call may be used within the left operand of the *suspend/resume* operator if the operand is an *enabling* or *choice* task of the form described above. The equivalent non-recursive form depends on the form of the first operand.

Recursive references must not be used as operands of the *order independency* ($\boxplus$) and *concurrency* operator ($|||$). The former denotes the choice between the top-level interleavings of its operands. As such the term $n := v \boxplus n$ is semantically equivalent to $n := v \gg n[\,]\, n \gg v$. Clearly, n is not tail recursive and hence does not correspond to a supported form of recursion involving the choice operator (Table 4.7). The latter denotes the choice of the deep interleavings of its operands and similar to the *order independency* operator leads to a form of recursion which is not tail recursive. As for the *disabling* operator ($[>$) a recursive reference may not be used as the first or second operand. The unfolding of the former leads to a term of form $n := n$ (which clearly is not meaningful) whereas the unfolding of the latter leads to a term of form $n := n \gg v$, which is clearly not tail recursive. The usage of recursion with the operators *iteration* ($n := n^*$), *proper iteration* ($n := n^+$), *optional* ($n := [n]$), *stop* ($n := stop(n)$), and *resume* ($n := resume(n)$) was also not considered as the expressions are clearly not meaningful.

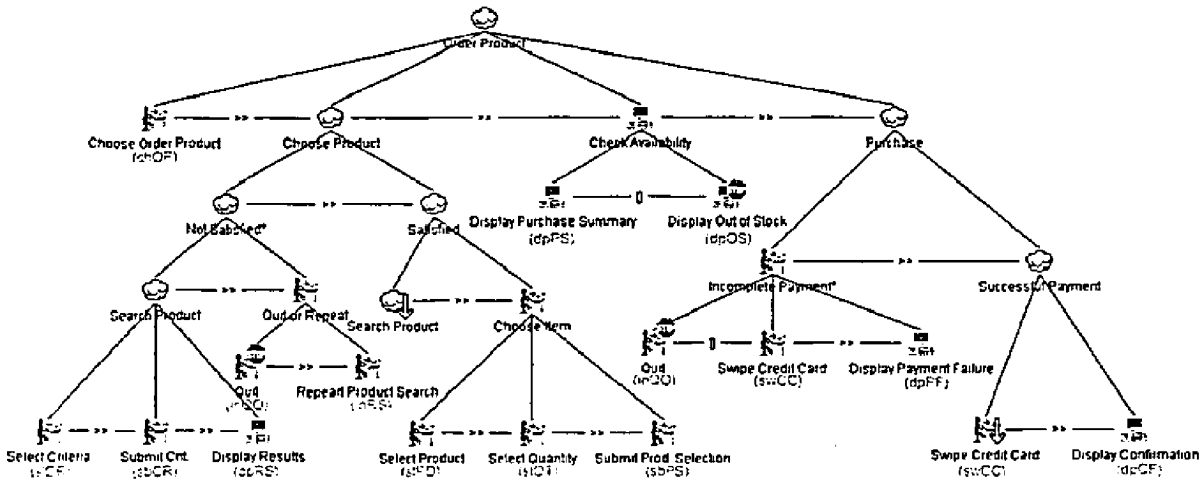


Figure 4.8. Visualization of "Order Product" Task Definition without Recursion

In what follows we find an equivalent ECTT task definition for the (tail) recursively defined "Order Product" task definition visualized in Figure 4.7 and formalized in Table 4.6. We first restructure the recursive task definition such that it corresponds to one of the supported forms of recursion. In this case the restructuring is done by expanding the choice constructs that decided between non-recursive and recursive behaviors. Once the task definition corresponds to a supported form we can apply the simplification rules of Table 4.7. The resulting non-recursive definition is visualized by Figure 4.8. A formalization is given in Table 4.8.

Table 4.8. Formalization of "Order Product" Task Definitions without Recursion

```
"Order Product" := "chOP" ≫ "Choose Product"  ≫ "Check Availability" ≫ "Purchase"
  "Choose Product" := ("Not Satisfied")*  ≫ "Satisfied"
     "Not Satisfied" := "Search Product"  ≫ "Quit or Repeat"
        "Search Product" := "slCR" ≫ "sbCR" ≫ "dpRS"
        "Quit or Repeat" := stop("inQO") [ ] "inRS"
     "Satisfied" := ("Search Product"  ≫ "Choose Item")
        "Choose Item" := "slPD" ≫ "slQT" ≫ "sbPS"
  "Check Availability" := "dpPS" [ ] stop("dpOS")
  "Purchase" := ("Incomplete Payment")*  ≫ "Successful Payment"
     "Incomplete Payment" := stop("inQO")[ ] ("swCC" ≫ "dpPF")
     "Successful Payment" := "swCC" ≫ "dpCF"
```

5 Intermediate Semantic Domains

In this chapter we introduce *Use Case Labeled Transition Systems* (UC-LTSs) and *Generic Task Models* (GTMs) as intermediate semantics domains for use case and task models, respectively. We also formally define the mappings from DSRG-style use case models and ECTT task models to their respective intermediate semantic domains.

5.1 Intermediate Semantic Domain for Use Case Models

The intermediate semantic domain for use case models is UC-LTSs. Its definition is similar to the definition of an ordinary LTS [Bacten & Weijland 1990] with the exception that transitions are associated with sets of labels rather than single labels. Additionally, each label is mapped to a unique type.

Definition 5.1. (Use Case Labeled Transition System). A *Use Case Labeled Transition System* (UC-LTS) is a tuple $U = (\Sigma, Q, q_0, F, \delta, \tau)$, where

$\Sigma \in \mathbb{P}\, EVENTNAME$ is the set of labels representing atomic use case steps,

$Q \in \mathbb{P}\, STATE$ is a set of states,

$q_0 \in Q$ is the initial state,

$F \subseteq Q$ is the set of final states,

$\delta: (Q \times \mathbb{P}_1(\Sigma)) \longrightarrow \mathbb{P}(Q)$ is the (total) transition function, and

$\tau: \Sigma \longrightarrow \{interaction, appliction, internal\}$ is a total typing function.

We believe that UC-LTS has been defined in a manner which easily and intuitively captures the nature of use cases, as we explain next. A use case primarily describes the possible execution order of user and system actions in the form of use case steps: from a given state, the execution of a step leads into another state. Accordingly, in UC-LTSs, the execution of a step (or set of steps, as we shall explain shortly) is denoted by a transition from a source state to a target state. Each transition is associated with a non-empty set of labels, where each label represents an atomic or primitive use case step. The execution order of use case steps is modeled using transition sequences, where the target state of a transition serves as the source state of the following transition. For a given transition, if

the associated label set contains more than one label, then no specific execution order exists between the corresponding primitive use case steps. I.e., a transition is triggered when all associated primitive steps are executed, the execution order, however, is arbitrary.

The mapping from a DRSG-style use case model to a UC-LTS is defined in two steps:

1. **Generation:** For each use case of the DSRG use case model, the main success scenario and extensions are mapped to UC-LTSs. Each such UC-LTS is a partial description of the respective use case, i.e., it represents either the main success scenario or an extension. Throughout generation, a global equivalence relation (~) is successively populated, which identifies equivalent states among the various UC-LTSs.

2. **Merging:** The various UC-LTSs are merged into a single UC-LTS. The merge is performed on the basis of the global equivalence relation (~).

Next, we describe the generation and merging processes in detail.

5.1.1 Generation of UC-LTSs

Given a DSRG-style use case model, we define the generation of a set of UC-LTSs in a bottom-up manner consisting of the following four steps:

Step 1: Map each individual use case step to a UC-LTS.

Step 2: Map use case step sequences to a UC-LTS.

Step 3: Map a DSRG-style use case to a set of UC-LTSs.

Step 4: Map a set of DSRG-style use cases to a set of UC-LTSs.

Definitions of the mappings require: (1) An input DSRG-style use case model in **canonical form**, (2) proper initialization of a **global environment** (*env*) and (3) a **global equivalence relation** (~) defined on the set of all states (*STATE*). In what follows details of each requirement will be given.

Canonical Form of a DSRG-style Use Case Model:

Definition 5.2. (Canonical Form of a DSRG-style Use Case Model). A DSRG-style use case model is in canonical form, if it is well-formed and if the following conditions are satisfied:

1. Each use case extension is associated with exactly one step.

2. Each use case (except for the root use case) is invoked by exactly one *Include* or *ExtInclude* step.

A well-formed use case model can be transformed into canonical form in a straight-forward manner. In order to satisfy condition (1), instead of the original extension, use case steps are associated with distinct copies of the respective extensions. If steps of the original extension are referenced by means of a *Goto* step, the respective reference is to be updated accordingly. Similarly, in order to satisfy condition (2) instead of the original sub-use case n, each *Include* or *ExtInclude* step is associated with a distinct copy (n') of n. For example, if, throughout the use case model, use case n is included three times by steps $Include(id_1, n)$, $Include(id_2, n)$ and $Include(id_3, n)$, then we create three distinct copies of n (n', n'', n''') and modify the inclusion steps as follows: $Include(id_1, n')$, $Include(id_2, n'')$ and $Include(id_3, n''')$.

Proper initialization of a global environment:

We also require the proper initialization of a global environment, *env*. As defined by Table 5.1, *env* has three fields, named *uc*, *ext* and *step*, where:

- *uc* is a function that maps use case names to $UCStateInfo$ which, according Table 5.2, defines for each use case the initial state (q_0) of the UC-LTS representing the main success scenario and the set of success (F_S) and failure final states (F_F) of all the UC-LTSs representing the respective use case. We note that the sets F_S and F_F correspond to the set of initial states of all the UC-LTSs representing *Success* steps in the given use case or *Failure* steps, respectively.
- *step* and *ext* are bijective functions that map a given step id or extension id to the initial state of the UC-LTS representing the use case step and use case extension,

respectively. Recall that step and extension ids are unique within any given use case model.

Table 5.1. Definition of Global Environment *env* in Isabelle

```
(* The global environment contains information required for the
   mapping to UC-LTS *)
record Environment =

  (* Mapping from a UCName to the corresponding UCStateInfo record*)
   uc :: "UCName => UCStateInfo"

  (* Mapping that associates each extension with the initial
     state of the corresponding UC-LTS*)
   ext :: "ExtensionID => STATE"

  (* Mapping that associates each step with the initial state of the
     corresponding UC-LTS*)
   step :: "StepID => STATE"
```

The following examples illustrate how entries in the global environment are accessed. The statement

$$q = env.uc(n).q_0$$

equates q with the initial state of the main success scenario representing the use case with name n. The statement

$$q = env.step(id)$$

equates q with the initial state of the UC-LTS representing the use case step whose id is id. For the following mappings we assume that env is globally available and is *read-only*—i.e., it is appropriately initialized with unique states prior to starting the mapping process.

Table 5.2. Definition of *UCStateInfo* in Isabelle

```
(* The record contains information specific to a particular use case *)
record UCStateInfo =

  (* Initial state of UC-LTS representing the main success scenario *)
   q0 :: "STATE"

  (* Set of initial states of the UC-LTSs representing Success steps
     in the respective use case *)
   Fs :: "STATE set"

  (* Set of initial states of the UC-LTSs representing Failure steps
     in the respective use case *)
   Ff :: "STATE set"
```

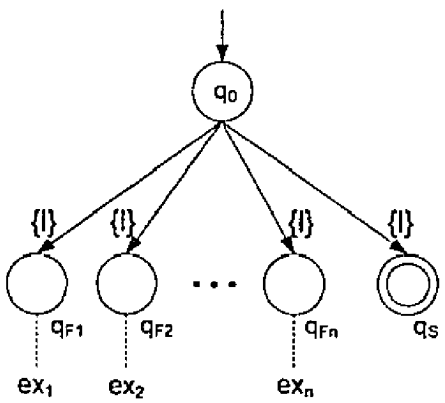
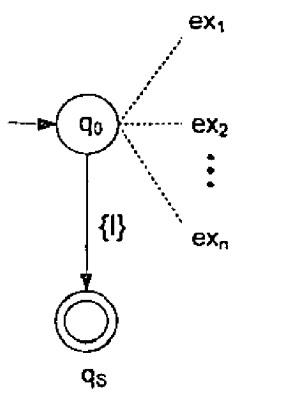
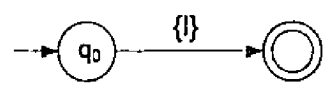
Global equivalence relation:

$\sim \subseteq STATE \times STATE$ is an equivalence relation (reflexive, symmetric, and transitive) defined over $STATE$, the set of all states. During merging, all equivalent states will be merged to a single state denoting its respective equivalence class. In order to satisfy the reflexivity requirement, $\sim$ is initialized as follows: $\sim = \{(q, q) \mid q \in STATE\}$. In what follows we describe each of the mapping steps in detail. Note that during the mapping process, any newly created state is unique.

Step 1: Map each individual use case step to a UC-LTS:

As defined in the abstract DSRG-style use case syntax (Table 4.3), there are 8 kinds of use case steps (*Atom*, *Choice*, *Concurrent*, *Goto*, *Include*, *ExtInclude*, *Success* and *Failure*). Each step kind has its own specific mapping to a UC-LTS. We start with an atomic use case step.

Table 5.3. UC-LTSs Representing Atomic Use Case Steps

(a) Internal	(b) Interaction	(c) Application
		

Definition 5.3 (Mapping an Atomic Step to a UC-LTS). An atomic use case step is mapped to a UC-LTS as follows:

$$\mathcal{M}_{StepUclts}[\![Atom(id, t, l, \{id_{ex_1}, id_{ex_2}, \ldots, id_{ex_n}\})]\!] = U$$

Case: $t = internal$

$$U = \left(\{l\}, \{q_0, q_S, q_{F_1}, q_{F_2}, \ldots, q_{F_n}\}, q_0, \{q_s\}, \delta_{int}, \{l \mapsto t\}\right) \text{ with}$$

$$q_0 = env.step(id)$$

$$\delta_{int} = \{(q_0, \{l\}) \mapsto \{q_S, q_{F_1}, q_{F_2}, \dots, q_{F_n}\}\}$$

$$\sim := \left(\sim \cup \left\{\left(q_{F_1}, q_{0_{ex_1}}\right), \left(q_{F_2}, q_{0_{ex_2}}\right), \dots, \left(q_{F_n}, q_{0_{ex_n}}\right)\right\}\right)^{*6} \text{ with}$$

$$q_{0_{ex_i}} = env.ext(id_{ex_i}, id)$$

Case: $t = interaction$

$$U = \left(\{l\}, \{q_0, q_S\}, q_0, \{q_S\}, \{(q_0, \{l\}) \mapsto \{q_S\}\}, \{l \mapsto t\}\right) \text{ with}$$

$$q_0 = env.step(id)$$

$$\sim := \left(\sim \cup \left\{\left(q_0, q_{0_{ex_1}}\right), \left(q_0, q_{0_{ex_2}}\right), \dots, \left(q_0, q_{0_{ex_n}}\right)\right\}\right)^{*} \text{ with}$$

$$q_{0_{ex_i}} = env.ext(id_{ex_i}, id)$$

Case: $t = application$

$$U = \left(\{l\}, \{q_0, q_S\}, q_0, \{q_S\}, \{(q_0, \{l\}) \mapsto \{q_S\}\}, \{l \mapsto t\}\right) \text{ with}$$

$$q_0 = env.step(id)$$

Depending on the step type (denoted by t) atomic steps are mapped to different UC-LTSs. The rationale behind each case is as follows:

- **Internal:** Each *internal* use case step has $n + 1$ different outcomes, among which one is captured in the main success scenario and the remaining $n \geq 0$ outcomes are captured by the corresponding extensions. From the user's point of view the decision of which outcomes is "selected" by the system happens nondeterministically (see Section 3). In order to preserve this nondeterminism, the resulting UC-LTS consists of $n + 1$ nondeterministic transitions (Table 5.3(a)); one transition for the main success scenario and n transitions for each defined extension. The former results in a final success state, which will be used for the sequential composition of use case steps. The latter result in a set of non-final states. During *merging*, these states will be joined

[6] * denotes the reflexive transitive closure

with the initial states of the UC-LTSs representing the various extensions. This is defined by adding the respective state pairs to the global equivalence relation (~).

- **Interaction:** In contrast to *internal* use case steps, which are performed by the systems and are hidden from the user, steps of type *interaction* are performed by the user. As such, they do not have an alternative outcome per se, but may be associated (by virtue of one or more extensions) with alternative steps which are performed instead of the actual step. As a result, the corresponding UC-LTS consists of only one transition (from q_0 to q_s), representing the use case step (Table 5.3(b)). Alternative steps are captured in the UC-LTSs representing the corresponding extensions. During *merging* the initial states of each UC-LTS representing an extension are identified with q_0. This is defined by updating ~ accordingly.

- **Application:** Steps of type *application* are performed by the system and have an externally visible effect to the user. As stipulated by the well-formedness rules in Definition 4.2, no extensions can be defined for steps of type *application*. As a result, the corresponding UC-LTS consists of only one transition (Figure 5.1(c)) and ~ remains unchanged.

In each case, regardless of the step type, the initial state of the resulting UC-LTS corresponds to the state defined in the global environment for the given step id ($env.step(id)$).

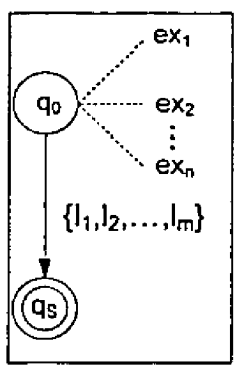


Figure 5.1. UC-LTS of a Concurrent Use Case Step

Definition 5.4 (Mapping of a Concurrent Step to UC-LTS). A concurrent use case step is mapped to a UC-LTS as follows:

$\mathcal{M}_{StepUclts}[\![Concurrent\ (id, \{l_1, l_2, \ldots, l_m\}, \{id_{ex_1}, id_{ex_2}, \ldots, id_{ex_n}\})]\!] = U$ with

$$U = (\{l_1, l_2, \ldots, l_m\}, \{q_0, q_S\}, q_0, \{q_S\}, \{(q_0, \{l_1, l_2, \ldots, l_m\}) \mapsto \{q_S\}\}, \tau) \text{ with}$$

$$q_0 = env.step(id)$$

$$\tau = \bigcup_{i=1}^{m} \{l_i \mapsto interaction\}$$

$$\sim := \left(\sim \cup \left\{\left(q_0, q_{0_{ex_1}}\right), \left(q_0, q_{0_{ex_2}}\right), \ldots, \left(q_0, q_{0_{ex_n}}\right)\right\}\right)^* \text{ with}$$

$$q_{0_{ex_i}} = env.ext(id_{ex_i}, id)$$

By definition, a concurrent use case step consists of a set of primitive use case steps which may be performed in any order by the primary actor. Similarly to atomic steps of type *interaction*, extensions to the concurrent step specify alternative behavior. As depicted in Figure 5.1, the corresponding UC-LTS consists of a single transition labeled with the set of labels $\{l_1, l_2, \ldots, l_m\}$, where each represents a primitive use case step (recall that, by definition, a primitive step cannot have extensions). Since the primitive use case steps are performed by the primary actor, each representing label is associated with type *interaction* (by τ).

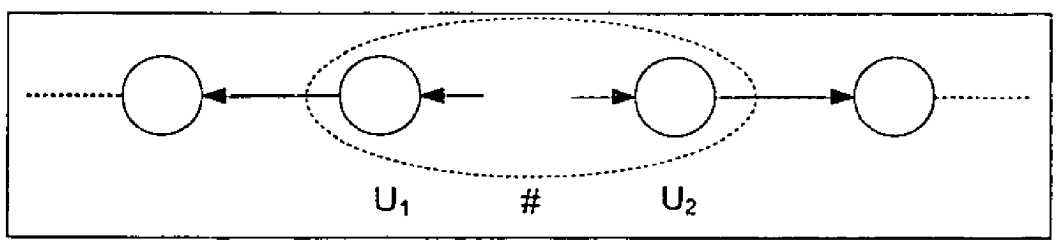


Figure 5.2. (Binary) Choice Composition of UC-LTSs U_1 and U_2

Definition 5.5 (Mapping a Choice Step to UC-LTS). A choice use case step is mapped to a UC-LTS as follows:

$\mathcal{M}_{StepUclts}[\![Choice\ (id, \langle s_1, s_2, \ldots s_m \rangle, \{id_{ex_1}, id_{ex_2}, \ldots, id_{ex_n}\})]\!] = U$ with

$$U = \mathcal{M}_{StepseqUcmsUts}[\![s_1]\!] \# \mathcal{M}_{StepseqUcsUts}[\![s_2]\!] \# \ldots \# \mathcal{M}_{StepseqUcsUts}[\![s_m]\!]$$

$$\sim := \left(\sim \cup \left\{\left(q_0, q_{0_{ex_1}}\right), \left(q_0, q_{0_{ex_2}}\right), \ldots, \left(q_0, q_{0_{ex_n}}\right)\right\}\right)^* \text{ with}$$

$$q_{0_{ex_i}} = env.ext(id_{ex_i}, id)$$

A choice step is a complex use case step which provides the primary actor with a list of alternative step sequences to choose from. The corresponding UC-LTS is the result of the choice composition (#) of the UC-LTSs representing the step sequences s_1 to s_m. (The meaning of $\mathcal{M}_{StepseqUcUts}$ will be explained in the next section.) Figure 5.2 schematically portrays that the binary choice composition corresponds to the merge of the initial states of the involved operand UC-LTSs, which can be generalized to the composition of $m > 2$ UC-LTSs in a straightforward manner. $\sim$ is updated by adding the state equivalences specific to the choice step and the corresponding extensions. A formal definitions of (#) is given next.

Definition 5.6 (Binary Choice Composition of UC-LTSs). Given two UC-LTSs $U_1 = (\Sigma_1, Q_1, q_{0_1}, F_1, \delta_1, \tau_1)$ and $U_2 = (\Sigma_2, Q_2, q_{0_2}, F_2, \delta_2, \tau_2)$ such that τ_1 and τ_2 are non-conflicting and Q_1 and Q_2 are disjoint, we define the *choice composition* (#) as follows: $U_1 \# U_2 = U$ with

$U = (\Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2 \cup \{q_0\}, q_0, F_1 \cup F_2, \delta, \tau_1 \cup \tau_2)$ where δ is defined as:

$$\delta(q,w) = \begin{cases} \delta_1(q,w), & if\ q \in (Q_1 \setminus \{q_{0_1}\}) \\ \delta_2(q,w), & if\ q \in (Q_2 \setminus \{q_{0_2}\}) \\ \delta_1(q_{0_1},w) \cup \delta_2(q_{0_2},w), & if\ q = q_0 \end{cases}$$

where $(q,w) \in dom\ \delta = (Q_1 \cup Q_2 \cup \{q_0\}) \times \mathbb{P}_1(\Sigma_1 \cup \Sigma_2)$.

Definition 5.7 (Mapping Goto, Include, ExtInclude Steps to UC-LTS). *Goto*, *Include* and *ExtInclude* are mapped into UC-LTS as follows.

$\mathcal{M}_{StepUcsUts}[\![Goto(id, id_{jump})]\!] = U$ with

$U = (\emptyset, \{q_0\}, q_0, \emptyset, \emptyset, \emptyset)$, with $q_0 = env.step(id)$

$$\sim := \left(\sim \cup \left\{\left(q_0, env.step(id_{jump})\right)\right\}\right)^*$$

$\mathcal{M}_{StepUcslts}[\![Include(id, n_{incl})]\!] = U$ with

$$U = (\emptyset, \{q_0, q_s\}, q_0, \{q_s\}, \emptyset, \emptyset) \text{ with } q_0 = env.step(id)$$

$$\sim := (\sim \cup \{(q_0, env.uc(n_{incl}, id).q_0)\} \cup \{(q_s, q) \mid q \in (env.uc(n_{incl}).F_S \cup env.uc(n_{incl}).F_F)\})^*$$

$\mathcal{M}_{StepUcslts}[\![ExtInclude(id, n_{incl}, id_{ex})]\!] = U$ with

$$U = (\emptyset, \{q_0, q_S, q_F\}, q_0, \{q_S\}, \emptyset, \emptyset) \text{ with } q_0 = env.step(id)$$

$$\sim := (\sim \cup \{(q_0, env.uc(n_{incl}).q_0)\} \cup \{(q_S, q) \mid q \in env.uc(n_{incl}).F_S\} \cup \{(q_F, q) \mid q \in env.uc(n_{incl}).F_F\})^*$$

Goto steps may be used at the end of a step sequence (e.g. the main success scenario) and denote a branching to the use case step with id = id_{jump}. The corresponding UC-LTS consists of a single state, which is defined equivalent (by means of ~) with the initial state of the UC-LTS representing the target use case step ($env.step(id_{jump})$).

Include and *ExtInclude* denote the invocation of a sub-use case (n_{incl}). For an *Include* step, regardless of whether the sub-use case terminates successfully or not, the overall *Include* step finishes successfully. The corresponding UC-LTS consists of two states (q_0 and q_s) which are not (yet) connected by any transition. During *merging* the initial state ($env.uc(n_{incl}).q_0$) of the UC-LTS representing the main success scenario of n_{incl} and all final states ($env.uc(n_{incl}).F_S \cup env.uc(n_{incl}).F_F$) of the UC-LTSs representing the sub-use case will be merged with q_0 and q_s, respectively.

For *ExtInclude* steps a distinction is made depending on whether the included use case terminates successfully or unsuccessfully. The corresponding UC-LTS consists of three states (q_0, q_S and q_F), which are not (yet) connected by any transition. During *merging*, the initial state ($env.uc(n_{incl}).q_0$) of the UC-LTS representing the main success scenario of the included use case is merged with q_0. The final states denoting a successful ($env.uc(n_{incl}).F_S$) and unsuccessful outcome ($env.uc(n_{incl}).F_F$) are marked for merging (by means of ~) with q_S and q_F, respectively. Additionally, the initial state

($\{env.ext(id_{ex})\}$) of the UC-LTS representing the extension, responsible for "catching" the unsuccessful termination of the included use case, is defined equivalent with q_F.

Definition 5.8 (Mapping a Success and Failure Step to UC-LTS). Success and Failure steps are mapped to a UC-LTS as follows:

$\mathcal{M}_{StepUcslts}[\![Success\ (id)]\!] = \mathcal{M}_{StepUcslts}[\![Failure\ (id)]\!] = U$ with

$$U = (\emptyset, \{q_0\}, q_0, \{q_0\}, \emptyset, \emptyset) \text{ with } q_0 = env.step(id)$$

Success and *Failure* steps denote the successful or unsuccessful termination of a use case scenario. In both cases the corresponding UC-LTS consists of only a single (final) state. We note that for a given use case n, the set of all states representing *Success* steps is captured by the entry $env(n).F_S$ in the global environment. Similarly, the set of states representing *Failure* steps is captured by $env(n).F_F$.

Step 2: Map use case step sequences to a UC-LTS:

As defined in the abstract syntax of a DSRG-style use case, the main success scenario, each extension scenario and the scenarios defined by a *Choice* step are lists of use case steps. The mapping of such a list of use case steps to a UC-LTS is defined in this sub-section.

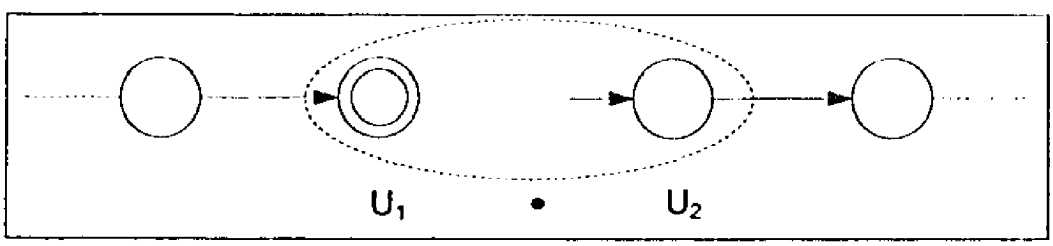


Figure 5.3. Sequential Composition of UC-LTSs U_1 and U_2

Definition 5.9. (Mapping a Step Sequence a UC-LTS). Let $(step_1, step_2, \ldots step_k)$ be a non-empty step sequence of $k > 1$ steps. We then define the mapping to a UC-LTS as follows:

$\mathcal{M}_{StepseqUcslts}[\![(step_1, step_2, \ldots step_k)]\!] = \mathcal{M}_{StepUcslts}[\![step_1]\!] \cdot \cdots \cdot \mathcal{M}_{StepUcslts}[\![step_k]\!]$

The mapping of a (non-empty) list of use case steps corresponds to the binary sequential composition (·) of the UC-LTSs of the individual steps. The mapping of an empty list is not supported since, according to the well-formedness conditions of Definition 4.2, such a case not possible. As schematically depicted in Figure 5.3, the sequential composition consists of unifying the final success states of the first operand and the initial state of the second operand. A formal definition of the sequential composition of UC-LTSs is given next.

Definition 5.10 (Binary Sequential Composition of UC-LTSs). Given two UC-LTSs $U_1 = (\Sigma_1, Q_1, q_{0_1}, F_1, \delta_1, \tau_1)$ and $U_2 = (\Sigma_2, Q_2, q_{0_2}, F_2, \delta_2, \tau_2)$ such that τ_1 and τ_2 are non-conflicting and Q_1 and Q_2 are disjoint, we define the *sequential composition* (·) as follows: $U_1 \cdot U_2 = U$ with

$U = \left(\Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2 \setminus \{q_{0_2}\}, q_{0_1}, F_2, \delta, \tau_1 \cup \tau_2\right)$ where δ is defined as:

$$\delta(q,w) = \begin{cases} \delta_1(q,w), & if\ q \in Q_1 \setminus F_{S_1} \\ \delta_2(q,w), & if\ q \in Q_2 \setminus \{q_{0_2}\} \\ \delta_2(q_{0_2},a) \cup \delta_1(q,w), & if\ q \in F_{S_1} \end{cases}$$

where $(q,w) \in dom\ \delta = (Q_1 \cup Q_2 \cup \{q_0\}) \times \mathbb{P}_1(\Sigma_1 \cup \Sigma_2)$.

Step 3: Map a DSRG-style use case to a set of UC-LTSs:

A DSRG-style use case is mapped into a set of UC-LTSs. The resulting set contains one UC-LTS for the main success scenario and one UC-LTS for each defined extension. Note that the mapping captures the behavioral part of the use case and abstracts away from some of the (static) information, such as the use case properties and the conditions of the use case extensions.

Definition 5.11. (Mapping a DSRG-style Use Case to UC-LTS). Let $uc = (n, Prop, Mss, \{ex_1, ex_2, \ldots, ex_n\})$ with $ex_i = (id_{ex_i}, condition_i, s_i)$ be a DSRG-style use case. We then obtain the corresponding set of UC-LTSs as follows:

$$\mathcal{M}_{UcUclts}[\![uc]\!] = \{\mathcal{M}_{StepseqUclts}[\![Mss]\!], \mathcal{M}_{StepseqUclts}[\![s_1]\!], \ldots, \mathcal{M}_{StepseqUclts}[\![s_n]\!]\}$$

For illustrative purposes, Figure 5.4 portrays the set of UC-LTSs of the "Order Product" use case. As depicted, the set consists of six UC-LTSs; one for the main success scenario and one for each of the five extensions. States that belong to the same equivalence class (by means of ~) are circled by a dashed line. During merging these states will be combined to a single state to obtain a single consolidated UC-LTS.

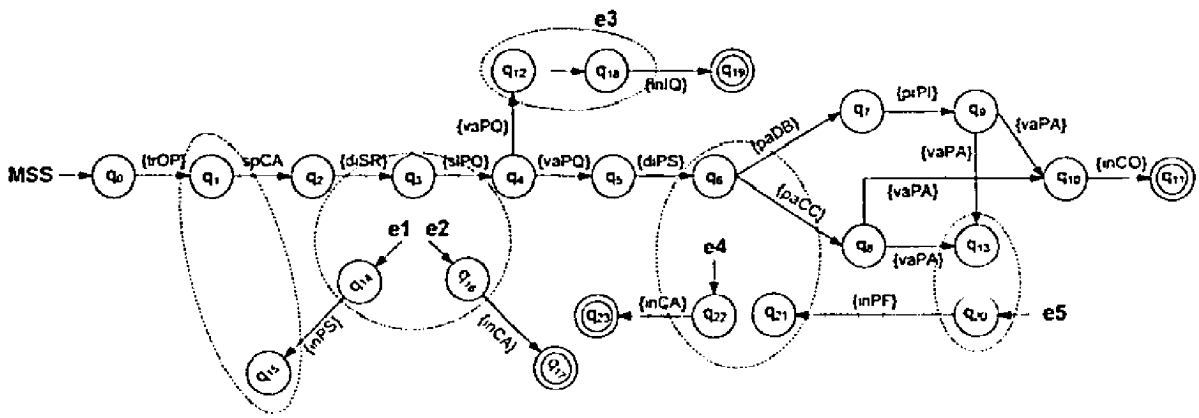


Figure 5.4 UC-LTSs of the "Order Product" Use Case

Step 4: Mapping a set of DSRG-style use cases to a set of UC-LTSs:

We define the mapping of a set of DSRG-style use cases to a set of UC-LTSs as the union of the sets of UC-LTSs representing the various use cases.

Definition 5.12. (Mapping a Set of DSRG-style Use Cases to a set of UC-LTSs). Let $\{uc_1, uc_2, \ldots, uc_m\}$ be a set of DSRG-style use cases. We then obtain the corresponding set of UC-LTSs as follows:

$$\mathcal{M}_{UcsUclts}[\![\{uc_1, uc_2, \ldots, uc_m\}]\!] = \bigcup_{i=1}^{m} \mathcal{M}_{UcUclts}[\![uc_i]\!]$$

In the next section we map a DSRG-style use case model to a UC-LTS by defining how a set of UC-LTSs is merged into a single UC-LTS.

5.1.2 Merging a Set of UC-LTSs to a UC-LTS

A DSRG-style use case model is mapped to the intermediate semantic domain of UC-LTS by merging the UC-LTSs representing the various entailed use cases. The merge is performed on basis of the global equivalence relation (~).

Definition 5.13. (Mapping a DSRG-style Use Case Model to UC-LTS) Let $D = (n_0, UC)$ be a well-formed DSRG-style use case model in canonical form, $\{uc_1, uc_2, \ldots, uc_m\}$ be the range of UC, and $\{U_1, U_2, \ldots, U_n\}$ be the result of $\mathcal{M}_{UcsUclts}[\![\{uc_1, uc_2, \ldots, uc_m\}]\!]$ with $U_i = (\Sigma_i, Q_i, q_{0_i}, F_i, \delta_i, \tau_i)$ and $n \geq m$. The mapping to UC-LTS is then defined as follows:

$\mathcal{M}_{UcmUclts}[\![n_0, UC]\!] = (\Sigma, Q, q_0, F, \delta, \tau)$ with

$\Sigma = \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n$

$Q = (Q_1 \cup Q_2 \cup \ldots \cup Q_n)/\sim$

$q_0 = [env.uc(n_0).q_0]$

$F = \Pi(env.uc(n_0).F_S \cup env.uc(n_0).F_F)$, where Π is the generalized canonical projection map defined as $\Pi(Q, \sim) = \{\pi(q, \sim) | \, q \in Q\}$

$$\delta([q]_\sim, w) = \bigcup_{\hat{q} \in [q]} \bigcup_{i=1}^{n} \Pi(\delta_i(\hat{q}, w), \sim)$$

$\tau = \tau_1 \cup \tau_2 \cup \ldots \cup \tau_n$

The set of states of the resulting UC-LTS is the set of equivalence classes in $(Q_1 \cup Q_2 \cup \ldots \cup Q_n)$ with respect to $\sim$. The initial and final states are the equivalence-class counterparts of the initial and (success and failure) final states of the root use case n_0. Rather than on states, the transition function δ is defined on equivalence classes of states. For a given equivalence class and a set of labels, it denotes the set of equivalence classes of all states that are reachable from any member of $[q]$ after having accepted w.

For illustrative purposes, let us assume that the "IMS" use case model consists of only the "Order Product" use case (which is clearly in canonical form). Then, the corresponding UC-LTS (Figure 5.5) is derived by merging the various UC-LTSs given in Figure 5.4. The resulting UC-LTS has four final states: $[q_{11}]$ denoting the successful outcome (i.e., the customer succeeded to order the product), $[q_{17}]$ and $[q_{23}]$ denoting the case where the user cancels the use case, and $[q_{19}]$ denoting the case where the product is not available. Notice that the resulting UC-LTS has fewer states than the accumulated number of states

of the involved UC-LTSs. This is because, during merging, two or more states are combined into a single state, representing the respective equivalence class.

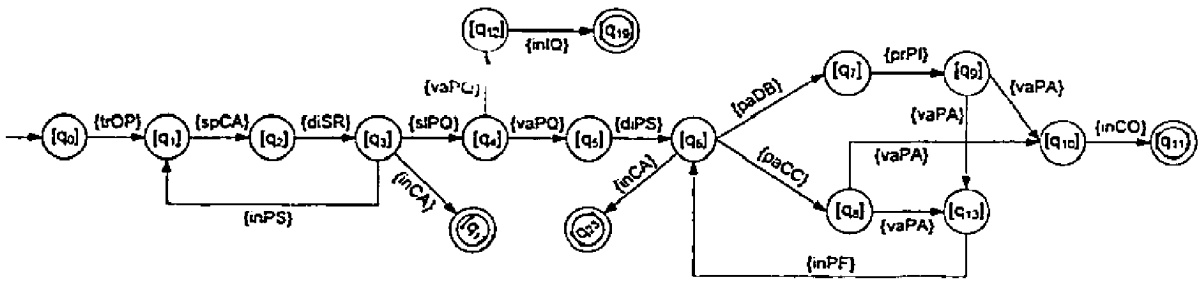


Figure 5.5. "Order Product" UC-LTS

5.2 Intermediate Semantic Domain for Task Models

In this section we define the intermediate semantic domain for task models called *Generic Task Model* (GTM). We also specify how non-recursive, well-formed ECTT task models are mapped into a corresponding GTM specification.

Definition 5.14. (Generic Task Model). A generic task model G is a tuple $G = (T, \psi, \tau)$ where,

> $T \in \mathbb{P}\, TASKNAME$ is a set of atomic actions, where each action corresponds to the name of the task it represents.
>
> $\psi \in GTE$ is a generic task expression (see Definition 5.15), and
>
> $\tau \in T \longrightarrow \{interaction, application\}$ is a (total) typing function that associates a type to each atomic task.

In contrast to an ECTT task model (Definition 4.3), a generic task model abstracts away from high-level task names (i.e. task definitions). Instead of using task definitions, the behavior of the task model is captured in a single generic task expression (Definition 5.15). While high-level task names are important at the modeling stage to foster the comprehension of the task model, they are irrelevant for capturing behavioral information. It is also notable that the typing function (τ) no longer maps tasks to type $abstract$. This is a direct consequence of the before-mentioned omission of high-level

task names and the fact that only high-level tasks may be of type *abstract* (Definition 4.5).

Definition 5.15. (Generic Task Expression). Let ψ and ρ be generic task expressions and $\alpha \in T$ be an atomic task, then the set of *generic task expression GTE* is the smallest set closed under following rules:

(1) α is a generic task expression

(2) $\psi \gg_{gte} \rho, \psi \,[\,]_{gte}\, \rho, \psi \,|||_{gte}\, \rho,\ [\psi]_{gte}, (\psi^*)_{gte}, stop_{gte}(\psi), resume_{gte}(\psi).$

Note that in what follows, for the sake of brevity, we will omit the gte subscript for GTE operators, if it is clear from the context. Compared to an ECTT task expression, a generic task expression may not contain high-level operators (e.g. *order independency*, *disabling*, or *suspend / resume*). These operators are important, as syntactic sugar, at the modeling stage to obtain a concise and comprehensible task model. However, they do not enrich the expressiveness of an ECTT task expression and can be rewriting using low-level operators. In what follows we present a mapping which transforms each ECTT task expression into a corresponding generic task expression.

Definition 5.16. (Mapping an ECTT Task Expression to a Generic Task Expression). Let v, φ be ECTT task expressions, n be a task name and $\mathcal{D}$ be a finite map of ECTT task definitions. We then define the mapping $\mathcal{M}_{ExprGte}$ to generic task expressions, relative to a given task definition map $\mathcal{D}$, as follows:

$$\mathcal{M}_{ExprGte}[\![n]\!]_{\mathcal{D}} = \begin{cases} \mathcal{M}_{ExprGte}[\![\mathcal{D}(n)]\!]_{\mathcal{D}}, & if\ n \in dom(\mathcal{D}) \\ n, & otherwise \end{cases}$$

$$\mathcal{M}_{ExprGte}[\![v \gg \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}} \gg \mathcal{M}_{ExprGte}[\![\varphi]\!]_{\mathcal{D}}$$

$$\mathcal{M}_{ExprGte}[\![v \,[\,]\, \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}} \,[\,]\, \mathcal{M}_{ExprGte}[\![\varphi]\!]_{\mathcal{D}}$$

$$\mathcal{M}_{ExprGte}[\![v \,|||\, \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}} \,|||\, \mathcal{M}_{ExprGte}[\![\varphi]\!]_{\mathcal{D}}$$

$$\mathcal{M}_{ExprGte}[\![v \boxplus \varphi]\!]_{\mathcal{D}} = \left(\mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}} \gg \mathcal{M}_{ExprGte}[\![\varphi]\!]_{\mathcal{D}}\right) [\,]$$

$$\left(\mathcal{M}_{ExprGte}[\![\varphi]\!]_{\mathcal{D}} \gg \mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}}\right)$$

$$\mathcal{M}_{ExprGte}[\![v\ [>\ \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{ExprGte}[\![\mathcal{O}[\![v]\!]_{\mathcal{D}} \gg \varphi\ ]\!]$$

$$\mathcal{M}_{ExprGte}[\![v\ |>\ \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{ExprGte}[\![\mathcal{I}[\![v]\!]_{\mathcal{D}}\ \varphi^*]\!]$$

$$\mathcal{M}_{ExprGte}[\![[v]]\!]_{\mathcal{D}} = [\mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}}]$$

$$\mathcal{M}_{ExprGte}[\![v^*]\!]_{\mathcal{D}} = (\mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}})^*$$

$$\mathcal{M}_{ExprGte}[\![v^+]\!]_{\mathcal{D}} = \mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}} \gg (\mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}})^*$$

$$\mathcal{M}_{ExprGte}[\![stop(v)]\!]_{\mathcal{D}} = stop(\mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}})$$

$$\mathcal{M}_{ExprGte}[\![resume(v)]\!]_{\mathcal{D}} = resume(\mathcal{M}_{ExprGte}[\![v]\!]_{\mathcal{D}})$$

While most expressions (i.e. $\gg$, [ ], $|||$, $*$, $stop$, $resume$) are directly mapped to a corresponding GTE expression, ECTT task expressions of form $v\ [> \varphi$ (disabling) or $v\ |> \varphi$ (suspend / resume) are first rewritten into an ECTT task expressions without [> and |>, before the semantic function is applied. For this purpose the auxiliary functions *deep optionalization* ($\mathcal{O}$) and *interleaved insertion* ($\mathcal{I}$) have been defined. The former is a function that defines every sub-task of its target task expression as optional. However, if the sub-tasks are executed, they have to be executed in their predefined order. The latter is a function that "injects" the task specified by its second operand at any possible position in between the (sub) tasks of the first operand. Formal definitions of $\mathcal{O}$ and $\mathcal{I}$ together with additional explanation why *disabling* and *suspend / resume* can be rewritten using them are given in Appendix C.

Definition 5.17. (Mapping an ECTT Task Model to a Generic Task Model). Let $C = (n_0, \mathcal{D}, \tau_{ECTT})$ be a well formed, recursion free ECTT task model. We then obtain the corresponding generic task model G as follows: $\mathcal{M}_{EcttGtm}[\![C]\!] = G = (T, \psi, \tau)$, where,

$$T = \{n \in dom(\tau_{ECTT}) \mid n \notin dom(\mathcal{D})\}$$

$$\psi = \mathcal{M}_{ExprGte}[\![n_0]\!]_{\mathcal{D}}$$

$$\tau(n) = \tau_{ECTT}(n) \text{ for } n \in T$$

In order to illustrate the proposed mapping, let us assume that the "IMS" task model consists only of the "Order Product" task definition without recursion as defined in Table 4.8. Then, the corresponding generic task model is defined as follows (Table 5.4):

Table 5.4. Generic Task Model Definition of "Order Product" Task Model

$$T = \{\text{"chOP", "slCR", "sbCR", "inQO", "inRS", "slPD", "slQT", "sbPS", "swCC", "dpRS", "dpPS", "dpOS", "dpCF", "dpPF"}\}$$

$$\psi = \text{"chOP"} \gg (\text{"slCR"} \gg \text{"sbCR"} \gg \text{"dpRS"} \gg (stop(\text{"inQO"}) [\,] \text{"inRS"}))^* \gg \text{"slCR"} \gg \text{"sbCR"} \gg \text{"dpRS"} \gg \text{"slPD"} \gg \text{"slQT"} \gg \text{"sbPS"} \gg (\text{"dpPS"} [\,] stop(\text{"dpOS"})) \gg (stop(\text{"inQO"})[\,] (\text{"swCC"} \gg \text{"dpPF"}))^* \gg \text{"swCC"} \gg \text{"dpCF"}$$

$$\tau(t) = \begin{cases} interaction, & if\ t \in \{\text{"inQA", "chOP", "slCR", "sbCR", "inQO", "inRS", "slPD", "slQT", "sbPS", "swCC"}\} \\ application, & if\ t \in \{\text{"dpRS", "dpPS", "dpOS", "dpCF", "dpPF"}\} \end{cases}$$

6 Semantic Domain: Sets of Partial Order Sets

In this chapter we define the second-level mappings (Figure 4.1) to the semantic domain of *Sets of Partial Order Sets* (posets). We start by providing necessary definitions, then we present a procedure that, given a UC-LTS, generates the corresponding set of posets. We also define a semantic function that maps any generic task model (GTM) to a corresponding set of posets.

6.1 Definitions

This section provides formal definitions of (sets) of partial order sets as well as related concepts and operations needed for the semantic mappings.

6.1.1 Preamble

Fundamental to our approach are the concepts *event* and *event name.*

Definition 6.1 (Events). Let $EVENTNAME$ represent the set of all possible event names. We then define an *event* as a pair consisting of an event name n and an index i. Correspondingly the set of all events is defined as:

$EVENT = EVENTNAME \times \mathbb{N}$.

For all $(n,\ i) \in EVENT$ we define the obvious projection functions

$$name: EVENTNAME \times \mathbb{N} \longrightarrow EVENTNAME, \text{such that } name(n, i) \longmapsto n$$

$$index: EVENTNAME \times \mathbb{N} \longrightarrow \mathbb{N}, \qquad (n, i) \longmapsto i$$

and their generalizations, applied to sets of events being applied to all elements of the set. In what follows, event names may be used to represent events with index 0; i.e., as needed, we assume the implicit conversion from $EVENTNAME \longrightarrow EVENT$, defined by $n \longmapsto (n, 0)$. We will use the symbol E, possibly decorated with primes ($E', E'', ...,$) and/or subscripts ($E_1,\ E_2, ...,$), to represent a subset of $EVENT$.

Definition 6.2 (STOP). We reserve the name $STOP \in EVENTNAME$; its semantics will be given later.

Definition 6.3 (Event Type). The function $\tau: EVENTNAME \longrightarrow \{internal,\ interaction,\ application\}$ associates an *event type* to each event name. We also naturally generalize τ to $e \in EVENT$: $\tau(e) = \tau(name(e))$.

Next we define the set operation *disjoint union*. It is used as an auxiliary function for the definition of composition operators for partially ordered sets, which are needed for the semantic mapping.

Definition 6.4 (Disjoint Union). Using standard notation, we represent the *disjoint union* of two event sets as $+$:

$$E_p + E_q = E_p^{\bullet 0} \cup E_q^{\bullet 1}, \text{ where}$$

$$E^{\bullet b} = \{e * b \mid e \in E\} \text{ for } b \in \{0,1\} \text{ with}$$

$$(n, i) * b = (n, i \times 2 + b)$$

Our definition of disjoint union is similar to what has been proposed in [Blyth 1975]. In both cases an index set is used to distinguish between events that have the same name. In contrast to Blyth, however, we use a natural number instead of an n-ary tuple over $\{0,1\}$.

We generalize $+$ and $\_^{\bullet b}$ to binary relations over $EVENT$; i.e. given $R: \mathbb{P}(EVENT \times EVENT)$ we define $R^{\bullet b} = \{(e * b, e' * b) | (e, e') \in R\}$ and $R_p + R_q = R_p^{\bullet 0} \cup R_q^{\bullet 1}$.

6.1.2 Partial Order Sets

The building blocks for the semantic domain presented in this chapter are partially ordered sets (posets).

Definition 6.5 (Poset over Events). A partially ordered set (poset) over *events* is a tuple $(E, \leq)$, where

$E \subseteq EVENT$ is a set of events, and

$\leq \subseteq E \times E$ is a partial order relation (reflexive, anti-symmetric, transitive) defined over E. This relation specifies the causal order of events.

Definition 6.6 (Empty and Singleton Poset). We will use the symbol $\emptyset_{poset}$ to denote the empty poset $(\emptyset, \emptyset)$. If e is an event, then e_{poset} denotes the singleton poset $(\{e\}, \{(e, e)\})$.

In order to be able to compose posets we define the operations *sequential* and *parallel composition.*

Definition 6.7 (Sequential Composition of Posets). Let $p = (E_p, \leq_p)$ and $q = (E_q, \leq_q)$ be posets. We define the *sequential composition* $r = p \cdot q$ as follows.

$$p \cdot q = \begin{cases} p, & STOP \in name(E_p) \\ (E_r, \leq_r), & otherwise \end{cases}$$

where

$$E_r = E_p + E_q$$

$$\leq_r = (\leq_p + \leq_q) \cup \{(e_p * 0, e_q * 1) \mid e_p \in E_p, e_q \in E_q\}$$

Intuitively, if the event set of p does *not* contain STOP, then the *sequential composition* $p \cdot q$ places all events of q strictly after all the events of p. Otherwise, $p \cdot q$ simplifies to p regardless of q. In contrast to the *sequential composition*, the *parallel composition* does not make a case distinction between posets that contain or do not contain an element named STOP.

Definition 6.8 (Parallel Composition of Posets). Let $p = (E_p, \leq_p)$ and $q = (E_q, \leq_q)$ be posets. We define the *parallel composition* $p \parallel q = (E_p + E_q, \leq_p + \leq_q)$.

The operations *event hiding* and *event renaming* are needed for the refinement definitions of Chapter 9. The former is additionally used as an auxiliary function of the subsequent *closing operation.*

Definition 6.9 (Event Hiding). Let $p = (E_p, \leq_p)$ be a poset and E_S be a set of events. We define the operation *event hiding*: $p \setminus E_S = (E_r, \leq_r)$, where

$$E_r = E_p - E_S$$

$$\leq_r = \leq_p \cap (E_r \times E_r)$$

Similarly, let Σ be a set of event names. We then define $p \setminus \Sigma$ as $p \setminus (\Sigma \times \mathbb{N})$.

Definition 6.10 (Event Renaming). Let $p = (E_p, \leq_p)$ be a poset and $map: \Sigma \rightarrow \Sigma$ be an injective event map. We then define the operation *event renaming* $rename(p, map) = (E_r, \leq_r)$ as follows:

$$E_r = \{(map(n), i) \mid (n, i) \in E_p\}$$

$$\leq_r = \{((map(n), i), (map(n'), i')) \mid (n, i) \leq_p (n', i')\}$$

The insertion and removal of an event named STOP into / from a poset are the so-called *closing* and *opening* operations and are formally defined as follows.

Definition 6.11 (Closing a Poset). Let $p = (E_p, \leq_p)$ be a poset. We define the closing operation $close(p) = (E_p \cup \{STOP\}, \leq_p \cup \{(STOP, STOP)\})$.

Definition 6.12 (Opening a Poset). Let $p = (E_p, \leq_p)$ be a poset. We define the opening operation $open(p) = p \setminus \{STOP\}$

Also fundamental to our model is the notion of a *trace*. In general, a trace of a partial order set corresponds to a totally ordered event-name sequence such that the corresponding sequence of events is a linear extension of the partial order. It is important to note that events with event name $STOP$ are *not* part of a trace.

Definition 6.13 (Set of All Traces). Let $p = (E_p, \leq_p)$ be a poset. We define the function $Tr: POSET \longrightarrow \mathbb{P}(seq\ EVENTNAME)$ which yields the set of all traces of p as follows:

$$Tr(p) = \left\{ \begin{array}{c} \langle name(e_1), name(e_2), \ldots, name(e_n) \rangle \mid \forall i, j \in \{1, \ldots, n\}\ i < j \Rightarrow e_j \not\leq_p e_i \wedge \\ \{e_1, e_2, \ldots, e_n\} = E_p \setminus \{\text{STOP}\} \end{array} \right\}$$

Note that $e_j \not\leq e_i$ holds true, if either $e_i \leq e_j$ or e_i and e_j are unrelated (by means of $\leq_p$).

We conclude this section with a discussion about the *closure properties* of posets for the operations defined.

Proposition 6.1. (Closure Properties of Posets). Posets are closed under the operations *sequential composition* ($\cdot$), *parallel composition* ($\parallel$), *event hiding* ($\setminus$) as well as *event renaming*.

PROOF: The correctness of the proposition is proven by a set of Isabelle/HOL lemmas which are given in Appendix E.1.

It follows directly from Proposition 6.1 that posets are also closed under the operations *close* and *open*.

6.1.3 Set of Partial Order Sets

In this section we define the semantic domain of *sets of partial order sets*. The provided definitions are directly needed for the semantic mappings (Section 6.2) as well as the definition of refinement between use case and task models (Chapter 9).

Definition 6.14 (Set of partial oder). A *set of partial order sets* P is a possibly infinite collection of posets $\{p_1, p_2, \ldots\}$.

We define the *set of all traces* for a set of posets. It forms the essential basis for establishing an equivalence relation between sets of posets and thus the definition of a refinement relation between use case and task models (Chapter 9).

Definition 6.15 (Set of All Traces of a Set of Posets). The *set of all traces* of a set of posets P is defined as:

$$Tr(P) = \bigcup_{p_i \in P} Tr(p_i)$$

Definition 6.16 (Trace Equivalence). Let P_1 and P_2 be two sets of posets. P_1 and P_2 are *trace equivalent* $P_1 \equiv_{trace} P_2$ iff: $Tr(P_1) = Tr(P_2)$

In order to illustrate the concept of trace equivalence let us consider the following example posets.

$$P_1 = \{(\{a, b\}, \{(a, b), (a, a), (b, b)\}), (\{a, b\}, \{(b, a), (a, a), (b, b)\})\}$$

$$P_2 = \{(\{a, b\}, \{(a, a), (b, b)\})\}$$

$$R_1 = \{(\{a, b\}, \{(a, b), (a, a), (b, b)\})\}$$

$$R_2 = \{(\{a, b\}, \{(b, a), (a, a), (b, b)\})\}$$

Clearly, in P_1 and P_2 the events a and b can appear in any order with $Tr(P_1) = Tr(P_2) = \{\langle a, b\rangle, \langle b, a\rangle\}$. Hence P_1 and P_2 are trace equivalent ($P_1 \equiv_{trace} P_2$). Contrary to this, in

R_1 and R_2 the order of a and b is restricted and since $Tr(R_1) = \{\langle a, b\rangle\}$ but $Tr(R_2) = \{\langle b, a\rangle\}$ we conclude that R_1 and R_2 and *not* trace equivalent.

Sets of posets can be composed by the following binary operators. In contrast to the binary operators defined on posets we additionally introduce the operation of *alternative* composition. It is needed for the semantic mappings defined in Section 6.2.

Definition 6.17 (Binary Operators for Sets of Posets). Let P and R be sets of posets. We define the *sequential composition* ($\cdot$), *parallel composition* ($\|$), and *alternative composition* ($\#$) as follows:

$$P \cdot R = \{p_i \cdot r_j \mid p_i \in P, r_j \in R\}$$
$$P \parallel R = \{p_i \parallel r_j \mid p_i \in P, q_j \in R\}$$
$$P \# R = P \cup R$$

We also define the *closure* operation for a set of posets. Similar to the Kleene star operation for regular expressions [Hopcroft et al. 2007], it returns the set of posets that are formed by computing the union of all possible (repeated) sequential compositions of a given set of posets.

Definition 6.18 (Closure of a Set of Posets). Let P be a set of posets. We define the *closure* (P^*) of P as follows.

$$P^* = \bigcup_{k=0}^{\infty} P^k$$

where P^k is defined as

$$P^k = \begin{cases} \{\emptyset_{poset}\}, & if\ k = 0 \\ P \cdot P^{k-1}, & k > 0 \end{cases}$$

The following four operations defined on a set of posets make use of their respective counter-part operation defined for posets.

Definition 6.19 (Closing and Opening a Set of Posets). Let P be a set of posets. We define the *closing* operation $close(P) = \{close(p) \mid p \in P\}$. Similarly we define the *opening* operation $open(P) = \{open(p) \mid p \in P\}$

Based on the definition of the set of all traces (Definition 6.15), we derive the following trace properties for sets of posets operations. The trace properties will be needed in Chapter 8, where we establish a formal correspondence between the sets of posets and nFSM semantics.

Proposition 6.2. (Trace Properties of Sets of Posets Operations). The sets of posets operations: *sequential composition* (·), *parallel composition* (‖), *alternative composition*(#), *closure* (*), *close* and *open* have the following trace properties:

Operation	**Trace Property**
$P \cdot R$	$Tr(P \cdot R) = \{x^{\wedge}y \mid x \in Tr(p) \wedge y \in Tr(R) \wedge p = (E_p, \leq_p) \in P \wedge STOP \notin E_p\} \cup \{x \mid x \in Tr(p) \wedge p = (E_p, \leq_p) \in P \wedge STOP \in E_p\}$, where ^ denotes the sequential composition of two event name sequences.
$P \parallel R$	$Tr(P \parallel R) = \cup\{x \mid\mid\mid y \mid x \in Tr(P) \wedge y \in Tr(R)\}$, where $\mid\mid\mid$ is defined as: [Roscoe 2005] $\langle\ \rangle \mid\mid\mid s = \{s\}$ $s \mid\mid\mid \langle\ \rangle = \{s\}$ $\langle a\rangle^{\wedge}s \mid\mid\mid \langle b\rangle^{\wedge}t = \{\langle a\rangle^{\wedge}u \mid u \in (s \mid\mid\mid \langle b\rangle^{\wedge}t)\} \cup \{\langle b\rangle^{\wedge}u \mid u \in (\langle a\rangle^{\wedge}s \mid\mid\mid t)\}$
$P \# R$	$Tr(P \# R) = Tr(P) \cup Tr(Q)$
P^*	$Tr(P^*) = \cup_{k=0}^{\infty} Tr(P^k)$, where $Tr(P^k)$ is defined as: $Tr(P^k) = \begin{cases} \{\langle\ \rangle\}, & k = 0 \\ Tr(P), & k = 1 \\ \{x^{\wedge}y \mid x \in Tr(p) \wedge p = (E_p, \leq_p) \in P \wedge STOP \notin E_p \wedge y \in Tr(P^{k-1})\}, & k > 1 \end{cases}$
$close(P)$	$Tr(close(P)) = Tr(P)$
$open(P)$	$Tr(open(P)) = Tr(P)$

The proof for the proposition is given in Appendix E.1.

Two important auxiliary functions for the refinement definitions of Chapter 9 are *event hiding* and *refinement mapping*.

Definition 6.20 (Event Hiding). Let P be a set of posets and Σ_{hide} be a set of event names. We define the operation *event hiding* as follows:

$$P \setminus \Sigma_{\text{hide}} = \{p \setminus \Sigma_{\text{hide}} \mid p \in P\}$$

Definition 6.21 (Refinement Mapping). Let P be a set of posets and $rmap: \mathbb{P}(\Sigma) \longrightarrow \Sigma$ be a event map with a finite domain such that:

1. $dom(rmap)$ partitions $\bigcup_{(E,\leq)\in P} name(E)$ such that:
   a. $\forall \Sigma', \Sigma'' \in dom(rmap). (\Sigma' \cap \Sigma'' \neq \emptyset) \Longrightarrow (\Sigma' = \Sigma'')$
   b. $\bigcup_{\Sigma \in dom(rmap)} \Sigma = \bigcup_{(E,\leq)\in P} name(E)$
2. $\forall \Sigma \in dom(rmap). \forall n \in \Sigma. \tau(n) = \tau(rmap(\Sigma))$
3. $\forall \Sigma_p, \Sigma_q \in dom(rmap) . \left(\Sigma_p \neq \Sigma_q \wedge rmap(\Sigma_p) = rmap(\Sigma_q) \Longrightarrow (\forall p, q \in P . (\Sigma_p \subseteq name(E_p) \wedge \Sigma_q \subseteq name(E_q) \Longrightarrow p \neq q)\right)$

In order to rule out ambiguous mappings (e.g., $\{a, b\} \mapsto d$ and $\{a, c\} \mapsto e$.), condition 1a ensures that the domain elements of $rmap$ are pairwise disjoint. Condition 1b ensures that the refinement mapping covers all events in P. With condition 2, it is guaranteed that only events of the same type are involved in a mapping. Finally, condition 3 ensures that, if two or more different source event names (e.g. a and b) are mapped to the same target event (e.g., $\{a\} \mapsto d$ and $\{b\} \mapsto d$), then the respective source events must be distributed over distinct posets in P.

We then obtain the resulting set of posets $P_{Ref} = ref(P, rmap)$ as follows:

(1)	***var*** map:$\Sigma \longrightarrow \Sigma$ initialized to $\emptyset$ ***var*** P_{Min}:SPOSET := P
(2)	***for each*** element Σ in $dom(rmap)$ ***do***
(3)	***select*** $n \in \Sigma$ ***do***
(4)	$map := map \cup \{n \mapsto rmap(\Sigma)\}$ $P_{Min} = P_{Min} \setminus (\Sigma - \{n\})$
(5)	***od*** ***od***
(6)	$P_{Ref} = \{rename((E, \leq), (name(E) \lhd map)) \mid (E, \leq) \in P_{Min}\}$

The procedure starts (1) with initializing the renaming function map with the empty set and then creating a copy of P named P_{Min}. (2) For each domain element Σ in the finite domain of $rmap$, (3) a "representative" though arbitrary event name n in Σ is selected and (4) the pair $\{n \mapsto rmap(\Sigma)\}$ is added to map. Condition 1a guaranties that there will not be any ambiguous mappings originating from n. (5) Next, all events with event names in Σ (except for n) are removed from P_{Min}. Once done, (6) the remaining events in P_{Min} are renamed according to map in order to obtain P_{Ref}. In order to ensure injectivity, the domain of map is restricted to event names defined in the currently "visited" poset. It follows from condition 3 that this is a sufficient enough condition.

We note that step 3 of the algorithm is nondeterministic. Hence, for a given input, the algorithm may produce different results. However, for the semantic mappings defined in the next section and the proposed usage of the event map $rmap$ (Chapter 9), all possible results are *trace equivalent*. It will be shown, that this is a strong enough condition for the verification problems tackled in this thesis.

The following example illustrates the refinement mapping:

$$P = \left\{\left(\{(n_1, 1), (n_2, 2), (n_3, 3)\}, \{((n_1, 1), (n_2, 2)), ((n_1, 1), (n_3, 3))\}^{=^7}\right)\right\}$$

$$rmap = \{\{n_1\} \mapsto a_1, \{n_2, n_3\} \mapsto a_2\}$$

Clearly, the domain elements of $rmap$ are pairwise disjoint ($\{n_1\} \cap \{n_2, n_3\} = \emptyset$), there are no event names in *different* domain elements that are mapped to the same target event name, and the refinement mapping covers all events in P. We also assume that all

[7] – denotes the reflexive closure

involved events have the same type. After completing steps 2-5 (n_1 and n_2 were selected as representative event names) we obtain P_{Min} and map as follows:

$$P_{Min} = \{(\{(n_1, 1), (n_2, 2)\}, \{((n_1, 1), (n_2, 2))\}^{=})\}$$

$$map = \{n_1 \mapsto a_1, n_2 \mapsto a_2\}$$

The event renaming according to map yields the following resulting set of posets:

$$P_{Ref} = ref(P, rmap) = \{(\{(a_1, 1), (a_2, 2)\}, \{((a_1, 1), (a_2, 2))\}^{=})\}$$

6.2 Semantic Rules

In this section we define the semantic mappings from the intermediate semantic domains (namely *UC-LTS* and *Generic Task Model*) to sets of posets.

6.2.1 Mapping UC-LTSs to Sets of Posets

Definition 6.22. (Mapping UC-LTS to a Set of Posets). Let $U = (\Sigma, Q, q_0, F, \delta, \tau)$ be a UC-LTS. We then define the mapping to a set of posets as follows:

$\mathcal{M}_{UcltsSposet}[\![U]\!] = LTS\_to\_SPO(U)$ with the global typing function τ_{sposet} defined as $\tau_{sposet} = \tau$. For this purpose we have devised the algorithm *LTS_to_SPO*. Table 6.1 gives the corresponding pseudo code.

Without loss of generality, the algorithm assumes that there are no outgoing transitions from any of the final states in the input UC-LTS. This is a valid assumption since, according to the well-formedness rules for DSRG-style use cases (Definition 4.2), *Failure* and *Success* steps are always at the end of any step sequence and cannot have any extensions. We also note that the main idea for the algorithm stems from the well-known algorithm that transforms a deterministic finite automaton into an equivalent regular expression [Hopcroft et al. 2007]. Instead of stepwise composing regular expressions, we compose sets of posets.

Table 6.1. ***LTS_to_SPO*** **Algorithm Transforming a UC-LTS to a Set of Posets**

(1)	*var* SPO:SPOSET[][] with all array elements initialized to ∅ *for each* transition (q_s, X, q_e) in δ *do* SPO$[q_s, q_e] := \{(X, id(X))\}$, where $id(X) = \{(l, l) \mid l \in X\}$ *od*
(2)	*for* each state s in $Q - (F \cup \{q_0\})$ *do*
(3)	*for* each pair of states q_k and p_m with $q_k \neq s \wedge p_m \neq s$ and $X, Y \in \mathbb{P}(\Sigma)$ such that $(q_k, X, s) \in \delta$ and $(s, Y, p_m) \in \delta$ *do*
(4)	$\mathbf{SPO}[q_k, p_m] := \mathbf{SPO}[q_k, p_m] \,\#\, (\mathbf{SPO}[q_k, s] \cdot \mathbf{SPO}[s, s]^* \cdot \mathbf{SPO}[s, p_m])$
(5)	$\delta := \delta \cup \{(q_k, \emptyset, p_m)\}$ *od*
(6)	$Q = Q - \{s\}$ *od*
(7)	*var* P_{result}:SPOSET := ∅ *for each* q_f in (F) *do* $P_{result} := P_{result} \,\#\, \mathbf{SPO}[q_0, q_f]$ *od*

(8)	*if* $\exists X \in \mathbb{P}(\Sigma)$ such that $(q_0, X, q_0) \in \delta$ *then*
	$P_{result} := \mathbf{SPO}[q_0, q_0]^* \cdot P_{result}$
	endif
	return P_{result}

The procedure starts (1) with the creation of an initial *generalized UC-LTS* internally represented by a two-dimensional array ('SPO'). The array is populated with all transitions of the given UC-LTS specification. Indexed by a source and a target state, an array cell contains a set of posets constructed from the label(s) associated to the representative transition. If the label is a singleton set, then the corresponding set of posets contains a single poset containing the respective event. If the label consists of multiple events, indicating the concurrent or unordered execution of use case steps, the set of posets will contain a poset which consists of several elements. Those elements, however, are not causally related. We note that the idea of the generalized UC-LTS is similar to the concept of a *generalized finite state machine* [Hopcroft et al. 2007]. Instead of labeling the transitions with regular expressions, however, transitions are labeled with sets of posets.

The core part of the algorithm consists of two nested loops. The outer loop (2) iterates through all states of the generalized UC-LTS (except for the initial and the final states)

whereas the inner loop (3) iterates through all pairs of incoming and outgoing transitions for a given state. For each found pair (q_k, p_m), we perform the following (4): Compute the *alternative composition* of:

SPO$[q_k, p_m]$ The set of posets associated with the transition from q_k to p_m. If such a transition does not exist we take the set of posets to be $\emptyset$.

and the result of the *sequential composition* of the following three sets of posets:

SPO$[q_k, s]$ Set of posets associated to the incoming transition

SPO$[s, s]^*$ The *closure* (Definition 6.18) of the set of posets associated to a possible self-transition defined over the currently visited state. If such a self transition does not exist then the iterative alternative composition yields $\{\emptyset_{poset}\}$.

SPO$[q_k, s]$ Set of posets associated to the outgoing transition.

Next (5) we add a new transition from the source state of the incoming transition to the target state of the outgoing transition. Note that the corresponding cell in *SPO* has already been populated with the result of (4).

Back in the outer loop, we eliminate (6) the currently visited state from the generalized UC-LTS and proceed with the next state. Once the generalized UC-LTS consists of only the initial state and the final states we exit the outer loop and perform the following two computations, in order to obtain the final result. First (7) we perform an *alternative composition* of the sets of posets of all the transitions from the initial state to a final state. Second, if the initial state additionally contains a self loop (8) then we *sequentially compose* the result of the *closure composition* of the set of posets denoted by that self loop and the result of the before-mentioned *alternative composition*.

If we apply the *LTS_to_SPO* algorithm to the "Order Product" UC-LTS of Section 5.1 we obtain the set of posets depicted in Table 6.2. For the sake of conciseness, we use a shorthand notation [] for sets of posets consisting of a single poset, within which all events are strictly sequentially ordered. For example,

$$[trOP] = \{(\{trOP\}, \{(trOP, trOP)\})\} = \{trOp_{poset}\}$$

$$[slPQ, vaPQ] = \{(\{slPQ, vaPQ\}, \{(slPQ, vaPQ)\}^*)\}$$

Table 6.2. Set of Posets Representation of "Order Product" Use Case

(1)	$[trOP] \cdot [spCA, diSR, inPS]^* \cdot [spCA, diSR, inCA]$ #
(2)	$[trOP] \cdot [spCA, diSR, inPS]^* \cdot [spCA, diSR, slPQ, vaPQ, inIQ]$ #
(3)	$[trOP] \cdot [spCA, diSR, inPS]^* \cdot [spCA, diSR, slPQ, vaPQ, diPS] \cdot$ $([paDB, prPI, vaPA, inPF]$ # $[paCC, vaPA, inPF])^* \cdot$
(a)	$[inCA]$ #
(b)	$[paDB, prPI, vaPA, inCO]$ #
(c)	$[paCC, vaPA, inCO]$

The various parts of the resulting set of posets are interpreted as follows: After having indicated the desire to order a product, the primary actor may search for a product for an arbitrary number of times until he/she either (1) elects to quit the system, (2) the selected product is not available in the desired quantity or (3) he/she decides to checkout and pay. In the latter case, the primary actor (possibly repeatedly) attempts to pay for the selected product until he/she either (3a) elects to quit the use case or the payment is successful (3b and 3c).

6.2.2 Mapping GTMs to Sets of Posets

This section specifies how a generic task model is mapped into a corresponding set of posets. We start with the definition of the semantic function.

Definition 6.23 (Mapping from GTM to Set of Posets). Let $G = (T, \psi, \tau)$ be a generic task model then the mapping to a set of posets is defined as follows:

$\mathcal{M}_{GtmSposet}[\![(T, \psi, \tau)]\!] = \mathcal{M}_{GteSposet}[\![\psi]\!]$ with the global typing function τ_{sposet} defined as $\tau_{sposet} = \tau$.

As given in Definition 6.24, $\mathcal{M}_{GteSposet}$ is defined in the common denotational style. An atomic generic task expression (denoted by α) is mapped to a set containing the corresponding singleton poset. Composite task expressions are represented by sets of posets, which are composed using the operators, defined in Section 6.1.3.

Definition 6.24 (Set of Posets Semantics of Generic Task Expressions). Let ψ, ρ be generic task expressions and α be an atomic task. We then define the mapping $\mathcal{M}_{GteSposet}$ to sets of posets as follows:

$$\mathcal{M}_{GteSposet}[\![\alpha]\!] = \{\alpha_{poset}\}$$
$$\mathcal{M}_{GteSposet}[\![\psi \gg \rho]\!] = \mathcal{M}_{GteSposet}[\![\psi]\!] \cdot \mathcal{M}_{GteSposet}[\![\rho]\!]$$
$$\mathcal{M}_{GteSposet}[\![\psi [\,] \rho]\!] = \mathcal{M}_{GteSposet}[\![\psi]\!] \# \mathcal{M}_{GteSposet}[\![\rho]\!]$$
$$\mathcal{M}_{GteSposet}[\![\psi \,|||\, \rho]\!] = \mathcal{M}_{GteSposet}[\![\psi]\!] \parallel \mathcal{M}_{GteSposet}[\![\rho]\!]$$
$$\mathcal{M}_{GteSposet}[\![[\psi]]\!] = \mathcal{M}_{GteSposet}[\![\psi]\!] \# \{\emptyset_{poset}\}$$
$$\mathcal{M}_{GteSposet}[\![\psi^*]\!] = \mathcal{M}_{GteSposet}[\![\psi]\!]^*$$
$$\mathcal{M}_{GteSposet}[\![stop(\psi)]\!] = close(\mathcal{M}_{GteSposet}[\![\psi]\!])$$
$$\mathcal{M}_{GteSposet}[\![resume(\psi)]\!] = open(\mathcal{M}_{GteSposet}[\![\psi]\!])$$

In what follows we illustrate the semantic rules by mapping the "Order Product" generic task model to a set of posets. We start with applying the semantic function $\mathcal{M}_{GteSposet}$ to the respective generic task expression.

$$\mathcal{M}_{GteSposet}[\![\text{chOP} \gg (\text{sICR} \gg \text{sbCR} \gg \text{dpRS} \gg (stop(inQO)\,[\,]\,inRS))^* \gg \text{sICR} \gg \text{sbCR} \gg \text{dpRS} \gg \text{sIPD} \gg \text{sIQT} \gg \text{sbPS} \gg (\text{dpPS}\,[\,]\,stop(\text{dpOS})) \gg (stop(\text{inQO})\,[\,]\,(\text{swCC} \gg \text{dpPF}))^* \gg \text{swCC} \gg \text{dpCF}]\!]$$

According to Definition 6.24 the application of $\mathcal{M}_{GteSposet}$ to the entire generic task expression is successively broken down into the application of $\mathcal{M}_{GteSposet}$ to sub-expressions and the corresponding set of posets operations. An intermediate result is given below.

$$\mathcal{M}_{GteSposet}[\![\text{chOP}]\!] \cdot \left(\mathcal{M}_{GteSposet}[\![(\text{slCR} \gg \text{sbCR} \gg \text{dpRS} \gg (stop(inQO)\ [\ ]\ inRS))]\!]\right)^{\bullet}$$
$$\cdot\ \mathcal{M}_{GteSposet}[\![\text{slCR}'' \gg \text{sbCR} \gg \text{dpRS} \gg \text{slPD} \gg \text{slQT} \gg \text{sbPS}]\!]$$
$$\cdot\ \mathcal{M}_{GteSposet}[\![\text{dpPS}[\ ]\ stop(\text{dpOS})]\!]$$
$$\cdot\ \left(\mathcal{M}_{GteSposet}[\![stop(\text{inQO})\ [\ ]\ (\text{swCC} \gg \text{dpPF})]\!]\right)^{\bullet} \cdot \mathcal{M}_{GteSposet}[\![\text{swCC} \gg \text{dpCF}]\!]$$

Table 6.3 depicts the resulting set of posets expression, using the same shorthand notation as introduced in the previous section. It allows a sub-set of the scenarios allowed by the set of posets of the corresponding use case (Table 6.2). Upon initiation, the user (possibly repeatedly) searches for a product until he/she either (1) elects to quit, (2) the selected item is out of stock or the product is available and the system displays the purchase summary. In the latter case, the user attempts to (possibly repeatedly) pay by credit card, until he/she (3a) elects to quit or (3b) the payment was successful. The option to pay by Debit Card (as specified in the use case) is not available. This may be due to restrictions of the supported user interface (refinement case 2, Chapter 3).

Table 6.3. Set of Posets Representation of "Order Product" Task Model

(1)	[chOP] · [slCR, sbCR, dpRS, *inRS*]* · [slCR, sbCR, dpRS, inQO, *STOP*] #
(2)	[chOP] · [slCR, sbCR, dpRS, *inRS*]* · [slCR, sbCR, dpRS, slPD, slQT, sbPS,dpOS, *STOP*] #
(3a)	[chOP] · [slCR, sbCR, dpRS, *inRS*]* · [slCR, sbCR, dpRS, slPD, slQT, sbPS,dpPS] · [swCC, dpPF]* · [inQO, *STOP*] #
(3b)	[chOP] · [slCR, sbCR, dpRS, *inRS*]* · [slCR, sbCR, dpRS, slPD, slQT, sbPS,dpPS] · [swCC, dpPF]* · [swCC, dpCF]

In this chapter, we defined the mappings to the semantic domain *sets of partial order sets*. In the next chapter, we will define the mappings to the alternative semantic domain of *nondeterministic finite state machines*. In Chapter 8, we will illustrate that both domains have their individual strengths and weaknesses.

7 Semantic Domain: Nondeterministic Finite State Machines

In this chapter, we define the second-level mapping to the semantic domain of *Nondeterministic Finite State Machines (nFSMs)*. We start by providing necessary definitions, relations and operations. Next, we specify an algorithm that generates an nFSM from a UC-LTS. We also define a semantic function that maps any generic task model to an nFSM.

7.1 Definitions

Similar to the definitions of the previous chapter, we assume the existence of a global typing function $\tau: \Sigma \rightarrow \{interaction, application, internal\}$ which associates each event name with a corresponding type. In contrast to posets, nFSMs are defined over event names. In what follows, we use the symbol Σ, possibly decorated with primes $(\Sigma', \Sigma'', \Sigma''', ...,)$ and/or subscripts $(\Sigma_1, \Sigma_2, \Sigma_3, ...,)$ to represent a subset of $EVENTNAME$. Further, we use the symbol Σ^* to denote a sequence of event names and the symbol Q to represent a sub-set of $STATE$.

7.1.1 Nondeterministic Finite State Machines (nFSM)

We start by reiterating the definition of a nondeterministic finite state machine (nFSM).

Definition 7.1 (Nondeterministic Finite State Machine). A *nondeterministic finite state machine* is defined as the following tuple: $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$, where

$Q \subseteq STATE$ is a finite set of states,

$\Sigma \subseteq EVENTNAME$ is a finite set of event names,

$\delta: Q \times (\Sigma \cup \{\lambda\}) \longrightarrow \mathbb{P}(Q)$ is the transition function, which yields for a given state and a given event name or λ[8] the set of (possible) states that can be reached,

q_0 is the initial state with $q_0 \in Q$,

$F_C \subseteq Q$ is the set of final states used by the sequential and iterative composition, and

$F_N \subseteq Q$ is the set of final states *not* used by the sequential and iterative composition.

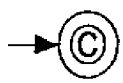
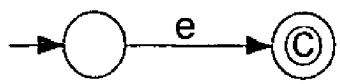
[8] λ denotes the empty sequence of event names

Note that in our definition, the set of final states is divided into two (possibly overlapping) subsets F_C and F_N. The semantics of both subsets will be given later in this chapter, when we define the binary operation *sequential composition* and the unary operation *iterative composition*.

Definition 7.2 (Skip and Singleton nFSM). We will use the symbol $Skip_{nFSM}$ to denote the *skip nFSM* $(\{q_0\}, \emptyset, \emptyset, q_0, \{q_0\}, \emptyset)$. If e is an event name, then we use the symbol e_{nFSM} to denote the *singleton nFSM* $(\{q_0, q_f\}, \{e\}, \{(q_0, e, q_f)\}, q_0, \{q_f\}, \emptyset)$.

Schematic representations of $Skip_{nFSM}$ and e_{nFSM} are given in Table 7.1. In what follows, we also use the symbol $\rightarrow\bigcirc$ to denote the initial state of an nFSM. Ⓒ and Ⓝ are used to represent states in F_C and states in F_N, respectively.

Table 7.1. Schematic Representations of $Skip_{nFSM}$ and e_{nFSM}

$Skip_{nFSM}$	e_{nFSM}
	

We define a set of auxiliary functions needed for the semantic mapping and the definition a refinement/equivalence relation between two FSMs.

Definition 7.3 (Extended Transition Function). The *extended transition function* $\delta^*: Q \times \Sigma^* \longrightarrow \mathbb{P}(Q)$ is defined in a standard way as:

$$\delta^*(q_i, w) = Q_j,$$

where Q_j is the set of possible states of the nFSM, having started in state q_i and after the sequence of inputs w. A formal recursive definition of the extended transition function can be found in [Hopcroft et al. 2007].

A partial trace of an nFSM is an event-name sequence w, for which the extended transition δ^*, when applied to the initial state q_0, yields a non-empty state set. Correspondingly, we define the set of *all partial traces* as follows:

Definition 7.4 (Set of all Partial Traces). Let $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ be an nFSM. We then define the function $Ptr: NFSM \longrightarrow \mathbb{P}(seq\ EVENTNAME)$ which yields the *set of all traces* of M as follows:

$$Ptr(M) = \{w \mid \delta^*(q_0, w) \neq \emptyset\}$$

Similar to the set of posets domain, we also define the notion of a *trace*. It is defined as a partial trace, which results in a state which is an element of F_C or F_N. Correspondingly, the set of all traces is a subset of the set of all partial traces and is defined as follows:

Definition 7.5 (Set of all Traces). Let $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ be an nFSM. We then define the function $Tr: NFSM \longrightarrow \mathbb{P}(seq\ EVENTNAME)$ which yields the *set of all traces* of M as follows:

$$Tr(M) = \{w \mid \delta^*(q_0, w) \cap (F_C \cup F_N) \neq \emptyset\}$$

The following three functions denote state properties and are needed for the equivalence and preorder definitions of Section 7.1.2. Each function is defined relative to a given nFSM.

Definition 7.6 (Initials). The *initials* of a given state q are the set of event names accepted by q. We define $initial: Q \longrightarrow \mathbb{P}(\Sigma)$ as follows:

$$initial(q) = \{a \in \Sigma \mid \exists q_t \in Q.\ (q, a, q_t) \in \delta^*\}$$

Note that a ambiguously denotes either an event name or the corresponding sequence of one event name.

Definition 7.7 (Out Set). The *Out Set* denotes the set of events that may be accepted by a given state q after having accepted an event-name sequence w. We define $out: Q \times \Sigma^* \longrightarrow \mathbb{P}(\Sigma)$ as follows:

$$out(q, w) = \bigcup_{q_i \in \delta^*(q,w)} initial(q_i)$$

Definition 7.8 (Acceptance Set). A set of event names Σ_A belongs to the *Acceptance Set* of a given state q after event-name sequence w, if and only if there is a state s reachable from q after having accepted w and Σ_A includes the set of event names accepted in s $(initial(s))$. Furthermore we require that Σ_A be included in the set of event names that may be accepted by q after having accepted w $(out(q,w))$. We define $Acc: Q \times \Sigma^* \longrightarrow \mathbb{P}(\mathbb{P}(\Sigma))$ as follows:

$$Acc(q,w) = \{\Sigma_A \mid \exists s \in \delta^*(q,w).inital(s) \subseteq \Sigma_A \subseteq out(q,w)\}$$

Our definition corresponds to the definition given in [Hennessy 1985]. The following properties of $Acc(q,w)$ can be derived from the definition [Khendek & Bochmann 1995]:

$$Acc(q,w) = \emptyset \iff \delta^*(q,w) = \emptyset$$

$$\forall \Sigma_1, \Sigma_2 \in Acc(q,w).\ \Sigma_1 \cup \Sigma_2 \in Acc(q,w)$$

$$\forall \Sigma_1, \Sigma_2 \in Acc(q,w)\ \exists \Sigma_3.\ \Sigma_1 \subseteq \Sigma_3 \subseteq \Sigma_2 \Longrightarrow \Sigma_3 \in Acc(q,w)$$

In order to illustrate the newly defined concepts, let us consider the nFSM depicted in Figure 7.1. It is formally defined as follows:

$M = (\{q_0, q_1, \ldots q_6\}, \{a,b,c,d,e\}, \delta, q_0, \{q_6\}, \emptyset)$ with

$\delta = \{(q_0, a, \{q_1\}), (q_1, \lambda, \{q_2, q_3\}), (q_2, b, \{q_4\}), (q_3, a, \{q_5\}), (q_4, a, \{q_6\}), (q_5, a, \{q_6\})\}$

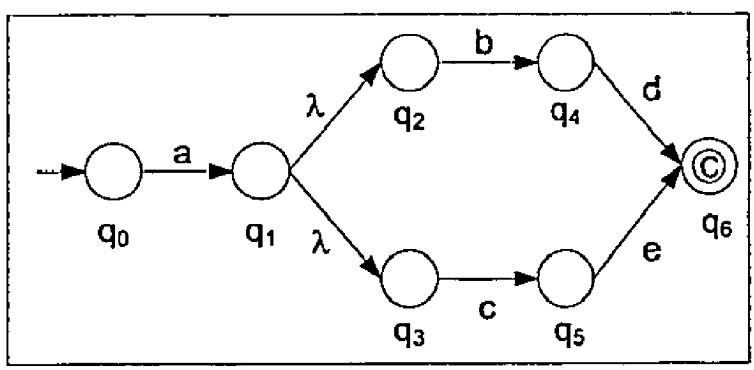


Figure 7.1. Example nFSM *M*

Using the formal definition of M as a starting point, we can derive the following properties. We start with the extended transition function. For the sake of conciseness

only a subset (i.e. the most interesting cases originating from the initial state) of δ^* is depicted.

$$\delta^* \supset \{(q_0, \langle\ \rangle, \{q_0\}), (q_0, \langle a\rangle, \{q_1, q_2, q_3\}), (q_0, \langle ab\rangle, \{q_4\}), (q_0, \langle ac\rangle, \{q_5\}),$$
$$(q_0, \langle abd\rangle, \{q_6\}), (q_0, \langle ace\rangle, \{q_6\})\}$$

The *initials* for all states are as follows.

$$initial = \{(q_0, \{a\}), (q_1, \{b, c\}), (q_2, \{b\}), (q_3, \{c\}), (q_4, \{d\}), (q_5, \{e\}), (q_6, \emptyset)\}$$

The *out sets* defined for q_0 and accepted sequences of event names (including the empty sequence λ) are given below.

$$out \supset \{(q_0, \langle\ \rangle, \{a\}), (q_0, \langle a\rangle, \{b, c\}), (q_0, \langle ab\rangle, \{d\}), (q_0, \langle ac\rangle, \{e\}),$$
$$(q_0, \langle abd\rangle, \emptyset), (q_0, \langle abc\rangle, \emptyset)\}$$

Using *initial* and *out* we can now derive the acceptance sets for state q_0 and any accepted sequence of event names.

$$Acc \supset \{(q_0, \langle\ \rangle, \{\{a\}\}), (q_0, \langle a\rangle, \{\{b, c\}, \{b\}, \{c\}\}), (q_0, \langle ab\rangle, \{\{d\}\}), (q_0, \langle ac\rangle, \{\{e\}\}),$$
$$(q_0, \langle abd\rangle, \{\emptyset\}), (q_0, \langle abc\rangle, \{\emptyset\})\}$$

Acceptance sets tend to be large in comparison to the amount of information they contain [Hennessy 1985]. Therefore, in the remainder of this thesis, instead of *saturated* acceptance sets, we give minimal subsets that are able to generate the respective acceptance sets based on the properties listed in Definition 7.8.

We conclude this example by providing the set of all partial traces and the set of all traces for M.

$$Ptr(M) = \{\langle\ \rangle, \langle a\rangle, \langle ab\rangle, \langle ac\rangle, \langle abd\rangle, \langle ace\rangle\}$$

$$Tr(M) = \{\langle abd\rangle, \langle ace\rangle\}$$

7.1.2 Relevant Equivalences and Preorders between nFSMs

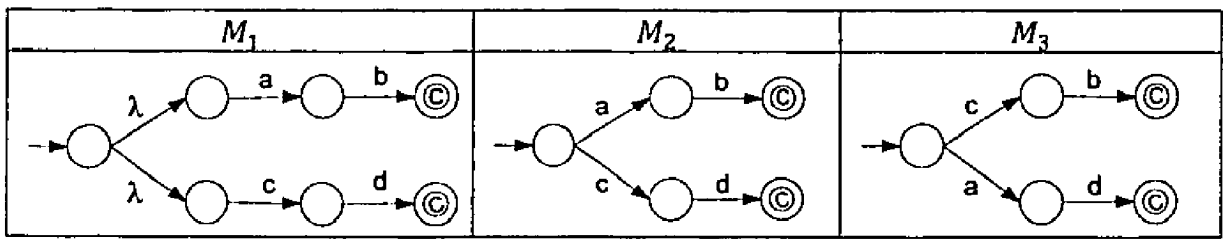
In Section 2.3.4.1, we noted that various equivalences and preorders have been defined for transition systems. In this section, we formally define *trace equivalence* and the preorder *deterministic reduction*. Both are required for formalizing the refinement

relations between use case and/or task models proposed in Chapter 3. The application and verification of the proposed equivalence and preorder relations are presented in detail in Chapter 9.

Definition 7.9 (Trace Equivalence). Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, F_{C1}, F_{N1})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, F_{C2}, F_{N2})$ be two nFSMs. M_1 and M_2 are *trace equivalent* $M_1 \equiv_{trace} M_2$ iff: $Tr(M_1) = Tr(M_2)$

In order to illustrate trace equivalence, let us consider the nFSMs depicted in Table 7.2. Clearly, and despite the different transition structure, the trace sets of M_1 and M_2 are identical $Tr(M_1) = Tr(M_2) = \{\langle a, b\rangle, \langle c, d\rangle\}$ and we conclude that M_1 and M_2 are trace equivalent. In contrast to this, we obtain the following trace set for M_3 $\{\langle c, b\rangle, \langle a, d\rangle\}$ and thus conclude that M_3 is *not* trace equivalent to M_1 (and M_2).

Table 7.2. Trace Equivalence Between nFSMs



Next we define the preorder *deterministic reduction*. It is similar to the reduction preorder proposed by Brinksma [1987]. We say that an nFSM M_2 is a *deterministic reduction* of an nFSM M_1, if M_2 has the same or less traces than M_1 *and* M_2 preserves all the nondeterminism in M_1.

Definition 7.10 (Deterministic Reduction). Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, F_{C1}, F_{N1})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, F_{C2}, F_{N2})$ be two FSMs. M_2 *deterministically reduces* M_1 ($M_2 \leq_{dred} M_1$) iff:

(1) $Tr(M_2) \subseteq Tr(M_1)$ (*safety property*)

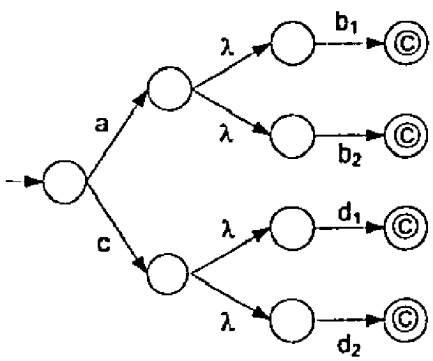
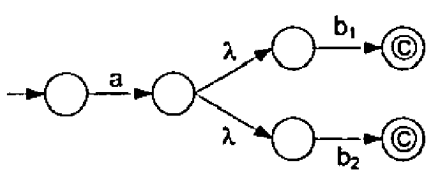
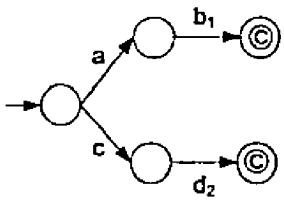
(2) $\forall w \in Ptr(M_2).\ |Acc_{M_1}(q_{0_1}, w)| = |Acc_{M_2}(q_{0_2}, w)|$ (*liveness property*)

The main differences to Brinksma's *reduction preorder* are twofold. (1) The *deterministic reduction preorder* requires trace inclusion (safety property) and not just partial-trace inclusion. (2) The introduction of new nondeterminism is not allowed, while

deterministic choices may be restricted to provide fewer alternatives. This is formalized by the liveness property which requires that for each partial trace, the cardinalities of the respective acceptance sets be the same.

In order to illustrate the deterministic reduction preorder, let us consider the nFSMs given in Table 7.3. We start with the verification of whether M_2 is a deterministic reduction of M_1. By analyzing the trace sets, we obtain $Tr(M_1) = \{\langle a, b_1\rangle, \langle a, b_2\rangle, \langle c, d_1\rangle, \langle c, d_2\rangle\}$ and $Tr(M_2) = \{\langle a, b_1\rangle, \langle a, b_2\rangle\}$. Clearly, $Tr(M_2) \subseteq Tr(M_1)$ and the safety property is satisfied.

Table 7.3. Deterministic Reduction Preorder Between nFSMs

M_1	M_2	M_3
		

For the verification of the liveness property, we analyze the acceptance sets of M_1 and M_2 for all partial traces of M_2 (given in Table 7.4). In particular, we verify that the respective acceptance sets have the same cardinalities. Clearly, in case of the empty trace ($\langle\ \rangle$) we have $|\{\{a, c\}\}| = |\{\{a\}\}| = 1$. Since the remaining acceptance sets are identical, we conclude that M_2 is a deterministic reduction of M_1.

Table 7.4. Acceptance Sets of M_1 and M_2 w.r.t. Partial Traces of M_2

Acceptance Sets of	M_1	M_2
$Acc(q_0, \langle\ \rangle) =$	$\{\{a, c\}\}$	$\{\{a\}\}$
$Acc(q_0, \langle a\rangle) =$	$\{\{b_1\}, \{b_2\}\}$	$\{\{b_1\}, \{b_2\}\}$
$Acc(q_0, \langle a, b_1\rangle) =$	$\{\emptyset\}$	$\{\emptyset\}$
$Acc(q_0, \langle a, b_2\rangle) =$	$\{\emptyset\}$	$\{\emptyset\}$

We conclude this sub-section with the verification of whether M_3, too, is a deterministic reduction of M_1. An analysis of the trace set of M_3 yields that $Tr(M_3) = \{\langle a, b_1\rangle, \langle c, d_2\rangle\}$. Clearly, $Tr(M_3) \subseteq Tr(M_1)$ and the safety property is satisfied. For the verification of the liveness property, we analyze the acceptance of M_1 and M_3 for all partial traces of M_3 (given in Table 7.5). For partial trace $\langle a\rangle$, we obtain the following acceptance sets. $Acc_{M_1}(q_0, \langle a\rangle) = \{\{b_1\}, \{b_2\}\}$ and $Acc_{M_2}(q_0, \langle a\rangle) = \{\{b_1\}\}$. Clearly, $|\{\{b_1\}, \{b_2\}\}| \neq |\{\{b_1\}\}|$ and we conclude that M_3 is *not* a deterministic reduction of M_1.

Table 7.5. Acceptance Sets of M_1 and M_3 w.r.t. Partial Traces of M_3

Acceptance Sets of	M_1	M_3
$Acc(q_0, \langle\ \rangle) =$	$\{\{a, c\}\}$	$\{\{a, c\}\}$
$Acc(q_0, \langle a\rangle) =$	$\{\{b_1\}, \{b_2\}\}$	$\{\{b_1\}\}$
$Acc(q_0, \langle c\rangle) =$	$\{\{d_1\}, \{d_2\}\}$	$\{\{d_2\}\}$
$Acc(q_0, \langle a, b_1\rangle) =$	$\{\emptyset\}$	$\{\emptyset\}$
$Acc(q_0, \langle a, d_2\rangle) =$	$\{\emptyset\}$	$\{\emptyset\}$

7.1.3 Composition Operations for nFSMs

In this section, we define a set of composition operations for nFSMs. All operations are required for the semantic mappings defined in the next section. We start with the binary operators, *sequential composition* ($\cdot$), *parallel composition* ($\|$) and *alternative composition* (#). Since we can rename states at will, each definition assumes, without loss of generality, that respective the state sets be disjoint. The operations are defined in terms of their corresponding representation as *acceptance graphs* (AGs) as introduced by [Hennessy 1985]. That is, for each operation, the involved nFSMs are *first* converted to AGs. *Then*, the actual operation is performed and the resulting AG is converted back to an nFSM.

Intuitively, an AG can be seen as an alternative representation of an nFSM. It corresponds to a *deterministic* finite state machine whose states are additionally labeled with acceptance sets. Due to the absence of nondeterminism, acceptance graphs are mathematically more tractable which simplifies the definition of the composition operations. A formal definition of an AG, the required conversion operations *nfsm* and *ag*, and the composition operations are given in Appendix D. Finally, we note that it has

been shown in [Khendek & Bochmann 1995] that the conversions to and from AGs preserve the trace sets and the acceptance sets of the original nFSMs.

Definition 7.11 (Binary Operations for nFSMs). Let $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, F_{C1}, F_{N1})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, F_{C2}, F_{N2})$ be nFSMs such that Q_1 and Q_2 are disjoint. We then define the *sequential composition* ($\cdot$), *parallel composition* ($\|$) and *alternative composition* (#) as follows:

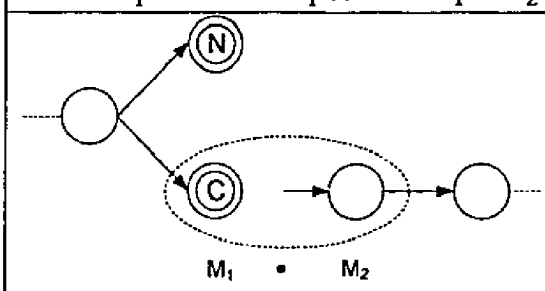
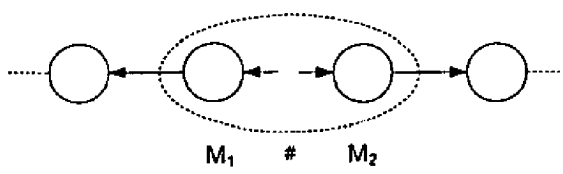
$$M_1 \cdot M_2 = nfsm(ag(M_1) \boxplus ag(M_2))$$

$$M_1 \# M_2 = nfsm(ag(M_1) \otimes ag(M_2))$$

$$M_1 \| M_2 = nfsm(ag(M_1) \| ag(M_2))$$

As schematically depicted in Table 7.6, the sequential composition of two nFSMs M_1 and M_2 merges the states in F_{C1} of M_1 with the initial state q_{0_2} of M_2. The merge is performed by applying the operation *sequential merge* ($\boxplus$) to the respective acceptance graphs of M_1 and M_2. Figure 7.2 portrays that, depending on the type of the initial events of the states in F_{C1} and state q_{0_2}, the merge operation is performed differently. If the initial events are of type *interaction*, then the states in F_{C1} and q_{0_2} are simply joined to one common state. If the initial events are of type *application*, then a new state is introduced which contains lambda transitions to all involved states in the merging process. Without loss of generality, the merge is undefined for states whose initial events have heterogeneous types.

Table 7.6. Sequential and Alternative Composition of nFSMs

Sequential Composition $M_1 \cdot M_2$	Alternative Composition $M_1 \# M_2$
	

As hinted by Table 7.6 (right hand side) the *alternative composition* of two nFSMs M_1 and M_2 merges the respective initial states q_{0_1} and q_{0_2}. Similar to the sequential

composition, the *merge* is performed on the basis of the respective acceptance graphs which are composed using the *choice merge* operation (⊗). The state merge itself depends on the types of the initial events of q_{0_1} and q_{0_2} (Figure 7.2).

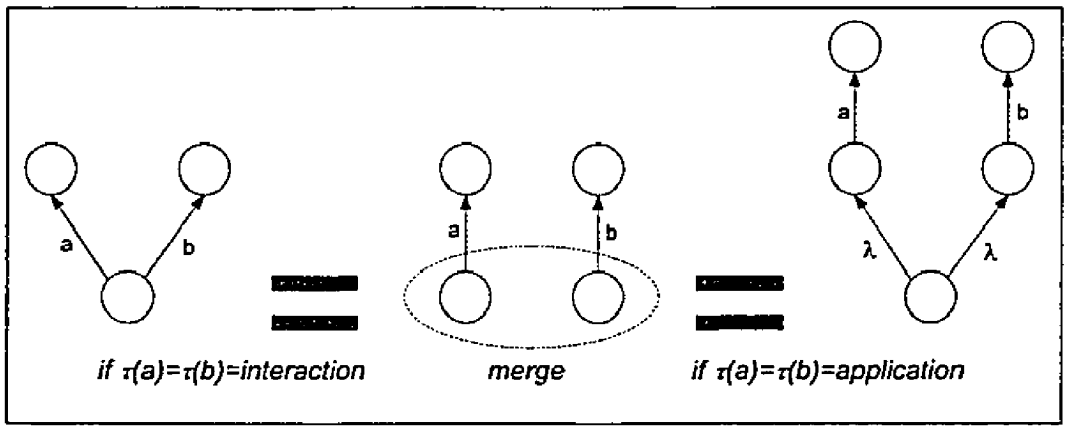


Figure 7.2. State Merge Depending on the Type of the Initial Events

As schematically depicted by Figure 7.3, the *parallel composition* of two nFSMs M_1 and M_2 constructs the so-called product machine that defines all possible "interleavings" of the transitions of M_1 and M_2. The actual construction is performed by composing the respective acceptance graphs. The obtained result is then transformed back to an nFSM representation.

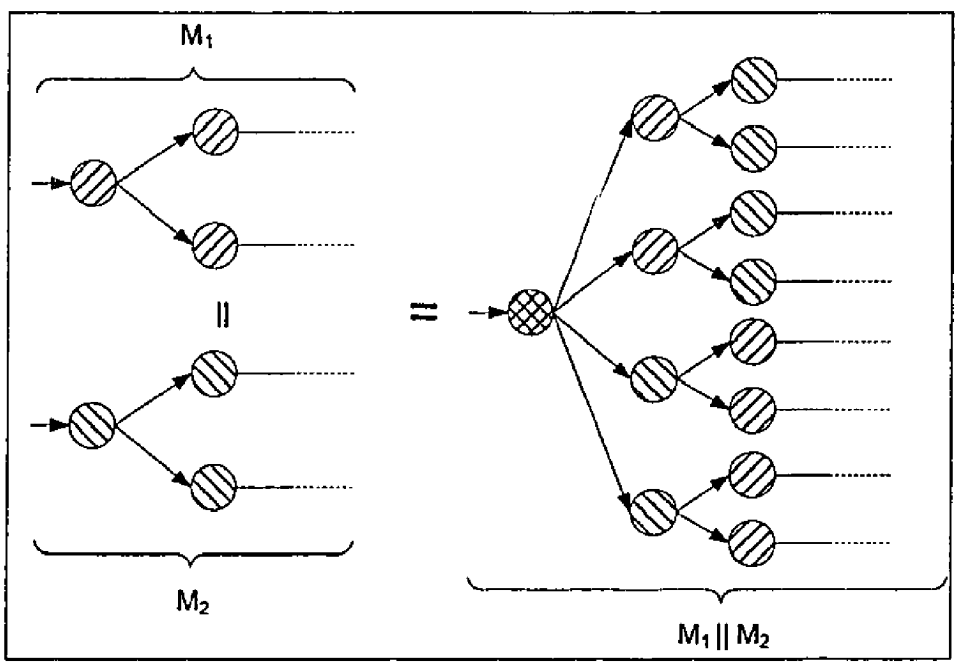


Figure 7.3. Parallel Composition of Two nFSMs

Also essential for the semantic mappings are the unary operations *iterative composition*, *closing* and *opening* of an nFSM. In contrast to the binary compositions, a conversion to acceptance graphs is not necessary.

Definition 7.12 (Iterative Composition). Let $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ be an nFSM. We then define the unary operation *iterative composition* $(*)$ as follows:
$M^* = (Q, \Sigma, \delta_*, q_0, F_C \cup \{q_0\}, F_N)$ where δ_* is defined as:

$$\delta_*(q,a) = \begin{cases} \delta(q,a) & q \notin F_C \\ \delta(q_0,a) \cup \delta(q,a) & q \in F_C \end{cases}$$

Intuitively, the *iterative composition* defines for each state in F_C, a new transition to each of the states directly reachable from q_0.

Definition 7.13 (Closing an nFSM). Let $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ be an nFSM. We then define the closing operation $close\ (M) = (Q, \Sigma, \delta, q_0, \emptyset, F_C \cup F_N)$.

Definition 7.14 Opening an nFSM). Let $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ be an nFSM. We then define the opening operation $open\ (M) = (Q, \Sigma, \delta, q_0, F_C \cup F_N, \emptyset)$.

The operations *closing* and *opening* have only an impact on the sets of final states F_C and F_N. In particular *closing* "shifts" all states in F_C to F_N, whereas *opening* "shifts" all states in F_N to F_C.

For each composition operation, we define the following trace properties, which will be needed in the next chapter, where we establish a formal correspondence between the set of posets semantics and the nFSM semantics.

Proposition 7.1. (Trace Properties of nFSM Operations). The nFSM operations: *sequential composition* $(\cdot)$, *parallel composition* $(\|)$, *alternative composition*$(\#)$, *iterative composition* $(*)$, *close* and *open* have the following trace properties:

Operation	Trace Property
$M_1 \cdot M_2$	$Tr(M_1 \cdot M_2) = \{x \hat{} y \mid x \in Tr(M_1) \wedge y \in Tr(M_2) \wedge \delta^*_{A_{M_1}}(q_0, x) \in F_{C_1}\} \cup \{x \mid x \in$

Operation	Trace Property
	$Tr(M_1) \wedge \delta^*_{A_{M_1}}(q_0, x) \in F_{N_1}\}$, where $\delta^*_{A_{M_1}}$ is the extended transition function of the corresponding acceptance graph $ag(M_1)$ of M_1.
$M_1 \parallel M_2$	$Tr(M_1 \parallel M_2) = \{x \mid\mid\mid y \mid x \in Tr(M_1) \wedge y \in Tr(M_2)\}$ (see Proposition 6.2 for a definition of $\mid\mid\mid$)
$M_1 \# M_2$	$Tr(M_1 \# M_2) = Tr(M_1) \cup Tr(M_2)$
M^*	$Tr(M^*) = \bigcup_{k=0}^{\infty} Tr(M)^k$, where $Tr(M)^k$ is defined as: $Tr(M)^k = \begin{cases} \{(\ )\}, & k = 0 \\ Tr(M), & k = 1 \\ \{x \hat{} y \mid x \in Tr(M) \wedge y \in Tr(M)^{k-1} \wedge \delta^*_{AG_{M_1}}(q_0, x) \in F_C\}, & k > 1 \end{cases}$
$close(M)$	$Tr(close(M)) = Tr(M)$
$open(M)$	$Tr(open(M)) = Tr(M)$

The proof of the proposition can be found in Appendix E.2.

Essential for the refinement definition of Chapter 9 are the operations *event hiding* and *event substitution*.

Definition 7.15 (Event Hiding). Let $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ be an nFSM and Σ_{hide} be a set of event names. We then define the operation *event hiding* as follows: $M \setminus \Sigma_{hide} = (Q, \Sigma', \delta', q_0, F_C, F_N)$ where:

$$\Sigma' = \Sigma - \Sigma_{hide}$$

$$\delta'(q, \lambda) = \delta(q, a), if\ a \in \Sigma_{hide}$$

$$\delta'(q, a) = \delta(q, a), if\ a \notin \Sigma_{hide}$$

Intuitively, *event hiding* replaces each transition that is triggered by an event name in Σ_{hide} with a corresponding lambda transition.

Definition 7.16 (Refinement Mapping). Let $M = (Q, \Sigma_{\text{nFSM}}, \delta, q_0, F_C, F_N)$ be an nFSM and $rmap: \mathbb{P}(\Sigma) \longrightarrow \Sigma$ be a valid event map with a finite domain, such that:

1. $dom(rmap)$ partitions Σ such that:
   a. $\forall \Sigma', \Sigma'' \in dom(rmap). (\Sigma' \cap \Sigma'' \neq \emptyset) \Longrightarrow (\Sigma' = \Sigma'')$
   b. $\bigcup_{\Sigma \in dom(rmap)} \Sigma = \Sigma_{\text{nFSM}}$

2. $\forall \Sigma \in dom(rmap). \forall n \in \Sigma. \tau(n) = \tau(rmap(\Sigma))$

Similar to Definition 6.21, the conditions ensure that (1a) the domain elements of $rmap$ are pairwise disjoint, (1b) the refinement mapping covers all events in Σ, and (2) for each mapping in $rmap$ all involved events have the same type.

We then define the operation *refinement mapping* as follows: $sub(M, map) = (Q, ran(rmap), \delta_{Sub}, q_0, F_C, F_N)$ where,

$$\delta_{Sub}(q, w) = \bigcup_{\Sigma \in rmap^{-1}(w)} \left(\bigcup_{\hat{w} \in \mathcal{P}(\Sigma)} \delta^*(q, \hat{w}) \right)$$

where $rmap^{-1}: \Sigma \rightarrow \mathbb{P}(\mathbb{P}(\Sigma))$ is the preimage of $rmap$ such that $rmap^{-1}(y) = \{x \mid rmap(x) = y\}$ and $\mathcal{P}(\Sigma)$ denotes the set of all permutations over Σ.

In order to illustrate the *refinement mapping*, let us consider the nFSM given on the left hand side of Table 7.7 and the following event map: $rmap = \{\{e_1\} \mapsto a_1, \{e_2, e_3\} \mapsto a_2\}$. With

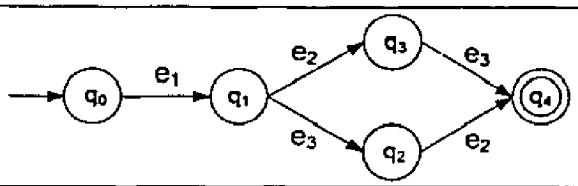
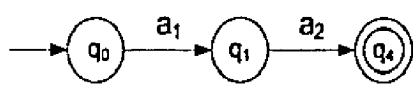
$rmap^{-1} = \{a_1 \mapsto \{\{e_1\}\}, a_2 \mapsto \{\{e_2, e_3\}\}\}$ we obtain the following new transitions:

$$\delta_{Sub}(q_0, a_1) = \delta^*(q_0, \langle e_1 \rangle) = \{q_1\}$$

$$\delta_{Sub}(q_1, a_2) = \delta^*(q_1, \langle e_1 e_2 \rangle) \cup \delta^*(q_1, \langle e_2 e_1 \rangle) = \{q_4\}$$

The resulting nFSM is depicted on the right hand side of Table 7.7.

Table 7.7. Event Substitution Example

M	$sub(M, \{\{e_1\} \mapsto a_1, \{e_2, e_3\} \mapsto a_2\}$
	 (only reachable states are shown)

7.2 Semantic Rules

In this section, we define the mappings from the intermediate semantic domains to nFSMs.

7.2.1 Mapping of UC-LTSs to nFSM

The semantic mapping from a UC-LTS to an nFSM is performed in two steps:

Flattening: Multi-event transitions are replaced by a set of single-event transitions

Transformation: The flattened UC-LTS is transformed into an nFSM

In what follows, we describe both steps in detail.

Flattening:

In a UC-LTS, transitions are triggered by sets of labels, whereas in an nFSM transitions are triggered by a single event name. In order to close this gap, we transform, in this step, a given UC-LTS U into a semantically equivalent flattened UC-LTS U_{flat} where all transitions are triggered by a set consisting of a single label. We know from Section 5.1 that, if the labeling set contains more than one label, no specific execution order exists between the corresponding use case steps. Hence, the behavior of U can be equivalently represented by a UC-LTS (U_{flat}) in which each multi-label transition is replaced by a set of interleaving single-label transitions which allow the respective use case steps to occur in an arbitrary order.

The derivation of U_{flat} from U requires a set of auxiliary definitions and functions, which are given next: We start with the so-called *product nFSM* over a given event name set Σ.

Definition 7.17 (Product nFSM). Let $\Sigma = \{e_1, e_2, \dots, e_n\}$ be a set of event names. We then define the *product nFSM* over Σ as follows: $M_\Sigma = e_{1_{nFSM}} \parallel e_{2_{nFSM}} \parallel \cdots \parallel e_{n_{nFSM}}$.

M_Σ accepts the elements in Σ in any order. As an example, Figure 7.4 portrays the product nFSM $M_{\{b,c\}}$.

Next, we define the *merge* operation. Given a UC-LTS, an nFSM and two states, *merge* results in a UC-LTS in which the transitions of the nFSM have been "inserted" in-between the two given states.

Definition 7.18 (UC-LTS – nFSM Merge). Let $U = (\Sigma, Q, q_0, F, \delta, \tau)$ be a UC-LTS and $M = (Q_{nFSM}, \Sigma_{\text{nFSM}}, \delta_{nFSM}, q_{nFSM}, F_{C_{nFSM}}, \emptyset)$ be an nFSM such that $\Sigma_{\text{fsm}} \subseteq \Sigma$ and Q,

Q_{nFSM} are disjoint. Further, let $q_s \in Q$, $q_t \in Q$ be a source and a target state. We then define the function $merge$: $UCLTS \times NFSM \longrightarrow UCLTS$ as follows:

$merge(U, M, q_s, q_t) = (\Sigma, Q_{merge}, q_0, F, \delta_{merge}, \tau)$ where,

$$Q_{merge} = Q \cup Q_{nFSM} - (\{q_{0_{nFSM}}\} \cup F_{C_{nFSM}})$$

$$\delta_{merge}(q, \Sigma) = \begin{cases} \delta(q, \Sigma) & q \neq q_s \wedge q \in Q \\ \delta_{nFSM}(q_{0_{nFSM}}, a) & q = q_s \wedge \Sigma = \{a\} \\ \delta_{nFSM}(q, a) & q \in Q_{nFSM} \wedge \Sigma = \{a\} \wedge \delta_{nFSM}(q, a) \notin F_{C_{nFSM}} \\ q_t & q \in Q_{nFSM} \wedge \Sigma = \{a\} \wedge \delta_{nFSM}(q, a) \in F_{C_{nFSM}} \end{cases}$$

The definition of the new transition function δ_{merge} distinguishes between the following four cases. (1) Transitions, originating from states of the original UC-LTS U (except for q_s) remain as specified by U. (2) The set of transitions originating from q_s are identified with the set of initial transitions in M. (3) Transitions originating from states in M (except for transitions that result in one of the final states) remain as specified in M. (4). Transitions in M that result in a final state are "re-directed" and result in the target insertion state q_t.

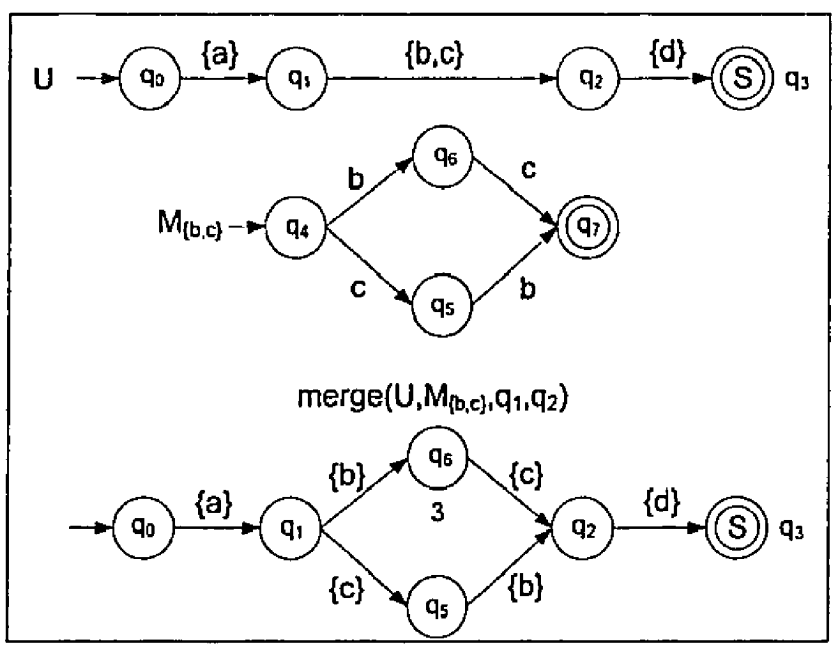


Figure 7.4. Sample Merge of an UC-LTS and an nFSM

Using the definitions of the *product nFSM* and *merge*, we define the function *flat*, which successively replaces each multi-label transition of a given UC-LTS with the set of transitions derived from the corresponding product nFSM until a fixpoint has been reached. The existence of such a fixpoint follows directly from the fact that each UC-LTS has only a finite number of transitions.

Definition 7.19 (Flattening a UC-LTS). Let $U = (\Sigma, Q, q_0, F, \delta, \tau)$ be a UC-LTS. We then define the function $flat: UCLTS \longrightarrow UCLTS$ as follows: $flat(U) = U_{flat}$, where U_{flat} is the fixpoint of the following sequence of equations:

$$U_{flat_0} = U$$

$$U_{flat_{n+1}} = \begin{cases} merge\left(U_{flat_n}, M_\Sigma, q_s, q_t\right) & (q_s, \Sigma, q_t) \in \delta \wedge |\Sigma| > 1 \\ U_{flat_n} & otherwise \end{cases}$$

An example is given in Figure 7.4, where the product nFSM $M_{\{b,c\}}$ is merged with UC-LTS U, in order to "flatten" the multi-label transition from state q_1 to state q_2.

Transformation:

In this step, a flattened UC-LTS is transformed into an nFSM. The singleton set transitions of the UC-LTS are replaced by transitions triggered by the corresponding individual event name.

Definition 7.20 (Mapping UC-LTS to an nFSM). Let $U = (\Sigma, Q, q_0, F, \delta, \tau)$ be a UC-LTS. We then define the semantic mapping to an nFSM $\mathcal{M}_{UcltsNfsm}: UCLTS \longrightarrow NFSM$ as follows:

$$\mathcal{M}_{UcltsNfsm}[\![U]\!] = M = \left(Q_{flat}, \Sigma_{flat}, \delta, q_{0_{flat}}, \delta, F_{flat}, \emptyset\right) \text{ where}$$

- $\delta(q, a) = \delta_{flat}(q, \{a\})$,
- $\left(\Sigma_{flat}, Q_{flat}, q_{0_{flat}}, F_{flat}, \delta_{flat}, \tau_{flat}\right) = flat(U)$,
- The global typing function τ_{nFSM} is defined as $\tau_{nFSM} = \tau$.

As an example, Figure 7.5 shows the nFSM representation of the "Order Product" use case. Since the "Order Product" UC-LTS only consists of single event transitions (Figure

5.4), the flattening step results in a UC-LTS which is equivalent to the original UC-LTS. During transformation, the transitions labeled with singleton sets are replaced by transitions triggered by respective individual event names.

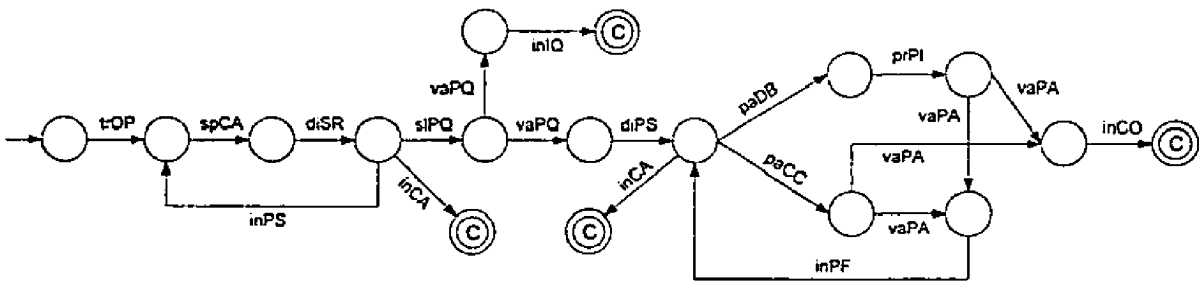


Figure 7.5. nFSM Representation of the "Order Product" Use Case

7.2.2 Mapping from GTM to nFSM

In this section we define the semantic mapping from generic task models to nFSMs.

Definition 7.21 (Mapping GTM to an nFSM). Let $G = (T, \psi, \tau)$ be a generic task model. We then define the mapping to an nFSM as follows:

$\mathcal{M}_{GtmNfsm}[\![(T, \psi, \tau)]\!] = \mathcal{M}_{GteNfsm}[\![\psi]\!]$ with the global typing function τ_{nFSM} defined as $\tau_{nFSM} = \tau$.

Similarly to the semantic mapping from GTE to sets of posets, an atomic generic task expression (denoted by α) is mapped into a singleton nFSM (see Definition 7.2). Composite generic task expressions are represented by more complex nFSMs, which result from the composition of the nFSMs representing sub-expressions.

Definition 7.22 (FSM Semantics of Generic Task Expressions). Let ψ, ρ be generic task expressions and α be an atomic task. We then define the mapping $\mathcal{M}_{GteNfsm}$ to an nFSM as follows:

$\mathcal{M}_{GteNfsm}[\![\alpha]\!] = \alpha_{nFSM}$

$\mathcal{M}_{GteNfsm}[\![\psi \gg \rho]\!] = \mathcal{M}_{GteNfsm}[\![\psi]\!] \cdot \mathcal{M}_{GteNfsm}[\![\rho]\!]$

$\mathcal{M}_{GteNfsm}[\![\psi [\,] \rho]\!] = \mathcal{M}_{GteNfsm}[\![\psi]\!] \# \mathcal{M}_{GteNfsm}[\![\rho]\!]$

$\mathcal{M}_{GteNfsm}[\![\psi ||| \rho]\!] = \mathcal{M}_{GteNfsm}[\![\psi]\!] \parallel \mathcal{M}_{GteNfsm}[\![\rho]\!]$

$\mathcal{M}_{GteNfsm}[\![[\psi]]\!] = \mathcal{M}_{GteNfsm}[\![\psi]\!] \# Skip_{nFSM}$

$$\mathcal{M}_{GteNfsm}[\![\psi^*]\!] = (\mathcal{M}_{GteNfsm}[\![\psi]\!])^*$$

$$\mathcal{M}_{GteNfsm}[\![stop(\psi)\ ]\!] = close(\mathcal{M}_{GteNfsm}[\![\psi]\!])$$

$$\mathcal{M}_{GteNfsm}[\![resume(\psi)\ ]\!] = open(\mathcal{M}_{GteNfsm}[\![\psi]\!])$$

We illustrate the semantic rules by mapping the "Order Product" generic task model, given in Table 5.4, to an nFSM. As depicted in Figure 7.6 the resulting nFSM has four final states, among which three are in F_N and represent the pre-mature abortion of the task model through a *STOP* task. One final state is in F_C and represents the successful completion of the task models. It is also noticeable that the system choices between the application tasks "Display Purchase Summary" (dpPS) and "Display Out of Stock" (dpOS) and the application tasks "Display Confirmation" (dpCF) and "Display Payment Failure" (dpPF) are modeled nondeterministically through the use of lambda transitions (λ). This corresponds to the state *merge* procedure discussed in Section 7.1.3 and illustrated in Figure 7.4.

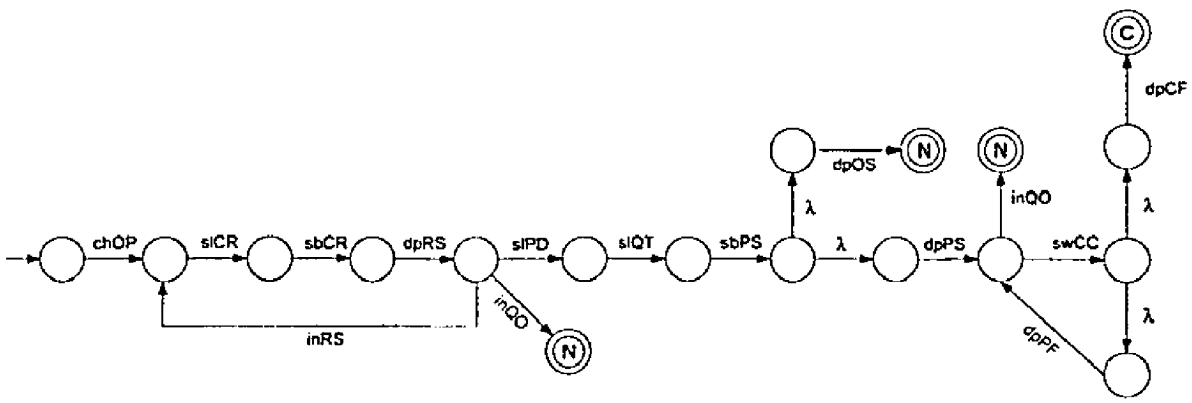


Figure 7.6. nFSM Representation of the "Order Product" Generic Task Model

8 Correspondence between Set of Posets and nFSM Semantics

In Chapters 6 and 7, we defined semantics for use case and task models based on sets of partially ordered sets (posets) and nondeterministic finite state machines (FSMs), respectively. In this chapter, we provide a general comparison of the two semantic domains in terms of their modeling capabilities. Having done that, specific to the semantics defined in this thesis, we establish a formal correspondence between the two semantics. In particular, we show that for a given UC-LTS or generic task model the nFSM semantics and set of posets semantics are trace equivalent. The results and insights gained in this chapter will support the next chapter, in which we formally model the different types of refinement defined for our integrated development methodology.

8.1 Comparing Semantic Domains

In this section, we provide a comparison of the semantic domains (*sets of posets* and *nFSMs*) in terms of their modeling capabilities and decidability of problems related to the integrated development methodology. From a generic point of view, it will be demonstrated that each semantic domain has its individual strengths and weaknesses, that may or may not lead to differences in the semantics for use case and task models.

Defined / Accepted Language Family: A (formal) language is defined as a set of strings drawn from some alphabet of symbols. More precisely, a formal language L over an alphabet Σ is a subset of Σ^*, i.e. $L \subseteq \Sigma^*$ [Hopcroft et al. 2007]. Within the context of this thesis, Σ is a set of event names. Taking this into account, we define the language accepted by a set of posets P as the set of all traces of P. Similarly, the language accepted by an nFSM M is the set of all traces of M.

From a theoretical point of view, sets of posets can define any (formal) language, whereas nFSMs can only be used to define regular languages[9] [Hopcroft et al. 2007]. The validity of the former statement draws from the fact that a set of posets may consist of an infinite number of posets. Hence, for any formal language L, one can find a set of posets

[9] A regular language can be accepted by a (nondeterministic) finite state machine, can be described by a regular expression and can be generated by a regular grammar.

P, that has the same cardinality as L and each entailed poset defines exactly one string in L (Figure 8.1).

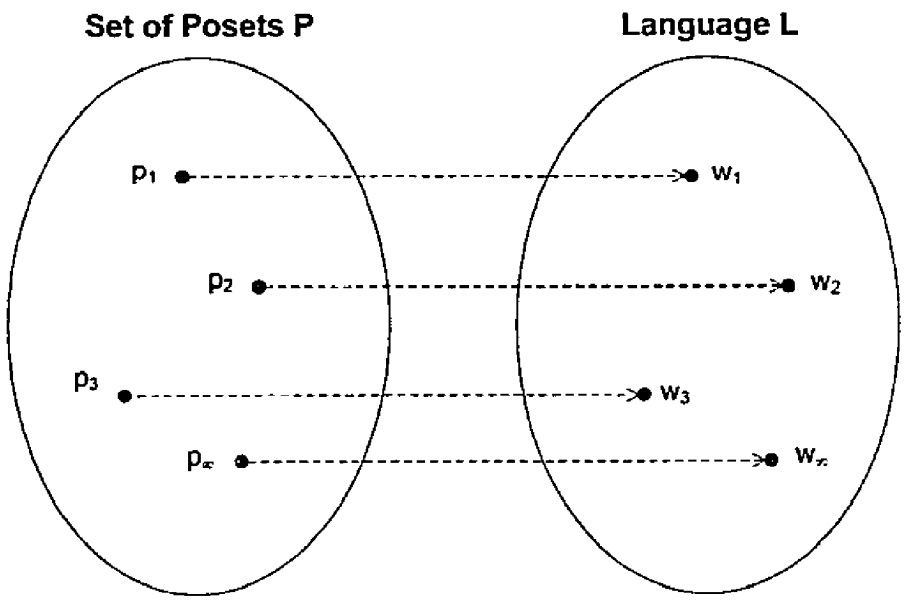


Figure 8.1. Definition of Language *L* through Enumeration by *P*

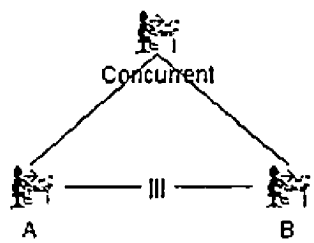
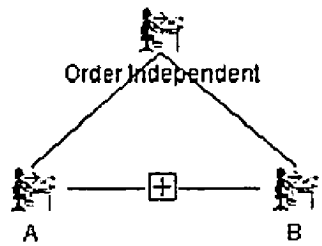
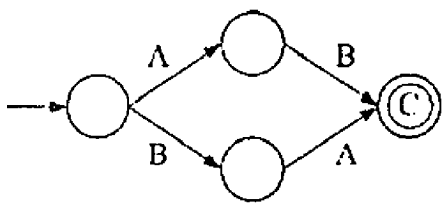
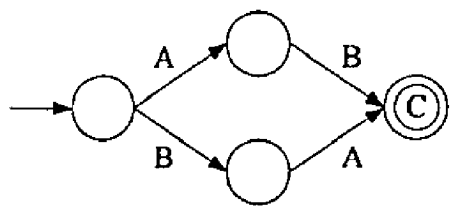
Within our formal framework, however, the limitation of nFSMs when compared to the sets of posets, in terms of its language-modeling capability is irrelevant. Neither UC-LTSs nor generic task models support specifications with trace sets that are outside the family of regular languages. This is proven in the next section, where we show that for every UC-LTS and generic task model, respectively, the corresponding set of posets and nFSMs are trace equivalent.

Model of Concurrency: In truly concurrent formalisms, a distinction is made between concurrent executions of actions and the choice between all possible orders of their execution [Marr 2006], whereas in interleaving models, the concept of true concurrency is omitted and concurrent system behavior is said to be equivalent to the (nondeterministic) choice of all possible interleavings. The set of posets formalism supports a non-interleaving (true) model of concurrency, whereas nFSMs support an interleaving model of concurrency.

Table 8.1 demonstrates the difference between the approaches by mapping two trace-equivalent task models to sets of posets and nFSMs, respectively. Both task models specify the execution of tasks 'A' and 'B'. In the task model on the left-hand side, tasks

'A' and 'B' are performed concurrently (|||), whereas in the task model on the right-hand side, 'A' and 'B' are performed sequentially in any order (⊞). In the interleaving semantics, the concurrent execution of tasks 'A' and 'B' and the choice between their sequential interleavings are identified and both task models are mapped to the same nFSM model. In the true concurrent semantics, a distinction between the two operators is made resulting in two different sets of posets.

Table 8.1. Difference Between Interleaving and Non-interleaving Semantics

ECTT		
GTE	$A \mid\mid\mid_{GTE} B$, with $\tau(A) = interaction$ $\tau(B) = interaction$	$(A \gg_{GTE} B) \square_{GTE} (B \gg_{GTE} A)$, with $\tau(A) = interaction$ $\tau(B) = interaction$
Set of Posets	$\{(\{A,B\},\{(A,A),(B,B)\})\}$	$\{(\{A,B\},\{(A,A),(B,B),(A,B)\}),$ $(\{A,B\},\{(A,A),(B,B),(B,A)\})\}$
nFSM		

Interleaving approaches do not support arbitrary refinement of events (or actions) into sub-events (or sub-actions). In an interleaving model, "exactly what is interleaved depends on which events one takes to be atomic" [Pratt 1986]. If a formerly atomic action is further refined, new interleavings among the sub-actions are introduced, which were not taken into account prior to the refinement. As it turns out, this observation has an implication for the verification of refinement between two specifications.

The refinement of use case steps or atomic tasks within the scope of a super-ordinate *Concurrent* step or *concurrency* operator (|||) into sub-steps or sub-tasks yields, in

general, a specification, which does not pass the refinement check (see Chapter 9) in the interleaving (nFSM) semantics. This is due to the fact that the required refinement mapping which relates events of the refining specification to events of the base specification does not take into account the different granularities of the interleavings defined in the base specification and the refining specification. This problem does not occur in the non-interleaving set of posets semantics where the concept of concurrency is fundamental.

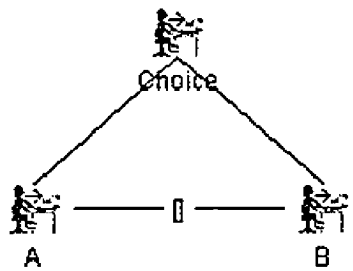
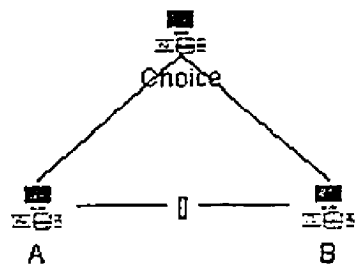
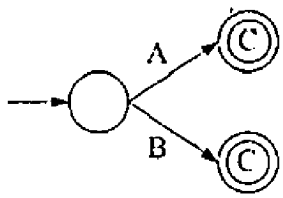
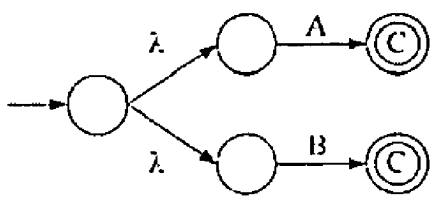
Support for Modeling Nondeterminism: A system exhibits nondeterminism, if two different copies of it may behave differently when given exactly the same inputs [Roscoe 2005]. As it turns out, the set of posets formalism (as defined in this thesis) has no intrinsic support to model nondeterministic systems, whereas in the nFSM formalism, nondeterminism can either be modeled by defining lambda (λ) transitions, or by defining a set of transitions that originate from the state and are triggered by the same event but result in different target states. The difference between the set of posets formalism and nFSM formalism in terms of supporting modeling nondeterminism clearly leads to differences in the semantics.

In the set of posets semantics, the alternative composition of tasks or use case steps results in a set of posets, in which each poset represents one alternative. As depicted on the right-hand side of Table 8.2, a distinction between user choices and system choices is *not* (and cannot be) made. In the nFSM semantics, the distinction between user and system choices is preserved. Alternative compositions are modeled using multiple transitions: one transition for each alternative. If the decision of which alternative to take is made by the user, the transitions are triggered by the respective events and the choice becomes deterministic. If the decision for one or the other alternative is made by the system, the transitions are triggered by λ and the choice becomes nondeterministic. The difference between these two choices is shown by examples in Table 8.2.

Within our integrated methodology, refinement cases 1-3 (Chapter 3) stipulate that in the refining model, only user choices may be restricted to provide the user with fewer alternatives to choose from. System choices, however, must remain the same. Clearly, since user choices and system choices are identified in set of posets semantics, this

refinement problem can only be adequately modeled and verified in the nFSM semantics. As a consequence, in the next chapter we will only provide nFSM formalizations for refinement cases 1-3. For all other refinements, we will provide set of posets and nFSM formalizations.

Table 8.2. Nondeterministic vs. Deterministic Choices

ECTT		
GTE	$A \,[]_{GTE}\, B$, with $\tau(A) = interaction$ $\tau(B) = interaction$	$A \,[]_{GTE}\, B$, with $\tau(A) = application$ $\tau(B) = application$
Set of Posets	$\{(\{A\},\{(A,A)\}),(\{B\},\{(B,B)\})\}$	$\{(\{A\},\{(A,A)\}),(\{B\},\{(B,B)\})\}$
nFSM		

Decidability Issues: Within our integrated development methodology, the following verification problems are of key interest: Scenario Equivalence (Refinement Cases 4-6), Scenario inclusion with restriction of user choices (Refinement Cases 1-3) and Membership (various applications). Let P_1 and P_2 be sets of posets and M_1 and M_2 be nFSMs, then these problems can be formalized in the set of posets semantics and nFSM semantics as depicted in Table 8.3.

Table 8.3. Relevant Problems Formalized in the Sets of Posets and nFSM Semantics

Problem	Set of Posets Formalization	nFSM Formalization
Scenario Equivalence	$Tr(P_1) = Tr(P_2)$ (undecidable)	$Tr(M_1) = Tr(M_2)$ (decidable)
Scenario Inclusion with Restriction of User Choices	N/A	$M_2 \leq_{dred} M_1$ (decidable)
Membership	$w \in Tr(P_1)$ (undecidable)	$w \in Tr(M_1)$ (decidable)

In general, when formalized in the set of posets semantics, all problems listed in Table 8.3 are undecidable, whereas, when formalized in the nFSM semantics, they are decidable. The former follows directly from the fact that a set of posets may entail an infinite number of posets, which makes an iterative analysis impossible. Proofs for the latter can be found in standard textbooks such as [Martin 2002; Linz 2006; Hopcroft et al. 2007]. Within the context of our development methodology, the observations of Table 8.3 have practical implications. *Only* in the nFSM semantics it is possible to provide comprehensive tool support for all listed verification problems.

A summary of the main findings of this section is given in Table 8.4. The results demonstrate that both, sets of posets and nFSMs, have their individual strengths and weaknesses with respect to the investigated modeling and decidability problems.

Table 8.4. Summary of the Comparison of the Semantic Domains

Criteria	Set of Posets	nFSM
Accepted Language Family	Any (formal) language	Regular languages, only
Model of Concurrency	Non-interleaving	Interleaving
Support for Nondeterminism	No	Yes
Decidability of rel. Problems	Undecidable	Decidable

8.2 Correspondence between Sets of Posets and nFSM Semantics

In this section, we establish a formal correspondence between the set of posets and nFSM semantics and prove that both are trace equivalent. As depicted in Figure 8.2, the proof consists of two parts. First, we show that the set of posets and nFSM semantics for

generic task models are trace equivalent. Second, we proof that this statement also holds for UC-LTSs.

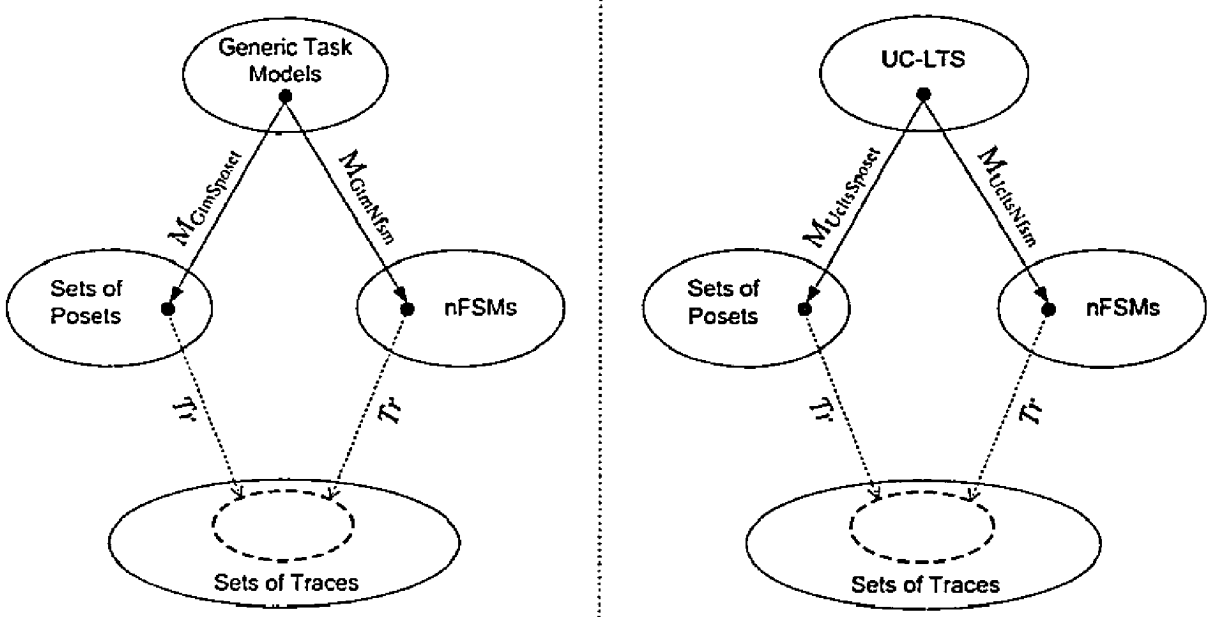


Figure 8.2. Outline of the Correspondence Proof

8.2.1 Trace Correspondence between $\mathcal{M}_{GtmSposet}$ and $\mathcal{M}_{GtmNfsm}$

In this section, we prove that for any given generic task model $G = (T, \psi, \tau)$, the results of the semantic have the same set of traces. It is sufficient to show that the semantic mapping of the generic task expression ψ results in a trace equivalent set of posets and nFSM. The proof is done by induction over the structure of generic task expressions. We start by defining an equivalence relation between sets of posets and nFSMs, which will be used as inductive hypothesis in the proof.

Definition 8.1 (Compositional Trace Equivalence). A set of posets P and an nFSM $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ are *compositionally trace equivalent* ($P \equiv_{CTR} M$) if:

(1) $Tr(P) = Tr(M)$

(2) $\forall x \in Tr(P).\left(\left(\exists p = (E_p, \leq_p) \in P. x \in Tr(p) \wedge STOP \in E_p\right) \Leftrightarrow (\delta_A^*(q_{A0}, x) \in F_{AN})\right)$

(3) $\forall x \in Tr(P).\left(\left(\exists p = (E_p, \leq_p) \in P. x \in Tr(p) \wedge STOP \notin E_p\right) \Leftrightarrow (\delta_A^*(q_{A0}, x) \in F_{AC})\right)$,

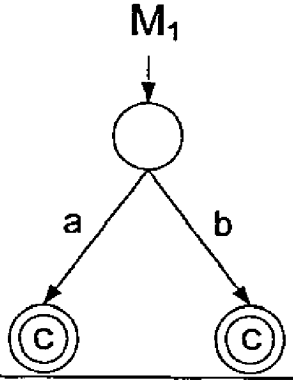
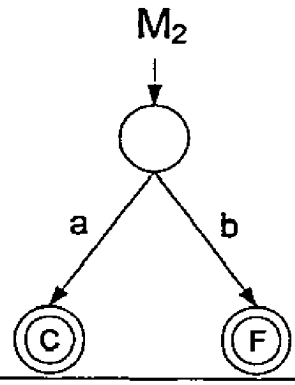
where δ_A^*, q_{A0}, F_{AN}, F_{AC} are the transition function, initial state, and final states of the acceptance graph representation of M. In what follows, for the sake of

simplicity, the terms "run of an nFSM" and "run of an acceptance graph" are used interchangeably.

Condition (1) ensures that P and M have the same set of traces. To satisfy the implication from left to right ($\Longrightarrow$) of conditions (2) and (3), it is required that each poset p in P corresponds to one or more runs (one for each trace of p) from the initial to one of the final states of M. If p contains the $STOP$ event, then the resulting state of the run is an element in F_{AN}. If the poset does not contain the $STOP$ event, then the resulting state is an element in F_{AC}. Contrary, to satisfy the implication form right to left ($\Longleftarrow$), it is required that each run of M corresponds to a trace of one or more posets of P. If the runs results in a state in F_{AC}, then P contains a poset that "implements" this trace and does not contain the $STOP$ event. If the run results in a state in F_{AN}, then P contains a posets that "implements" this trace and contains $STOP$.

In contrast to trace equivalence, $P \equiv_{CTR} M$ requires that P and M not only have the same traces *but* the same compositional properties. Why such a (strong) equivalence relation is needed, is motivated next: In general, although two sets of posets or nFSMs are trace equivalent, they may have different compositional properties. For illustrative purposes, let us consider the sets of posets P_1 and P_2 and the nFSMs M_1 and M_2 as depicted in Table 8.5. All four specifications have the same traces, but only P_1 and M_1, and P_2 and M_2 are also compositionally trace equivalent. In P_1 and M_1 each entailed poset and final state, respectively are used for the sequential composition, whereas in P_2 and M_2 only the poset which does not contain $STOP$ and the 'C' final state are used for the sequential composition.

Table 8.5. Trace Equivalent Sets of Posets / nFSMs

$P_1 = \{(\{a\},\{(a,a)\}),(\{b\},\{(b,b)\})\}$	$P_2 = \left\{\begin{matrix}(\{a\},\{(a,a)\}),\\(\{b,STOP\},\{(b,b),(STOP,STOP)\})\end{matrix}\right\}$
	

In order to prove trace correspondence between $\mathcal{M}_{GtmSposet}$ and $\mathcal{M}_{GtmNfsm}$ we make use of a set of supporting lemmas, which relate each of the composition operations (relevant for the semantic mapping) defined on sets of posets to a corresponding operation defined on nFSMs. Formal proofs for each of the lemmas follow more or less directly from Proposition 6.2 and Proposition 7.1 and can be found in Appendix E.3. The first lemma relates a singleton set of posets containing an atomic or empty poset to the atomic nFSM or the Skip nFSM.

Lemma 8.1. $(\{\alpha_{poset}\} \equiv_{CTR} \alpha_{nFSM}$ **and** $\{\emptyset_{poset}\} \equiv_{CTR} Skip_{nFSM})$**.** The set consisting only of an atomic poset holding a single event α (with $\alpha \neq STOP$) and the corresponding atomic nFSM are compositionally trace equivalent: $\{\alpha_{poset}\} \equiv_{CTR} \alpha_{nFSM}$. Similarly, the set consisting of only the empty poset and the Skip nFSM are compositionally trace equivalent: $\{\emptyset_{poset}\} \equiv_{CTR} Skip_{nFSM}$.

The following lemmas relate the binary operations *sequential*, *parallel* and *choice composition* defined for sets of posets to the corresponding binary operations defined for nFSMs.

Lemma 8.2. $(P_1 \cdot P_2 \equiv_{CTR} M_1 \cdot M_2)$**.** Let P_1, P_2 be sets of posets and M_1, M_2 be nFSMs such that $P_1 \equiv_{CTR} M_1$ and $P_2 \equiv_{CTR} M_2$, then the results of the respective *sequential compositions* are *compositionally trace equivalent.*

$$P_1 \cdot P_2 \equiv_{CTR} M_1 \cdot M_2$$

Lemma 8.3. ($P_1 \parallel P_2 \equiv_{CTR} M_1 \parallel M_2$). Let P_1, P_2 be sets of posets and M_1, M_2 be nFSMs such that $P_1 \equiv_{CTR} M_1$ and $P_2 \equiv_{CTR} M_2$, then the results of the respective *parallel compositions* are *compositionally trace equivalent.*

$$P_1 \parallel P_2 \equiv_{CTR} M_1 \parallel M_2$$

Lemma 8.4. ($P_1 \# P_2 \equiv_{CTR} M_1 \# M_2$). Let P_1, P_2 be sets of posets and M_1, M_2 be nFSMs such that $P_1 \equiv_{CTR} M_1$ and $P_2 \equiv_{CTR} M_2$, then the results of the respective *choice compositions* are *compositionally trace equivalent.*

$$P_1 \# P_2 \equiv_{CTR} M_1 \# M_2$$

Lemma 8.5 and Lemma 8.6 relate the unary operations *iteration*, *close* and *open* defined for sets of posets to the corresponding operations defined for nFSMs.

Lemma 8.5 ($P^* \equiv_{CTR} M^*$). Let P be a set of posets and M be an nFSM such that $P \equiv_{CTR} M$, then the results of the respective *iterative compositions* are *compositionally trace equivalent.*

$$P^* \equiv_{CTR} M^*$$

Lemma 8.6. ($close(P) \equiv_{CTR} close(M)$ **and** $open(P) \equiv_{CTR} open(P)$). Let P be a set of posets and M be an nFSM such that $P \equiv_{CTR} M$, then the results of the respective *close* ($close(P) \equiv_{CTR} close(M)$) and *open* ($open(P) \equiv_{CTR} open(M)$) operations are *compositionally trace equivalent.*

Using the lemmas we prove trace correspondence between $\mathcal{M}_{GtmSposet}$ and $\mathcal{M}_{GtmNfsm}$ by showing that for a given generic task expression ψ $Tr(\mathcal{M}_{GteSposet}[\![\psi]\!]) = Tr(\mathcal{M}_{GteNfsm}[\![\psi]\!])$ using structural induction over the set of composition operator for generic task expressions.

Theorem 8-1. Correspondence between $\mathcal{M}_{GtmSposet}$ and $\mathcal{M}_{GtmNfsm}$). Let $G = (T, \psi, \tau_{GTE})$ be a generic task model, then $\mathcal{M}_{GteSposet}[\![\psi]\!]$ and $\mathcal{M}_{GteNfsm}[\![\psi]\!]$ are trace equivalent.

$$Tr(\mathcal{M}_{GteSposet}[\![\psi]\!]) = Tr(\mathcal{M}_{GteNfsm}[\![\psi]\!])$$

PROOF: The proof is conducted by structural induction on the set of all generic task expressions GTE, following the recursive definition presented in Section 5.2.

BASIS: The basic elements of each generic task expression are atomic tasks (α). We show that $\mathcal{M}_{GteSposet}[\![\alpha]\!] \equiv_{CTR} \mathcal{M}_{GteNfsm}[\![\alpha]\!]$. According to Definition 6.24, $\mathcal{M}_{GteSposet}[\![\alpha]\!] = \{\alpha_{poset}\}$ and according to Definition 7.22 $\mathcal{M}_{GteNfsm}[\![\alpha]\!] = \alpha_{nFSM}$. It follows from Lemma 8.1 that $\mathcal{M}_{GteSposet}[\![\alpha]\!] \equiv_{CTR} \mathcal{M}_{GteNfsm}[\![\alpha]\!]$.

INDUCTION: We assume that the statement of the theorem is true for immediate sub-expressions of a given generic task expression ψ_1 and ψ_2 with less than i operators. Formally we assume that $\mathcal{M}_{GteSposet}[\![\psi_1]\!] \equiv_{CTR} \mathcal{M}_{GteNfsm}[\![\psi_1]\!]$ and $\mathcal{M}_{GteSposet}[\![\psi_2]\!] \equiv_{CTR} \mathcal{M}_{GteNfsm}[\![\psi_2]\!]$. Let ψ have i operators. There are 7 different cases depending on the form of ψ.

CASE 1: $\psi = \psi_1 \gg_{gte} \psi_2$. According to Definition 6.24 and Definition 7.22, ψ is semantically mapped to $\mathcal{M}_{GteSposet}[\![\psi_1]\!] \cdot \mathcal{M}_{GteSposet}[\![\psi_2]\!]$ and $\mathcal{M}_{GteNfsm}[\![\psi_1]\!] \cdot \mathcal{M}_{GteNfsm}[\![\psi_2]\!]$. From the inductive hypothesis we know that $\mathcal{M}_{GteSposet}[\![\psi_1]\!] \equiv_{CTR} \mathcal{M}_{GteNfsm}[\![\psi_1]\!]$ and $\mathcal{M}_{GteSposet}[\![\psi_2]\!] \equiv_{CTR} \mathcal{M}_{GteNfsm}[\![\psi_2]\!]$, which satisfies the pre-condition of Lemma 8.2, from which we conclude that $\mathcal{M}_{GteSposet}[\![\psi]\!] \equiv_{CTR} \mathcal{M}_{GteNfsm}[\![\psi]\!]$.

CASE 2: $\psi = \psi_1 [\,]_{gte} \psi_2$ and **CASE 3:** $\psi = \psi_1 |||_{gte} \psi_2$ are discharged in a similar manner to CASE 1 by first applying the semantic functions (Definition 6.24 and Definition 7.22) and drawing conclusions from Lemma 8.3 and Lemma 8.4.

CASE 4: $\psi = [\psi_1]$. According to Definition 6.24 and Definition 7.22 ψ is semantically mapped to $\mathcal{M}_{GteSposet}[\![\psi_1]\!] \# \{\emptyset_{poset}\}$ and $\mathcal{M}_{GteNfsm}[\![\psi_1]\!] \# Skip_{nFSM}$. From the

inductive hypothesis it follows that $\mathcal{M}_{GteSposet}[\![\psi_1]\!] \equiv_{CTR} \mathcal{M}_{GteNfsm}[\![\psi_1]\!]$ and from Lemma 8.1 we know that $\{\emptyset_{poset}\} \equiv_{CTR} Skip_{nFSM}$. Both assumptions form the pre-condition for Lemma 8.4, from which we conclude that $(\mathcal{M}_{GteSposet}[\![\psi_1]\!] \# \{\emptyset_{poset}\}) \equiv_{CTR} (\mathcal{M}_{GteNfsm}[\![\psi_1]\!] \# Skip_{nFSM})$.

CASE 5: $\psi = \psi_1^*$, **CASE 6:** $\psi = \boldsymbol{stop}_{gte}(\psi_1)$, and **CASE 7:** $\psi = \boldsymbol{resume}_{gte}(\psi_1)$ are readily discharged, after applying Definition 6.24 and Definition 7.22, from Lemma 8.5 and Lemma 8.6. ■

8.2.2 Trace Correspondence between $\mathcal{M}_{UcltsSposet}$ and $\mathcal{M}_{UcltsNfsm}$

As depicted in Figure 8.3, we prove this theorem by showing that the sets of all traces of $\mathcal{M}_{UcltsSposet}[\![U]\!]$ and $\mathcal{M}_{UcltsNfsm}[\![U]\!]$ are equal to the set of all traces of U. This will allow us to conclude that $\mathcal{M}_{UcltsSposet}[\![U]\!] = \mathcal{M}_{UcltsNfsm}[\![U]\!]$.

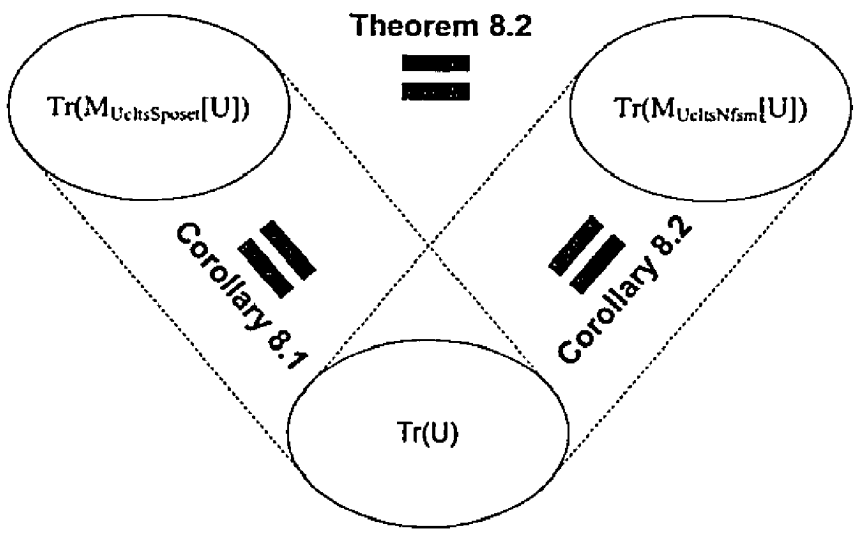


Figure 8.3. Correspondence Proof between $\mathcal{M}_{UcltsSposet}$ and $\mathcal{M}_{UcltsNfsm}$

We define the set of all traces of an UC-LTS as follows:

Definition 8.2 (Set of All Traces of a UC-LTS). The *set of all traces of a UC-LTS* $U = (\Sigma, Q, q_0, F, \delta, \tau)$ is defined as follows:

$$Tr(U) = \bigcup \{\mathcal{P}(X_1) \cdot \mathcal{P}(X_2) \cdot \ldots \cdot \mathcal{P}(X_n) \mid \delta(q_{i-1}, X_i) = q_i \text{ for } i \in \{1, \ldots, n\} \text{ and } q_n \in F\}$$

where $q_0, q_1, \ldots q_n$ is a run of U and $\mathcal{P}(X)$ computes all possible permutations of event sequences over the event set X.

Formally, $\mathcal{P}(X) = \{\langle x_1 \rangle\} \,|||\, \{\langle x_2 \rangle\} \,|||\, \ldots \,|||\, \{\langle x_n \rangle\}$, with $|\{x_k\}| = 1$ for $k \in \{1 \ldots n\}$, $\bigcup_{i=1}^{n} \{x_i\} = X$ and the $\{x_i\}'s$ are pairwise disjoint. The interleaving operator $|||$ applied to sets of event sequences computes the set of all possible interleavings of its operands. Formally, $X \,|||\, Y = \{x \,|||\, y \mid x \in X, y \in Y\}$. (See Proposition 6.2 for a formal definition of $|||$ applied to event sequences) The concatenation $(\cdot)$ of sets of events sequences (traces) is defined as follows: $w_1 \cdot w_2 = \{x \hat{} y \mid x \in w_1 \wedge y \in w_2\}$.

In the next sub-section, we show that the set of traces of a given UC-LTS U equals the set of traces of the set of posets resulting from the *LTS_to_SPO* algorithm applied to U.

8.2.2.1 Trace Equivalence of $Tr(U)$ and $\mathcal{M}_{UcltsSposet}[\![U]\!]$

The *LTS_to_SPO* algorithm (Section 6.2.1) transforms a given UC-LTS U into a set of posets P_{result} in three steps.

1. **Generation** of an initial *generalized UC-LTS* $U_{SPOSET-INIT}$, where each transition is augmented with a set of posets.

2. **Transformation** of $U_{SPOSET-INIT}$ to a minimized generalized UC-LTS $U_{SPOSET-MIN}$ through stepwise elimination of states.

3. **Extraction** of a set of posets expression P_{result} from $U_{SPOSET-MIN}$.

In what follows, we provide a lemma for each step, which proves that the set of traces of the input to the step equals the set of traces of the output of the step.

Generation:

Step 1 of the *LTS_to_SPO* algorithm associates a set of posets with each pair of states of U and creates a so-called generalized UC-LTS $U_{SPOSET} = (\Sigma, Q, q_0, F, \delta, \tau, SPO)$, where $SPO: (Q \times Q) \longrightarrow SPOSET$ is a function that associates a given pair of states with a set of posets. We define the set of all traces of U_{SPOSET} as the union over all paths from the initial state to a final state formed by sequentially composing the associated sets of posets along that path.

Definition 8.3 (Set of All Traces of a Generalized UC-LTS). The *set of all traces of a generalized UC-LTS* U_{SPOSET} is defined as follows:

$$Tr(U_{SPOSET}) = \bigcup \{Tr(SPO(q_0, q_1) \cdot SPO(q_1, q_2) \cdot \ldots \cdot SPO(q_{n-1}, q_n)) \mid \delta(q_{i-1}, X_i) = q_i \text{ for } i \in \{1, \ldots, n\} \text{ and } q_n \in F\}.$$

Lemma 8.7. $(Tr(U) = Tr(U_{SPOSET-INIT}))$**.** The set of all traces of a given UC-LTS U equals the set of all traces of the initial configuration of the corresponding generalized UC-LTS.

$$Tr(U) = Tr(U_{SPOSET-INIT})$$

PROOF: The lemma can be easily proven by showing that $\mathcal{P}(X) = Tr(\{(X, \{(l, l) \mid l \in X\})\})$ and $Tr(P_1) \cdot Tr(P_2) \cdot \ldots \cdot Tr(P_n) = Tr(P_1 \cdot P_2 \cdot \ldots \cdot P_n)$. The full proof can be found in Appendix E.3 ■.

Transformation:

During transformation, the initial generalized UC-LTS $U_{SPOSET-INIT}$ is incrementally transformed into a minimized generalized UC-LTS $U_{SPOSET-MIN}$. The following lemma proves that for each state-elimination step, the obtained reduced generalized UC-LTS U_{SPOSET}' has the same traces as its predecessor model $(Tr(U_{SPOSET}) = Tr(U_{SPOSET}'))$. This allows us to conclude that the resulting minimized generalized UC-LTS $U_{SPOSET-MIN}$ is trace equivalent to the initial generalized UC-LTS $U_{SPOSET-INIT}$.

Lemma 8.8. $(Tr(U_{SPOSET}) = Tr(U'_{SPOSET}))$**.** Let U_{SPOSET} be the automaton before the elimination and U'_{SPOSET} be the automaton after the elimination of state s with $s \in Q$ and $s \neq q_0$ and $s \notin F$, then the set of traces of U_{SPOSET} equals the set of traces of U'_{SPOSET}.

$$Tr(U_{SPOSET}) = Tr(U'_{SPOSET})$$

PROOF: The left of hand side of Figure 8.4 shows U_{SPOSET} and a generic state s about to be eliminated. We assume that s has k predecessor states $q_1, q_2, \ldots, q_k$ and m successor states $p_1, p_2, \ldots, p_m$. While it is possible that some of the q's are also p's we assume that

that s is not among the q's or p's, even when there is a loop from s to itself. Each transition is associated with a set of posets.

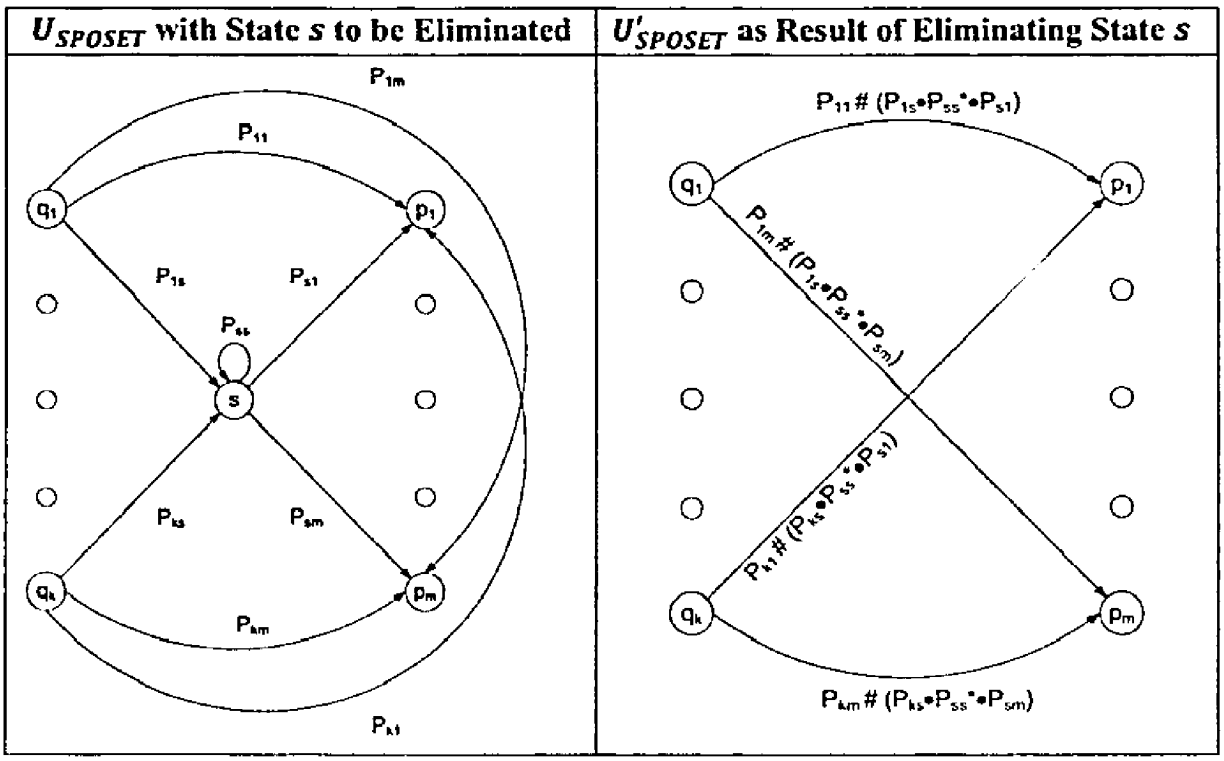


Figure 8.4. Elimination of State s (inspired from [Hopcroft et al. 2007])

As depicted, there are two possibilities to go from a state q_i to state p_j: (1) We go directly from q_i to p_j or (2) we go from q_i to s then loop around s zero or more times and finally go to p_j. Clearly, the set of all traces to go from q_i to p_j is the union of the set of traces of the posets associated with the direct walk from q_i to p_j ($Tr(P_{ij})$) with the set of traces associated with the detour through state s. The latter can be further decomposed into the set of traces from q_i to s ($Tr(P_{is})$) concatenated with the set of traces derived from looping around s (zero or more times) ($\bigcup_{k=0}^{\infty} Tr(P_{ss}^k)$) concatenated with the set of traces from s to p_j ($Tr(P_{sj})$). The complete set of traces for going from q_i to p_j, while possibly going through s is as follows:

$$Tr(P_{ij}) \cup \left(Tr(P_{is}) \cdot \bigcup_{k \geq 0}^{\infty} Tr(P_{ss}^k) \cdot Tr(P_{sj}) \right)$$

From Proposition 6.2 and the fact that none of the involved sets of posets contains $STOP$ it follows that the expression can be rewritten as follows:

$$Tr\left(P_{ij} \# (P_{is} \cdot P_{ss}^{*} \cdot P_{si})\right)$$

When we eliminate state s, all the paths that went through s no longer exist in U'_{SPOSET} (right hand side of Figure 8.4). If the set of all traces is not to change, we must modify the set of posets expression associated for each transition that goes directly from q_i to p_j such that it takes into account the set of poset expression associated with the transitions that went from q_i to p_j through s. This is done by Step 4 of the LTS_to_SPO algorithm. It associates the expression $P_{ij} \# (P_{is} \cdot P_{ss}^{*} \cdot P_{si})$ to the transition from q_i to p_j. If such a transition from q_i to p_j does not exist in the automaton, a new transition is added (*Step 5*) and P_{ij} evaluates to the empty set. Clearly, the set of all traces of U'_{SPOSET} equals the set of all traces of U_{SPOSET}. The design of Figure 8.4 is inspired from [Hopcroft et al. 2007], who tackled the related problem of obtaining an equivalent regular expression form a finite state machine. ◘

Extraction:

The iterative application of *Steps 4-6* finally results into an automaton $U_{SPOSET-MIN}$ of form depicted in Figure 8.5. It only consists of an initial and a set of final states. There are no outgoing transition from any of the final states. This follows directly from well formedness condition 3 for DSRG-style use case models (Definition 4.2). From the initial state q_0 a run of $U_{SPOSET-MIN}$ can loop around q_0 any number of times followed by a transition to one of the final states.

The following lemma proves that the set of traces obtained from the set of posets (P_{result}) extracted from steps 7-8 in the algorithm equals to the set of traces of $U_{SPOSET-MIN}$.

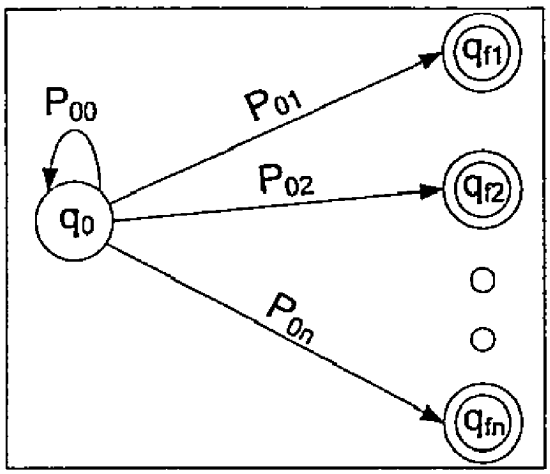


Figure 8.5. Minimized Generalized UC-LTS $U_{SPOSET-MIN}$

Lemma 8.9. $(Tr(U_{SPOSET-MIN}) = Tr(P_{result}))$**.** The set of traces of the minimized generalized UC-LTS $U_{SPOSET-MIN}$ equals the set of traces of the resulting set of posets P_{result}.

$$Tr(U_{SPOSET-MIN}) = Tr(P_{result})$$

PROOF: The correctness of the lemma becomes plausible by analyzing the trace set of $P_{result} = P_{00}^{*} \cdot (P_{01} \# P_{02} \# \ldots \# P_{0n})$ and the set of possible runs of $U_{SPOSET-MIN}$. A formal proof can be found in Appendix E.3■.

From Lemma 8.7, Lemma 8.8 and Lemma 8.9, we can now deduce that $Tr(U) = Tr(\mathcal{M}_{UcltsSposet}[\![U]\!])$ in a straightforward manner.

Corollary 8.1 $(Tr(U) = Tr(\mathcal{M}_{UcltsSposet}[\![U]\!]))$**.** Let U be a UC-LTS, then the set of traces of U equals the set of the traces of $\mathcal{M}_{UcltsSposet}[\![U]\!]$.

$$Tr(U) = Tr(\mathcal{M}_{UcltsSposet}[\![U]\!])$$

PROOF: It follows from Lemma 8.7 that $Tr(U) = Tr(U_{SPOSET-INIT})$. From Lemma 8.8 we know that for each state elimination step, the obtained reduced generalized UC-LTS U_{SPOSET}' has the same traces as its predecessor model $(Tr(U_{SPOSET}) = Tr(U_{SPOSET}'))$. This allows us to conclude that the resulting minimized generalized UC-LTS $U_{SPOSET-MIN}$ is trace equivalent to U $(Tr(U) = Tr(U_{SPOSET-MIN}))$. Finally it follows

from Lemma 8.9 that $Tr(U_{SPOSET-MIN}) = Tr(P_{result})$ and we conclude that $Tr(U) = Tr(\mathcal{M}_{UcltsSposet}[\![U]\!])$. ■

8.2.2.2 Trace Equivalence of UC-LTS U and $\mathcal{M}_{UcltsNfsm}[\![U]\!]$

The semantic mapping of U to an nFSM is performed in two steps.

1. **Flattening:** The UC-LTS U is transformed into a so-called flattened UC-LTS U_{flat}.
2. **Transformation:** The flattened UC-LTS U_{flat} is transformed into an nFSM M.

Similar to the previous section, we show for both steps that the set of traces of the input to a step equals the set of traces of the produced output.

Flattening:

During flattening, all multi-label set transitions of the input UC-LTS U are replaced by a sets of single-label transitions, which "emulate" the behavior of the multi-label transitions.

Lemma 8.10. (Trace Equivalence of U and U_{flat}): Let U be a UC-LTS and U_{flat} be the result of flattening U (Definition 7.19), then U and U_{flat} have the same set of traces.

$$Tr(U_{flat}) = Tr(U)$$

PROOF: U_{flat} is obtained by incrementally replacing all multi-label transition of U with sets of single-label transitions that define all possible interleavings. For a given multi-label transition (q_n, X, q_m), with $|X| \geq 2$, the set of all possible interleavings are obtained by computing the so-called product nFSM (M_X) and merging it with U (using q_n and q_m as source and target states, respectively). The set of traces of U_{flat} is obtained as follows:

$$Tr(U_{flat}) = \bigcup \{Tr(X_1) \cdot Tr(X_2) \cdot \ldots \cdot Tr(X_n) \mid \delta(q_{i-1}, X_i) = q_i \text{ for } i \in \{1,..,n\} \text{ and } q_n \in F\}$$

with

$$Tr(X) = \begin{cases} \{\langle e \rangle\} & if\ X = \{e\} \\ Tr(M_X) & if\ |X| \geq 2 \end{cases}$$

In order to prove that $Tr(U_{flat}) = Tr(U)$, we need to show that $Tr(X) = \mathcal{P}(X)$. Trivially $\{\langle e \rangle\} = \mathcal{P}(\{e\})$. To show that $Tr(M_X) = \mathcal{P}(X)$, for $|X| \geq 2$, we prove that $Tr(e_{1_{nFSM}} \parallel e_{2_{nFSM}} \parallel \cdots \parallel e_{n_{nFSM}}) = \mathcal{P}(X)$ with $\bigcup_{i=1}^{n}\{e_n\} = X$ and the $\{x_i\}'s$ are pairwise disjoint. From Definition 6.17 it follows that, in general, the parallel composition of sets of posets is associative and commutative (mainly because the *union* operation for sets is also commutative and associative). Taking this into account, together with the fact that $Tr(M_1) = \{\langle e_1 \rangle\}$, $Tr(M_2) = \{\langle e_2 \rangle\}$, ..., $Tr(M_1) = \{\langle e_n \rangle\}$, it follows from Proposition 7.1 that the set of all traces of M_X can be rewritten as follows: $Tr(M_X) = \{\langle e_1 \rangle\} \,|||\, \{\langle e_2 \rangle\} \,|||\, \ldots \,|||\, \{\langle e_n \rangle\}$. ■

Transformation:

The goal of this step is to obtain an nFSM M from a flattened UC-LTS, by replacing all *singleton set* transition of U_{flat} by *event* transitions.

Lemma 8.11. (Trace Equivalence of U_{flat} and Corresponding nFSM M). Let U_{flat} be a UC-LTS which contains only singleton set transitions, then U_{flat} and the corresponding nFSM M (as defined in Definition 7.20) have the same set of traces.

$$Tr(U_{flat}) = Tr(M)$$

PROOF: The correctness of the lemma follows directly from the definition of the set of all traces of an nFSM. A formal proof can be found in Appendix E.3 ■.

Using Lemma 8.10 and Lemma 8.11 we can now deduce that $Tr(U) = Tr(\mathcal{M}_{UcltsNfsm}[\![U]\!])$.

Corollary 8.2. (Trace Equivalence of UC-LTS U and $\mathcal{M}_{UcltsNfsm}[\![U]\!]$). Let U be a UC-LTS, then the set of all traces of U equals the set of all traces of $\mathcal{M}_{UcltsNfsm}[\![U]\!]$.

$$Tr(U) = Tr(\mathcal{M}_{UcltsNfsm}[\![U]\!])$$

PROOF: The proof follows directly from Lemma 8.10 and Lemma 8.11 and the fact that $Tr(U) = Tr(U_{flat})$ and $Tr(U_{flat}) = Tr(M)$, where $M = \mathcal{M}_{UcltsNfsm}[\![U]\!]$. ■

8.2.2.3 Trace Equivalence of $\mathcal{M}_{UcltsSposet}[\![U]\!]$ and $\mathcal{M}_{UcltsNfsm}[\![U]\!]$

The proof that, for a given UC-LTS U, the results of the semantic mappings to a set of posets and nFSM are trace equivalent, follows in a straightforward manner from Corollary 8.1 and Corollary 8.2.

Theorem 8-2. (Correspondence between $\mathcal{M}_{UcltsSposet}$ and $\mathcal{M}_{UcltsNfsm}$). Let U be a UC-LTS, then $\mathcal{M}_{UcltsSposet}[\![U]\!]$ and $\mathcal{M}_{UcltsNfsm}[\![U]\!]$ are trace equivalent.

$$Tr(\mathcal{M}_{UcltsSposet}[\![U]\!]) = Tr(\mathcal{M}_{UcltsNfsm}[\![U]\!])$$

PROOF:

From Corollary 8.1 we know that $Tr(\mathcal{M}_{UcltsSposet}[\![U]\!]) = Tr(U)$. From Corollary 8.2 we know that $Tr(\mathcal{M}_{UcltsNfsm}[\![U]\!]) = Tr(U)$. Thus, we conclude that $Tr(\mathcal{M}_{UcltsSposet}[\![U]\!]) = Tr(\mathcal{M}_{UcltsNfsm}[\![U]\!])$. ■

9 Refinement Between Use Case and Task Models

In this chapter we formally define the different types of refinement between use case and/or task models identified in Chapter 3. The formalizations are given in the nFSM semantics and, if applicable, in the set of posets semantics. We also introduce the *Use Case - Task Model - Verifier* tool, which we developed to automate the verification of refinement.

9.1 Preliminary Steps

Before we can formally compare two specifications for refinement, we have to ensure that both operate on the same event-name alphabet. In the following, we discuss two techniques to resolve alphabet conflicts, called *refinement mapping* and *event hiding*.

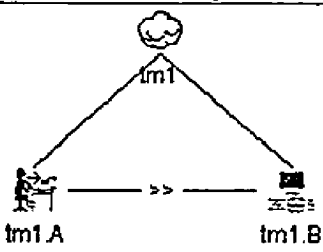
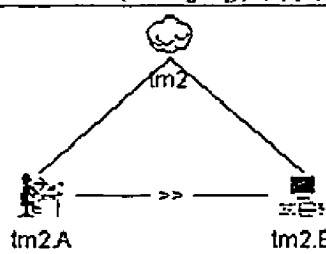
9.1.1 Refinement Mapping

A mapping from event names of the refining specification to event names of the base specification is termed *refinement mapping*. Let P be a set of posets and M be an nFSM specification, then a refinement mapping $rmap$ is applied to P and M as follows: $ref(P,rmap)$ and $ref(M,rmap)$. Note that the function ref for sets of posets and nFSMs is defined in Chapters 6 and 7, respectively.

Since the definition of a refinement mapping requires domain knowledge, it is assumed that $rmap$ is provided by the domain expert and is *not* inferred automatically. Based on our experiences, we identified three principle types of event refinement: *Isomorphic refinement*, *choice refinement* and *many-to-one refinement*. Each type is detailed in the following.

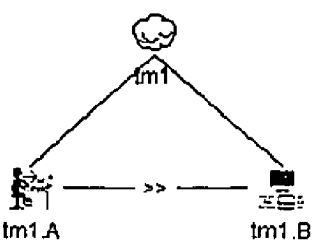
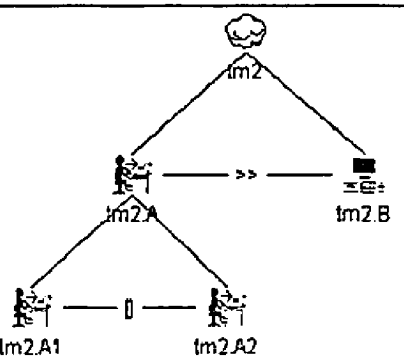
Isomorphic Refinement: An atomic element of the base specification is *isomorphically refined* by an atomic element of the refining specification, if and only if, there is a one-to-one mapping between the events representing both elements. In other words, both elements are identical, except for the given label or task name. Table 9.1 shows two examples of isomorphic refinement and the corresponding refinement mappings.

Table 9.1. Examples of Isomorphic Refinement

Base Model	Refining Model	Refinement Mapping
ucm1 1. do A ($ucm_1.s_A$) (inter.) 2. do B ($ucm_1.s_B$) (app.)	**ucm2** 1. do A ($ucm_2.s_A$) (inter.) 2. do B ($ucm_2.s_B$) (app.)	$\{ucm_2.s_A\} \mapsto ucm_1.s_A$ $\{ucm_2.s_B\} \mapsto ucm_1.s_B$
		$\{tm_2.A\} \mapsto tm_1.A$ $\{tm_2.B\} \mapsto tm_1.B$

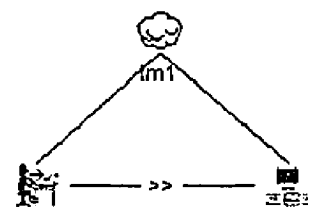
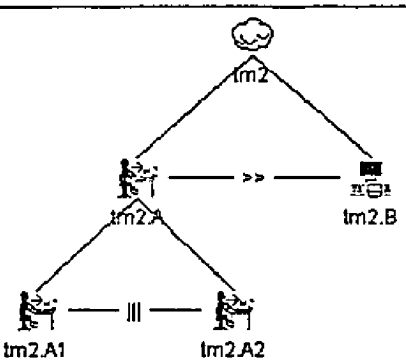
Choice Refinement: An atomic element of the base specification may be refined by a *set* of atomic elements which are alternatively composed by either the ECTT choice ([]) operator or a *Choice* use case step. Examples of a *choice refinement* are given in Table 9.2. As depicted, use case step $ucm_1.s_A$ and task $tm_1.A$ are choice-refined by use case steps $ucm_2.s_{A_1}$ and $ucm_2.s_{A_2}$ and tasks $tm_2.A_1$ and $tm_2.A_2$, respectively. (Additionally, step $ucm_1.s_B$ and task $tm_1.B$ are isomorphically refined by step $ucm_2.s_B$ and task $tm_2.B$.) In the corresponding refinement map, for each refining element, a mapping to the base element is defined. In contrast to isomorphic refinement, the mapping is *not* injective.

Table 9.2. Examples of Choice Refinement

Base Model	Refining Model	Refinement Mapping
ucm1 1. do A ($ucm_1.s_A$) (inter.) 2. do B ($ucm_1.s_B$) (app.)	**ucm2** 1. *PA performs choice of:* 1.1 do A.1 ($ucm_2.s_{A_1}$) (inter.) 1.2 do A.2 ($ucm_2.s_{A_2}$) (inter.) 2. do B ($ucm_2.s_B$) (app.)	$\{ucm_2.s_{A_1}\} \mapsto ucm_1.s_A$ $\{ucm_2.s_{A_2}\} \mapsto ucm_1.s_A$ $\{ucm_2.s_B\} \mapsto ucm_1.s_B$
		$\{tm_2.A_1\} \mapsto tm_1.A$ $\{tm_2.A_2\} \mapsto tm_1.A$ $\{tm_2.B\} \mapsto tm_1.B$ (non-injective mappings)

Many-To-One Refinement: A previously atomic element is refined into a set of sub-elements. In contrast to choice refinement, the execution of the entirety of sub-elements resembles the execution of the base element. Table 9.3 provides examples of *many-to-one* refinement. Use case step $ucm_1.s_A$ is refined into sub-steps $ucm_2.s_{A_1}$ and $ucm_2.s_{A_2}$. Similarly, task $tm_1.A$ is refined to sub-tasks $tm_2.A_1$ and $tm_2.A_2$. In each case, the execution of both sub-steps or sub-tasks corresponds to the execution of the respective base step or task. In the refinement map, a set containing all refining elements is mapped to the respective base element.

Table 9.3. Examples of Many-to-One Refinement

Base Model	Refining Model	Refinement Mapping
ucm1 1. do A ($ucm_1.s_A$) (inter.) 2. do B ($ucm_1.s_B$) (app.)	**ucm2** 1. *PA performs the following steps in any order:* 1.1 do A.1 ($ucm_2.s_{A_1}$) (inter.) 1.2 do A.2 ($ucm_2.s_{A_2}$) (inter.) 2. do B ($ucm_2.s_B$) (app.)	$\{ucm_2.s_{A_1}, ucm_2.s_{A_2}\} \mapsto ucm_1.s_A$ $\{ucm_2.s_B\} \mapsto ucm_1.s_B$
		$\{tm_2.A_1, tm_2.A_2\} \mapsto tm_1.A$ $\{tm_2.B\} \mapsto tm_1.B$

9.1.2 Event Hiding

The second technique that can be used to unify the event-name alphabet of two specifications is *event hiding*. According to the integrated development methodology, use case models are used to document functional requirements while task models specify UI requirements and design details. As such, they abstract from internal system actions, which are irrelevant for UI design. Hence, in order to compare use case and task models for refinement, we have to abstract the use case model from all internal system steps. In the set of posets semantics, this is achieved by hiding all events of type *internal*.

Formally, if P is a set of posets then $P' = P \setminus \{e \mid \tau(e) = internal\}$ denotes the corresponding set of posets without events of type *internal*. In the nFSM semantics, the abstraction from internal events is achieved by replacing all transitions that are triggered by internal events with "lambda" transitions. Formally, if M is an nFSM, then $M' = M \setminus \{e \mid \tau(e) = internal\}$ is the corresponding nFSM without internal events.

9.2 Formal Definition of Refinement

Having discussed two techniques that can be used to make the base and refining specification comparable, we proceed to the formal definition of refinement between use case- and/or task models. As mentioned in Chapter 3, we distinguish between six different cases.

Definition 9.1 (Refinement between Use Case Models and/or Task Models) Let UCM_1 and UCM_2 be (well formed) use case models, TM_{R_1} and TM_{R_2} be (well formed, recursion-free) requirements-level task models, TM_{D_1} and TM_{D_2} be (well formed, recursion-free) design-level task models, and for $i \in \{1,2\}$ let

$$P_{UCM_i} = \mathcal{M}_{UcItsSposet}[\![\mathcal{M}_{UcUcIts}[\![UCM_i]\!]]\!] \qquad M_{UCM_i} = \mathcal{M}_{UcItsNfsm}[\![\mathcal{M}_{UcmUcIts}[\![UCM_i]\!]]\!]$$

$$P_{TM_{R_i}} = \mathcal{M}_{GtmSposet}[\![\mathcal{M}_{EcttGtm}[\![TM_{R_i}]\!]]\!] \qquad M_{TM_{R_i}} = \mathcal{M}_{GtmNfsm}[\![\mathcal{M}_{EcttGtm}[\![TM_{R_i}]\!]]\!]$$

$$P_{TM_{D_i}} = \mathcal{M}_{GtmSposet}[\![\mathcal{M}_{EcttGtm}[\![TM_{D_i}]\!]]\!] \qquad M_{TM_{D_i}} = \mathcal{M}_{GtmNfsm}[\![\mathcal{M}_{EcttGtm}[\![TM_{D_i}]\!]]\!]$$

be the corresponding semantic representations. Furthermore, let $rmap: \mathbb{P}(\Sigma) \longrightarrow \Sigma$ be a valid refinement mapping (w.r.t. Definition 6.21 and Definition 7.16). We then define *refinement between use case and/or task models* as follows:

Table 9.4. Refinement Formalizations in the nFSM and Set of Posets Semantics

1. $UCM_1 \sqsubseteq UCM_2$ (Use Case Model – Use Case Model Refinement)	
Set of Posets Formalization	*N/A*
nFSM Formalization	$M_{UCM_1} \geq_{dred} ref(M_{UCM_2}, rmap)$
2. $UCM_1 \sqsubseteq TM_{R_2}$ (Use Case Model – Req. Task Model Refinement)	
Set of Posets Formalization	*N/A*
nFSM Formalization	$(M_{UCM_1} \setminus \{e \mid \tau(e) = internal\}) \geq_{dred} ref(M_{TM_{R_2}}, rmap)$

3. $TM_{R_1} \sqsubseteq TM_{R_2}$ (Req. Task Model – Req. Task Model Refinement)	
Set of Posets Formalization	*N/A*
nFSM Formalization	$M_{TM_{R_1}} \geq_{dred} ref\left(M_{TM_{R_2}}, rmap\right)$
4. $TM_{R_1} \sqsubseteq TM_{D_2}$ (Req. Task Model – Design Task Model Refinement	
Set of Posets Formalization	$P_{TM_{R_1}} \equiv_{trace} ref\left(P_{TM_{D_2}}, rmap\right)$
nFSM Formalization	$M_{TM_{R_1}} \equiv_{trace} ref\left(M_{TM_{D_2}}, rmap\right)$
5. $UCM_1 \sqsubseteq TM_{D_2}$ (Use Case Model – Design Task Model Refinement)	
Set of Posets Formalization	$\left(P_{UCM_1} \setminus \{e \mid \tau(e) = internal\}\right) \equiv_{trace} ref\left(P_{TM_{D_2}}, rmap\right)$
nFSM Formalization	$\left(M_{UCM_1} \setminus \{e \mid \tau(e) = internal\}\right) \equiv_{trace} ref\left(M_{TM_{D_2}}, rmap\right)$
6. $TM_{D_1} \sqsubseteq TM_{D_2}$ (Design Task Model – Design Task Model Refinement)	
Set of Posets Formalization	$P_{TM_{D_1}} \equiv_{trace} ref\left(P_{TM_{D_2}}, rmap\right)$
nFSM Formalization	$M_{TM_{D_1}} \equiv_{trace} ref\left(M_{TM_{D_2}}, rmap\right)$

Note that at the semantic level, all six refinement relations can be expressed by either using trace equivalence ($\equiv_{trace}$) and deterministic reduction ($\geq_{dred}$). Both[10] relations are formally defined in Chapters 6 and 7. Refinements cases 4 and 6 even have identical formalizations. Depending on the nature of the artifacts involved, a preliminary step of *event hiding* may be necessary. For each refinement case, a valid refinement mapping needs to be provided by the requirements engineer. Using the examples of Chapter 3, we now discuss the formalizations of each refinement case in detail.

Case 1: Use Case Model – Use Case Model Refinement: In Chapter 3, we stated that a refinement between use case models is deemed valid, if the refined use case model does not allow more scenarios than its base model *and* if all (nondeterministic) system choices are preserved. As portrayed in Table 9.4, the refinement can only be formalized in the nFSM semantics. A set of posets formalization is not possible, since a distinction

[10] $\geq_{dext}$ is only supported by the nFSM formalism.

between deterministic and nondeterministic choices cannot be made (as discussed in Chapter 8).

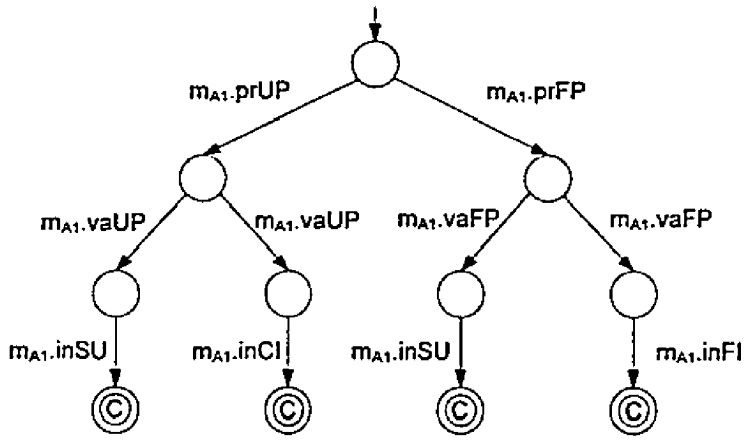


Figure 9.1. nFSM $M_{m_{A1}}$ Representing "Login" Use Case m_1

For illustrative purposes, let us recall the "Login" use case m_{A1} and its refining use case m_{A2} (as given in Section 3.2). Figure 9.1 and Figure 9.2 (a) show the corresponding nFSM formalizations. For the required refinement mapping we note that use case steps $m_{A1}.prUP$, $vaUP, m_{A1}.inSU$, and $m_{A1}.inSU$ of m_{A1} are *isomorphically* refined by steps $m_{A2}.prUP$, $m_{A2}.cvUP$, and $m_{A2}.inSU$ of m_{A2}. Extension step $m_{A1}.inCI$ of m_{A1} is *choice refined* by steps $m_{A2}.inCF$ and $m_{A2}.inCI$ of m_{A2}. The refinement mapping is formalized as follows:

$$rmap = \begin{Bmatrix} \{m_{A2}.prUP\} \mapsto m_{A1}.prUP, \{m_{A2}.cvUP\} \mapsto m_{A1}.vaUP, \{m_{A2}.inSU\} \mapsto m_{A1}.inSU, \\ \{m_{A2}.inCF\} \mapsto m_{A1}.inCI, \{m_{A2}.inCI\} \mapsto m_{A1}.inCI \end{Bmatrix}$$

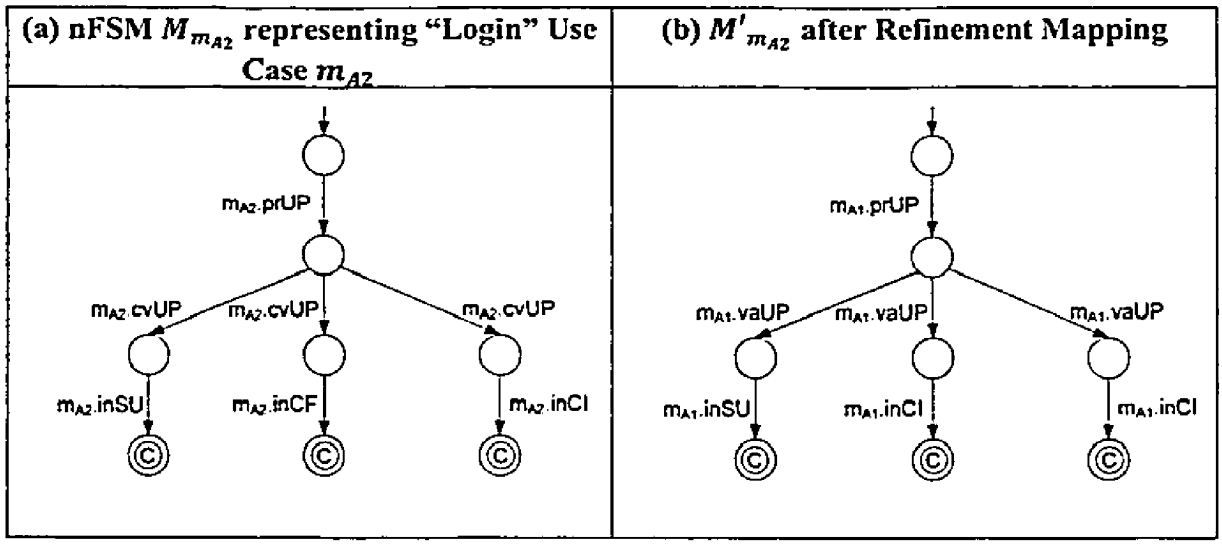


Figure 9.2. nFSMs of "Login" UC (m_{A2}) before (a) and after (b) Refinement Mapping

In order to validate that use case m_{A2} is a proper refinement of use case m_{A1}, we need to verify that $M_{m_{A1}} \geq_{dred} M'_{m_{A2}}$, where $M'_{m_{A2}}$ is the nFSM obtained by applying the refinement mapping to $M_{m_{A2}}$ (Figure 9.2 (b)). We start with the verification of the safety property (trace inclusion). With

$Tr(M_{m_{A1}}) = \{$
$\langle m_{A1}.prUP, m_{A1}.vaUP, m_{A1}.inSU\rangle,$
$\langle m_{A1}.prUP, m_{A1}.vaUP, m_{A1}.inCI\rangle,$
$\langle m_{A1}.prUP, m_{A1}.vaUP, m_{A1}.inFI\rangle,$
$\}$

$Tr(M'_{m_{A2}}) = \{$
$\langle m_{A1}.prUP, m_{A1}.vaUP, m_{A1}.inSU\rangle,$
$\langle m_{A1}.prUP, m_{A1}.vaUP, m_{A1}.inCI\rangle,$
$\}$

we can easily see that $Tr(M'_{m_{A2}}) \subseteq Tr(M_{m_{A1}})$. Hence, the first condition is fulfilled and we continue with the verification of the liveness property.

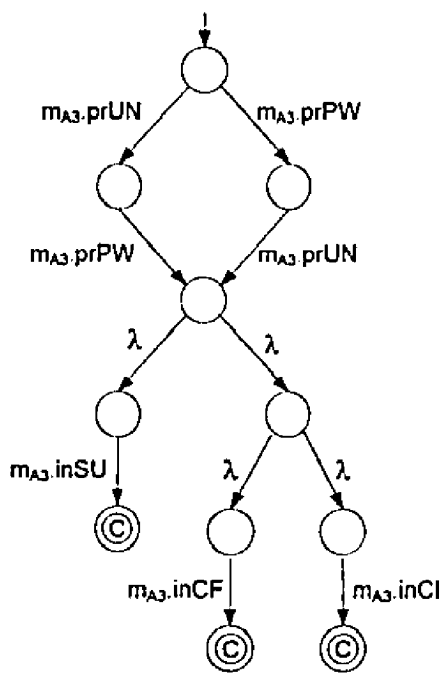
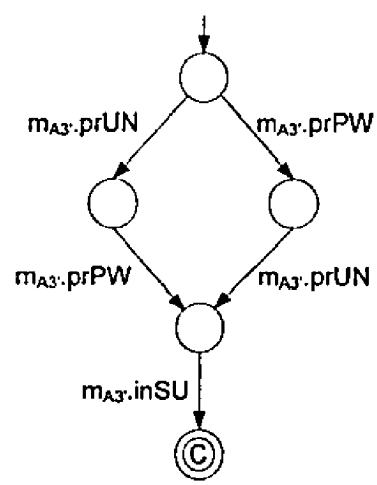
Table 9.5. Acceptance Sets of $M_{m_{A1}}$ and $M'_{m_{A2}}$

Acceptance Sets of	$M_{m_{A1}}$	$M'_{m_{A2}}$
$Acc(q_0, \langle\ \rangle) =$	$\{\{m_{A1}.prUP, m_{A1}.prFP\}\}$	$\{\{m_{A1}.prUP\}\}$
$Acc(q_0, \langle m_{A1}.prUP\rangle) =$	$\{\{m_{A1}.vaUP\}\}$	$\{\{m_{A1}.vaUP\}\}$
$Acc(q_0, \langle m_{A1}.prUP, m_{A1}.vaUP\rangle) =$	$\{\{m_{A1}.inSU\}, \{m_{A1}.inCI\}\}$	$\{\{m_{A1}.inSU\}, \{m_{A1}.inCI\}\}$
$Acc(q_0, \langle m_{A1}.prUP, m_{A1}.vaUP, m_{A1}.inSU\rangle) =$	$\{\emptyset\}$	$\{\emptyset\}$
$Acc(q_0, \langle m_{A1}.prUP, m_{A1}.vaUP, m_{A1}.inCI\rangle) =$	$\{\emptyset\}$	$\{\emptyset\}$

The respective acceptance sets are depicted in Table 9.5. In case of the empty trace (⟨ ⟩) we have $|\{\{m_{A1}.s_1, m_{A1}.s_{1a1}\}\}| = |\{\{m_{A1}.s_1\}\}|$. Since the acceptance sets for the remaining (partial) traces of $M_{m_{A2}}$ are identical for $M_{m_{A1}}$ and $M'_{m_{A2}}$ we conclude that $M_{m_{A1}} \geq_{dred} M'_{m_{A2}}$ and thus use case m_{A2} is a proper refinement of use case m_{A1}.

Case 2: Use Case Model – (Req.) Task Model Refinement: A task model at the requirements level is a valid refinement of a use case model, if it has the same or less traces *and* if all choices of the use case model, that happen nondeterministically from the user's point of view, are preserved in the task model. In order to illustrate the refinement, we prove (using the nFSM semantics) that the "Login" task model m_{A3} (as given in Chapter 3) is a valid refinement of the "Login" use case m_{A2}, whereas the "Login" task model m'_{A3} is not a valid refinement.

Table 9.6. nFSMs of "Login" Task Models m_3 and m'_{A3}

nFSM $M_{m_{A3}}$ Representing TM m_3	nFSM $M_{m'_{A3}}$ Representing TM m'_{A3}
	

The nFSM representations $M_{m_{A3}}$ and $M_{m'_{A3}}$ of m_{A3} and m'_{A3}, respectively, are given in Table 9.6. In order to be able to compare $M_{m_{A3}}$ and $M_{m'_{A3}}$ with $M_{m_{A2}}$ (Figure 9.2 (a)), we *first* replace all transitions of $M_{m_{A2}}$ that are triggered by events representing internal

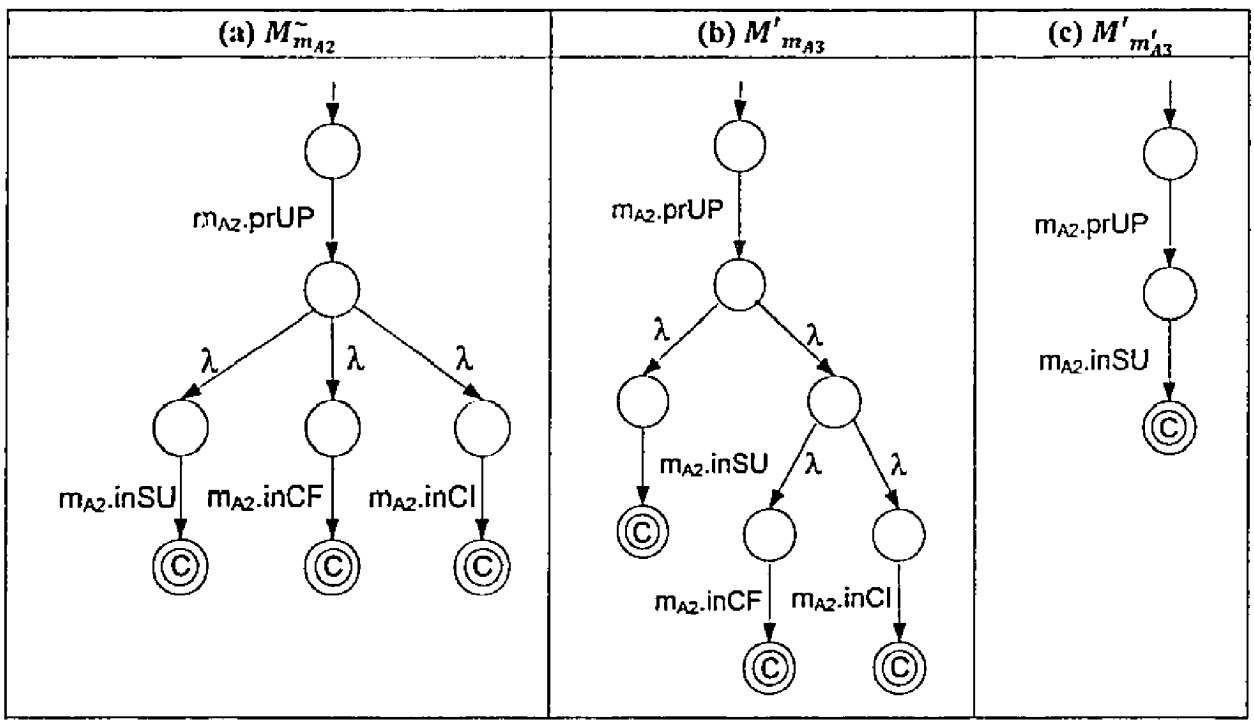
system steps (which are not captured in task models) with lambda transitions. The resulting nFSM $M_{m_{A2}}^{\sim}$ is depicted in Table 9.7 (a). *Second*, we apply the following refinement mappings to $M_{m_{A3}}$ and $M_{m'_{A3}}$:

$$rmap = \begin{Bmatrix} \{m_{A3}.prUN, m_{A3}.prPW\} \mapsto m_{A2}.prUP, \{m_{A3}.inSU\} \mapsto m_{A2}.inSU, \\ \{m_{A3}.inCF\} \mapsto m_{A2}.inCF, \{m_{A3}.inCI\} \mapsto m_{A2}.inCI \end{Bmatrix}$$

$$rmap' = \{\{m'_{A3}.prUN, m'_{A3}.prPW\} \mapsto m_{A2}.prUP, \{m'_{A3}.inSU\} \mapsto m_{A2}.inSU\}$$

There is a *many-to-one* refinement between use case step $m_{A2}.prUP$ and the refining tasks $m_{A3}.prUN$, $m_{A3}.prPW$ of m_{A3} and $m'_{A3}.prUN$, $m'_{A3}.prPW$ of m'_{A3}, respectively. Further, in the case of m_{A3} use case steps $m_{A2}.inSU$, $m_{A2}.inCF$ and $m_{A2}.inCI$ are *isomorphically* refined by the tasks $m_{A3}.inSU$, $m_{A3}.inCF$ and $m_{A3}.inCI$, whereas in case of m'_{A3} only the use case step $m_{A2}.inSU$ is *isomorphically* refined by $m'_{A3}.inSU$. In contrast to m_{A3}, use case steps $m_{A2}.inCF$ and $m_{A2}.inCI$ do not have a corresponding task in m'_{A3}.

Table 9.7. nFSMs after Hiding of Internal Events (a) and Refinement Mappings (b,c)



The resulting nFSMs $M'_{m_{A3}}$ and $M'_{m'_{A3}}$ after the refinement mapping are depicted in Table 9.7 (b, c). In order to validate that task model m_{A3} is a proper refinement of use

case m_{A1} we need to verify that $M^{\sim}_{m_{A2}} \geq_{dred} M'_{m_{A3}}$. Clearly, the first condition is fulfilled, since both nFSMs have the exact same set of traces:

$$Tr(M^{\sim}_{m_{A2}}) = Tr(M'_{m_{A3}})$$
$$= \{(m_{A2}.prUP, m_{A2}.inSU), (m_{A2}.prUP, m_{A2}.inCF), (m_{A2}.prUP, m_{A2}.inCI)\}$$

As for the liveness condition, Table 9.8 depicts that $M^{\sim}_{m_{A2}}$ and $M'_{m_{A3}}$ have the same acceptance sets. Hence we conclude that the "Login" task model m_{A3} is a valid refinement of the "Login" use case m_{A2}.

Table 9.8. Acceptance Sets of $M_{m_{A1}}$ and $M'_{m_{A2}}$

Acceptance Sets of	$M^{\sim}_{m_{A2}}$ **and** $M'_{m_{A3}}$	$M'_{m'_{A3}}$
$Acc(q_0, \langle\ \rangle) =$	$\{\{m_{A2}.prUP\}\}$	$\{\{m_{A2}.prUP\}\}$
$Acc(q_0, \langle m_{A2}.prUP\rangle) =$	$\{\{m_{A2}.inSU\}, \{m_{A2}.inCI\}, \{m_{A2}.inCF\}\}$	$\{\{m_{A2}.inSU\}\}$
$Acc(q_0, \langle m_{A2}.prUP, m_{A2}.inSU\rangle) =$	$\{\emptyset\}$	$\{\emptyset\}$
$Acc(q_0, \langle m_{A2}.prUP, m_{A2}.inCI\rangle) =$	$\{\emptyset\}$	N/A
$Acc(q_0, \langle m_{A2}.prUP, m_{A2}.inCF\rangle) =$	$\{\emptyset\}$	N/A

We continue validating whether m'_{A3} is a valid refinement of m_{A2}. Clearly, with $Tr\left(M'_{m'_{A3}}\right) = \{(m_{A2}.prUP, m_{A2}.inSU)\}$ the set of traces of $M'_{m'_{A3}}$ is a sub-set of the set of traces of $M^{\sim}_{m_{A2}}$ which satisfies the safety property. Table 9.8, however, depicts that the sizes of the acceptance sets of $M'_{m'_{A3}}$ and $M^{\sim}_{m_{A2}}$ are not always identical. In particular, after partial trace $\langle m_{A2}.prUP\rangle$ the corresponding acceptance set of $M^{\sim}_{m_{A2}}$ consists of multiple events, whereas in case of $M'_{m'_{A3}}$ it is a singleton set. Hence $M'_{m'_{A3}}$ is *not* a deterministic reduction of $M^{\sim}_{m_{A2}}$ and we conclude that task model m'_{A3} is not a valid refinement of m_{A2}. The result corresponds to our informally stated requirement that system choices need to be preserved in all refining task models. This, however, is not the case for the m'_{A3} "Login" task model, which restricts the nondeterministic system response to only success messages. Failure cases, as specified in the use case, are not taken into account.

Case 3: (Req.) Task Model – (Req.) Task Model Refinement: In a valid refinement between task models at the requirements level, every scenario of the refining model must

also be a valid scenario on the base model. Further, only user choices may be restricted, while limiting a system choice to provide fewer alternatives is not allowed. In what follows, we illustrate this refinement principle by showing that the "Login" task model m_{A4} is a valid refinement of its relative base-task model m_{A3}. Intuitively, m_{A4} implements a sub-set of the scenarios offered by m_{A3}. The previously arbitrary order of providing the username and the password has been restricted to a sequential order. The system choice between displaying a success message and displaying a failure message has been preserved.

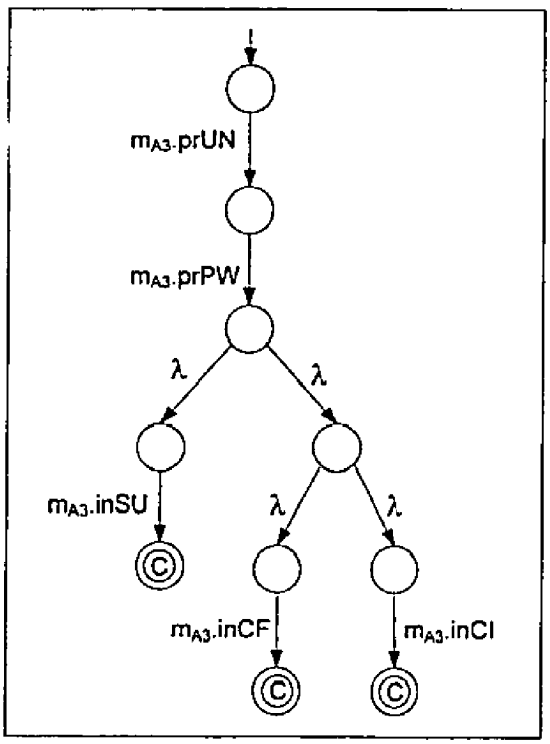


Figure 9.3. nFSM of "Login" Task Model (m_{A4}) after Refinement Mapping

Formally, the validity of the refinement is proven by showing that the nFSM formalization of m_{A4} (after the refinement mapping) is a deterministic reduction of the nFSM formalization of m_{A3} ($M_{m_{A3}}$ in Table 9.6). The former $(M'_{m_{A4}})$ is depicted in Figure 9.3. For the sake of conciseness, the nFSM before the refinement mapping and the obvious refinement mapping (which consists only of isomorphic refinements) are omitted. Clearly, we have trace inclusion with $Tr(M'_{m_{A4}}) \subseteq Tr(M_{m_{A3}})$. Further, the respective acceptance sets for each partial trace of $M'_{m_{A4}}$ are identical, with the exception of $Acc(q_0, \langle\ \rangle)$, which, in case of $M_{m_{A3}}$, is $\{\{m_{A3}.prUN, m_{A3}.prPW\}\}$ and in case of

$M'_{m_{A4}}$ is $\{\{m_{A3}.prUN\}\}$. However, since the two acceptance sets have the same size, we conclude that $M'_{m_{A4}}$ is a deterministic reduction of $M_{m_{A3}}$ and hence m_{A4} is a valid refinement of m_{A3}.

Case 4: (Req.) Task Model – (Design) Task Model Refinement: When moving from a requirements to a design model, we need to ensure that the design model implements the *same* set of scenarios as its requirements-level counterpart. In contrast to the previous refinement cases, a restriction of the set of scenarios is not allowed. In particular, we require that the respective set of posets and nFSM formalizations are trace equivalent. In what follows, we illustrate (for both semantic domains) that the design-level "Login" task model m_{A5} is a valid refinement of its relative base model m_{A4}.

For this purpose, we apply the following refinement mapping:

$$rmap = \left\{ \begin{array}{c} \{m_{A5}.slUN, m_{A5}.tpUN, m_{A5}.smUN\} \mapsto m_{A4}.prUN, \\ \{m_{A5}.slPW, m_{A5}.tpPW, m_{A5}.smPW\} \mapsto m_{A4}.prPW, \\ \{m_{A5}.diSU\} \mapsto m_{A4}.inSU, \{m_{A5}.diCF\} \mapsto m_{A4}.inCF, \{m_{A5}.diCI\} \mapsto m_{A4}.inCI \end{array} \right\}$$

We note the *many-to-one* refinement between the "Provide Username" task ($m_{A4}.prUN$) of m_{A4} and the respective sub-tasks in m_{A5}, namely "Select UN Field" ($m_{A5}.slUN$), "Type Username" ($m_{A5}.tpUN$) and "Click Submit UN" ($m_{A5}.smUN$). Further, there is a *many-to-one* refinement between "Provide Password" in m_{A4} and "Select PW Field" ($m_{A5}.slPW$), "Type Password" ($m_{A5}.tpPW$) and "Click Submit PW" ($m_{A5}.smPW$) in m_{A5}. The remaining tasks in m_{A4} are *isomorpically* refined in m_{A5}.

Figure 9.4 shows the nFSM representation $M'_{m_{A5}}$ (after application of the refinement mapping) of the design-level task model m_{A5}. It is clearly identical to the requirements-level task model $M_{m_{A4}}$ and we conclude that $M'_{m_{A5}}$ and $M_{m_{A4}}$ are trace equivalent and thus m_{A5} is a valid refinement of m_{A4}.

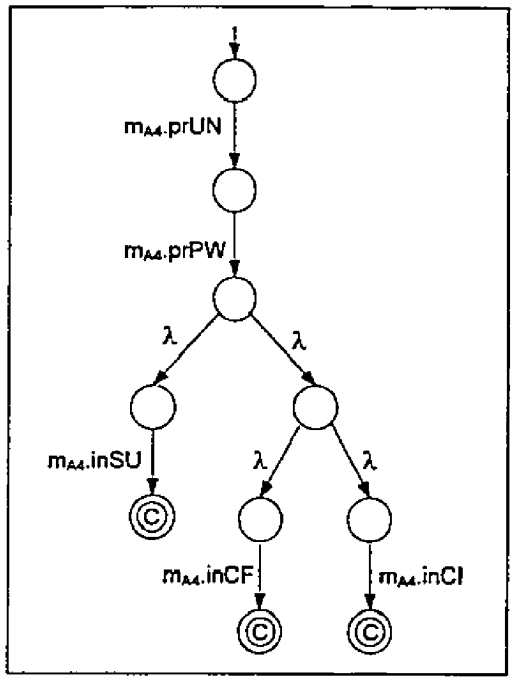


Figure 9.4. nFSM of "Login" Task Model (m_{A5}) after Refinement Mapping

We continue by showing that m_{A5} is also a valid refinement of m_{A4} in the set of posets semantics. The corresponding formalization $P'_{m_{A5}}$ of m_{A5} (after application of the before-mentioned refinement mapping) is as follows (for the sake of conciseness, we use the short-hand notation introduced in Chapter 6):

$$P'_{m_{A5}} = [m_{A4}.prUN, m_{A4}.prPW, m_{A4}.inSU] \# [m_{A4}.prUN, m_{A4}.prPW, m_{A4}.inCF] \# [m_{A4}.prUN, m_{A4}.prPW, m_{A4}.inCI]$$

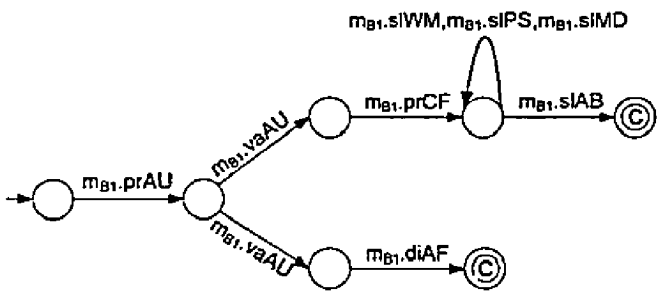
The set of traces of $P'_{m_{A5}}$ (depicted below) is clearly identical to the trace set of $M_{m_{A4}}$ and $M'_{m_{A5}}$. Hence, also in the set of posets semantics, m_{A5} is a valid refinement of m_{A4}.

$$Tr(P'_{m_{A5}}) = \left\{ \begin{matrix} \langle m_{A4}.prUN, m_{A4}.prPW, m_{A4}.inSU \rangle, \langle m_{A4}.prUN, m_{A4}.prPW, m_{A4}.inCF \rangle, \\ \langle m_{A4}.prUN, m_{A4}.prPW, m_{A4}.inCI \rangle \end{matrix} \right\}$$

Case 5: Use Case Model – (Design) Task Model Refinement: We have to ensure that the task model adopts the entirety of the functional requirements specified in the use case. An omission of user alternatives, as allowed in refinement case 2, is not possible. That is, we require that the task model allow the same set of scenarios as the use case model and vice versa. Table 9.9 shows the nFSM and set of posets formalization of the "Use ATM" use case (m_{B1}) of Chapter 3. For the sake of this example, the inclusions of sub-use cases

are not resolved and the respective inclusion steps are treated as atomic steps of type *interaction*.

Table 9.9. Semantic Representations of "Use ATM" Use Case (m_{B1})

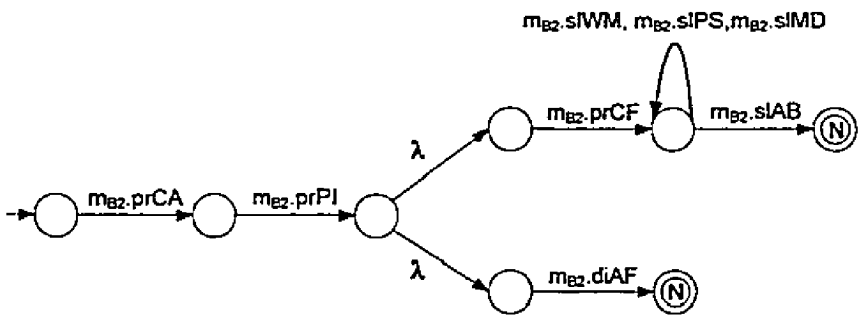
nFSM Formalization $(M_{m_{B1}})$ of "Use ATM" Use Case (m_{B1})

Sets of Posets Formalization $(P_{m_{B1}})$ of "Use ATM" Use Case (m_{B1})
$(m_{B1}.prAU, m_{B1}.prCF, m_{B1}.vaAU \cdot$ $([m_{B1}.slWM]\#\,[m_{B1}.slPS]\,\#\,[m_{B1}.slMD])^* \cdot [m_{B1}.slAB])\,\#$ $[m_{B1}.prAU, m_{B1}.vaAU, m_{B1}.diAF]$

The semantic representations of the design-level task model (m_{B2}) are given in Table 9.10. Similar to m_{B1}, sub-ECTT task definitions are not resolved and the respective tasks are treated as atomic tasks of type *interaction*. In order to prove that m_{B2} is a valid refinement of m_{B1}, we show that the respective semantic representations are trace equivalent. We apply the following refinement mapping:

$$rmap = \left\{ \begin{array}{c} \{m_{B2}.prCA, m_{B2}.prPI\} \mapsto m_{B1}.prAU, \{m_{B2}.prCF\} \mapsto m_{B1}.prCF, \\ \{m_{B2}.slWM\} \mapsto m_{B1}.slWM, \{m_{B2}.slPS\} \mapsto m_{B1}.slPS, \\ \{m_{B2}.slMD\} \mapsto m_{B1}.slMD, \{m_{B2}.slAB\} \mapsto m_{B1}.slAB, \{m_{B2}.diAF\} \mapsto m_{B1}.diAF \end{array} \right\}$$

Use case step "Provide Authentication" ($m_{B1}.prAU$) is *many-to-one* refined by the sub-tasks "Provide Card" ($m_{B2}.prCA$) and "Provide PIN" ($m_{B2}.prPI$). The remaining use case steps (except for the *internal* "Authentication" step ($m_{B1}.vaAU$)) are *isomorpically* refined by the respective sub-tasks in m_{B2}. Since the resulting nFSMs and sets of posets have the same traces than the respective semantic representations of m_{B1} (after all internal events have been hidden), we conclude that task model m_{B2} is a valid design-level refinement of use case m_{B1}.

Table 9.10. Semantic Representations of "Use ATM" Task Model (m_{B2})

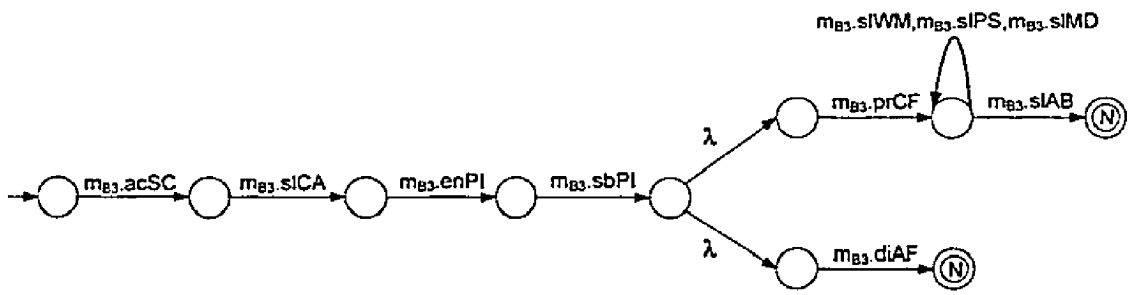
nFSM Formalization $(M_{m_{B2}})$ of "Use ATM" Task Model (m_{B2})

Sets of Posets Formalization $(P_{m_{B2}})$ of "Use ATM" Task Model (m_{B2})
$([m_{B2}.prCA, m_{B2}.prPI] \cdot ([m_{B2}.slWM]\#\,[m_{B2}.slPS]\#[m_{B2}.slMD])^* \cdot$ $[m_{B2}.slAB, STOP])\,\#$ $[m_{B2}.prCA, m_{B2}.prPI, m_{B1}.diAF, STOP]$

Case 6: (Design) Task Model – (Design) Task Model Refinement: At the design level, a task model is a valid refinement of a given base-task model, if it allows the same set of scenarios as its base model. Formally, we verify that the semantic representation of the refining model is, after the application of the refinement mapping, *trace equivalent* to the respective semantic representation of the base-task model. Table 9.11 shows the nFSM and set of posets formalizations of the design-level task model (m_{B3}) of Chapter 3. In order to prove that m_{B3} is a valid refinement of m_{B2} we use the following (obvious) refinement mapping:

$$rmap = \left\{ \begin{array}{c} \{m_{B3}.acSC, m_{B3}.slCA\} \mapsto m_{B2}.prCA, \{m_{B3}.enPI, m_{B3}.sbPI\} \mapsto m_{B2}.prPI, \\ \{m_{B3}.prCF\} \mapsto m_{B2}.prCF, \{m_{B3}.slWM\} \mapsto m_{B2}.slWM, \{m_{B3}.slPS\} \mapsto m_{B2}.slPS, \\ \{m_{B3}.slMD\} \mapsto m_{B2}.slMD, \{m_{B3}.slAB\} \mapsto m_{B2}.slAB, \{m_{B3}.diAF\} \mapsto m_{B2}.diAF \end{array} \right\}$$

The task "Provide Card" ($m_{B2}.prCA$) is *many-to-one* refined by sub-tasks "Activate Screen" ($m_{B3}.acSC$) and "Slide ATM Card" ($m_{B3}.slCA$). Similarly, the task "Provide PIN" ($m_{B2}.prPI$) is *many-to-one* refined by "Enter PIN" ($m_{B3}.enPI$) and "Submit" ($m_{B3}.sbPI$). The remaining tasks of m_{B2} are isomorphically refined by the respective sub-tasks in m_{B3}. If we apply the refinement mapping to $M_{m_{B3}}$ and $P_{m_{B3}}$, we obtain an nFSM and a set of posets identical to $M_{m_{B2}}$ and $P_{m_{B2}}$, from which we conclude that the design-level task model m_{B3} is a valid refinement of design-level task model m_{B2}.

Table 9.11. Semantic Representations of "Use ATM" Task Model (m_{B3})

nFSM Formalization $(M_{m_{B3}})$ of "Use ATM" Task Model (m_{B3})

Sets of Posets Formalization $(P_{m_{B3}})$ of "Use ATM" Task Model (m_{B3})
$([m_{B3}.acSC, m_{B3}.slCA, m_{B3}.enPI, m_{B3}.sbPI] \cdot$ $([m_{B2}.slWM]\#\,[m_{B2}.slPS]\#[m_{B2}.slMD])^* \cdot [m_{B2}.slAB, STOP])\,\#$ $[m_{B3}.acSC, m_{B3}.slCA, m_{B3}.enPI, m_{B3}.sbPI, m_{B3}.diAF, STOP]$

The refinement examples, discussed in this section, are sufficiently simple to "manually" prove refinement by pairwise comparing the relevant trace and acceptance sets. However, as the specifications become more complex, refinement verification requires supporting tools. Therefore in the next section, we introduce the *Use Case - Task Model – Verifier*, which, given two specifications from the intermediate semantic domain, performs the refinement check automatically.

9.3 Tool Support

Based on the nFSM semantics of Chapter 7, we have developed the *Use case - Task Model Verifier*. It assists the developer in formally validating and comparing use case and/or task models.

9.3.1 Input Specification

The input language for the *Use Case - Task Model – Verifier* is the "Scenario Specification and Refinement Language." (SSRL). It consists of five parts: *Event Declaration*, *UC-LTS Specification*, *GTM Specification*, *Refinement Mapping*, and *Refinement Assertion*. In what follows, we provide an overview of each part. The complete grammar in EBNF is given in Appendix F.

Event Declaration: Before an event can be used within a UC-LTS or GTM specification it must be globally declared. Each event declaration consists of a representative (distinct) name followed by the event type. Figure 9.5 shows the event declarations for the "Order Product" UC-LTS (Figure 5.5) and GTM (Table 5.4).

```
EVENTS:
  //Events in "Order Product" Use Case Model
  trOP: INTERACTION; spCA: INTERACTION; diSR: APPLICATION;
  inPS: INTERACTION; inCA: INTERACTION; slPQ: INTERACTION;
  vaPQ: INTERNAL; diPS: APPLICATION; paDB: INTERACTION;
  paCC: INTERACTION; prPI: INTERACTION;  vaPA: INTERNAL;
  inCO: APPLICATION; inPF: APPLICATION; inIQ: APPLICATION;

  //Events in "Order Product" Task Model
  chOP: INTERACTION; slCR: INTERACTION; sbCR: INTERACTION;
  dpRS: APPLICATION; inQO: INTERACTION; inRS: INTERACTION;
  slPD: INTERACTION; slQT: INTERACTION; sbPS: INTERACTION;
  dpPS: APPLICATION; dpOS: APPLICATION; swCC: INTERACTION;
  dpPF: APPLICATION; dpCF: APPLICATION;
```

Figure 9.5. Declaration Section for "Order Product" Use Case and Task Model

UC-LTS Specification: This part defines a list of UC-LTS specifications. Each such UC-LTS specification consists of a (distinct) name, followed by a (non-empty) list of transitions, and a list of final states. Each transition is specified as a triple consisting of a source state, a list of triggering events, and a target state. It is assumed that the source state of the first transition in the transition list is the initial state of the UC-LTS. Figure 9.6 shows the SSRL formalization of the "Order Product" UC-LTS.

```
USECASEDEFS:
  //UC-LTS of the "Order Product" Use Case Model
  OrderProductUC = (
  //Main Success Scenario:
     {
       (q0, {trOP}, q1), (q1, {spCA}, q2), (q2, {diSR}, q3),
       (q3, {slPQ}, q4), (q4, {vaPQ}, q5), (q5, {diPS}, q6),
       (q6, {paDB}, q7), (q6, {paCC}, q8), (q7, {prPI}, q9),
       (q8, {vaPA}, q13),(q9, {vaPA}, q13),(q8, {vaPA}, q10),
       (q9, {vaPA}, q10),(q10, {inCO}, q11),
  //Extensions:
       // 4a. Primary actor is not satisfied with the search results:
       (q3, {inPS}, q1),
       // 4b. Primary actor decides to cancel the use case:
       (q3, {inCA}, q17),
       // 5a. The desired product is unavailable:
       (q4, {vaPQ}, q12), (q12, {inIQ}, q19),
       // 7a. Primary actor decides to cancel the use case:
       (q6, {inCA}, q23),
       // 8a. The payment was not authorized:
       (q13, {inPF}, q6)
     },
  //Set of FinalStates of the UC-LTS
    {q17, q19, q23, q11});
```

Figure 9.6. UC-LTS Specification of "Order Product" Use Case

GTM Specification: Provides a list of GTM specifications, where each specification consists of a (distinct) name followed by a generic task expression. In accordance with

Definition 5.15 a generic task expression is formed by relating events using the following temporal operators >>, ||, [], *, stop, resume. The binary operators are listed in the order of their precedence, with >> having the lowest precedence. As usual, parentheses can be used to overrule operator priorities. As an example, the GTM specification for the "Order Product" task model is given in Figure 9.7.

```
TASKDEFS:
  //Order Product Generic Task Model
  OrderProductTM =  chOP >> (slCR >> sbCR >> dpRS >>
                    (STOP (inQO) [] inRS))* >> slCR >> sbCR >> dpRS >>
                    slPD >> slQT >> sbPS >> (dpPS [] STOP (dpOS)) >>
                    (STOP (inQO) [] (swCC >> dpPF))* >> swCC >> dpCF;
```

Figure 9.7. SSRL GTM Specification of the "Order Product" Task Model

Refinement Mapping and Assertions: In order to compare two specifications, a refinement mapping is needed. In SSRL, each refinement mapping consists of a (distinct) name followed by a list of pairs, which relate events of the refining specification to events of the based specification. The refinement mapping is needed in the assertion section which defines a list of verification problems. Each verification problem consists of four parts: (1) name of the base specification, (2) refinement type (trace equivalence (EQU) or deterministic reduction (DET)), (3) name of the refining specification, and (4) name of the refinement mapping. A refinement problem, involving the "Order Product" use case and task model, together with the corresponding refinement mapping is given in Figure 9.8. Recall that the "Order Product" task model is a requirements specification; hence *deterministic reduction* was chosen as the appropriate refinement type.

```
REFMAPS:
  //Refinement Mapping from OrderProductTM to OrderProductUCM
  OrderProductUCtoTM = {({chOP},trOP), ({slCR,sbCR},spCA),
                       ({dpRS},diSR), ({inQO},inCA), ({inRS},inPS),
                       ({slPD, slQT, sbPS}, slPQ), ({dpPS},inIQ),
                       ({dpOS},diPS), ({swCC},paCC), ({dpPF},inPF),
                       ({dpCF},inCO)};

ASSERT:
  /*Assert that the Order Product Task Model is a deterministic
    Reduction of the Order Product Use Case Model. */
  OrderProductUCM DET OrderProductTM WITH OrderProductUCtoTM;
```

Figure 9.8. Refinement Mapping and Assertion for "Order Product" UC and TM

9.3.2 Translation and Verification Phases

A path to the file containing an SSRL specification is passed as a command line parameter to the *Use Case - Task Model – Verifier*. The tool, then, performs the following actions:

1. **Parsing:** The specification is parsed for syntax errors based on the grammar specification given in Appendix F. If the input is syntax-error free an abstract syntax tree (AST) is created.
2. **Static Analysis:** The AST is statically analyzed to ensure that important well-formedness conditions are satisfied such as:
   - Uniqueness of event and specification names
   - Events used in UC-LTS and GTM specification have been declared
   - List of final states of UC-LTS specification are included as source or target states in the list of all transitions
   - etc.
3. **Semantic Mapping to nFSMs:** The sub-trees representing specifications used in any of the assertions are traversed, and the respective nFSMs are generated (using the semantic mappings given in Chapter 7).
4. **Refinement Mapping and Event Abstraction:** The respective *refinement mappings* are applied to all refining specifications. Additionally, in case of use case – task model refinement, *event abstraction* is applied to the base specification.
5. **Refinement Verification:** For each assertion, the tool verifies its validity and presents the verification result. In case of a failure, a counter example is produced.

Figure 9.9 presents the verification result for the assertion given in Figure 9.8. As expected, the assertion is proven correct as the nFSM (Figure 7.6) representing the "Order Product" task model implements a subset of the traces of the nFSM (Figure 7.5) representing the "Order Product" use case. Furthermore, all nondeterministic choices have been preserved.

```
Verification Result:

OrderProductUCM DET OrderProductTM WITH OrderProductUCtoTM: Valid
Refinement: (76 state pairs explored)
```

Figure 9.9. Positive Verification Result

We conclude this section by presenting a refinement problem which fails the verification. For this purpose, let us assume that the "Order Product" task model is a design-level task model and hence *trace equivalence* is the appropriate refinement type. In this case, the tool reports a verification failure together with the shortest[11] partial trace of the base specification, after which the verification failed.

```
Verification Result:

OrderProductUCM EQU OrderProductTM WITH OrderProductUCtoTM: Refinement
Failure after partial trace <trOP, spCA, diSR, slPQ, LAMBDA, diPS> event
paDB expected (23 state pairs explored)
```

Figure 9.10. Negative Verification Result

In the next section, we provide algorithmic details for both implemented refinement checks, viz. *trace equivalence* and *deterministic reduction.*

9.3.3 Algorithms

Our tool implements a model-checking algorithm similar to the one used in FDR [Roscoe 1994]. Instead of checking two LTSs for *failure-divergence refinement*, we adapted the algorithm to compare two nFSMs for refinement in terms of *trace inclusion* and *deterministic reduction.* The verification of refinement is performed in two steps. (1) **Normalization**: The nFSMs are transformed into acceptance graphs. (2) **Model Checking**: The refinement is proven by pairwise comparing the states in the base and refining model, which have a trace in common.

9.3.3.1 Normalization

The nFSMs produced by the semantic mappings potentially contain a high degree of nondeterminism. Any method for deciding refinement between a given base nFSM M_1 and a refining nFSM M_2 will have to keep track of all states in M_1 after a given partial trace w and the different paths possible for each of these states. Refinement verification would be more straightforward, if there were exactly one state corresponding to each

[11] This is due to the breadth-first nature of the model-checking algorithm, which will be detailed in the next section.

possible trace w. This can be achieved by transforming the nFSMs into acceptance graphs before the proof of refinement is conducted.

For this purpose, we implemented the conversion function ag given in Definition D.10 (Appendix D). Similar to the algorithm defined by Roscoe [1994], we start with the creation of the initial node, which corresponds to the *lambda closure* of the initial state of the nFSM. Next, the following step is performed repeatedly until all new nodes generated are ones that have been previously expanded:

For each created node q and for each event $e \in Initials(q)$ **do**:

- Create a new node q' representing the set of nodes reachable from q after action e and any number of lambda transitions.
- Create a transition from q to q' triggered by e.
- Calculate the acceptance set for q as specified in Definition D.10.

Since the algorithm builds a transition system over the powerspace of the original nFSM, there is the possibility that the acceptance graph will be exponential in the size of the nFSM. A pathological example of such a case can be found in [Roscoe 1994]. Fortunately, according to our experience and the experiences of others [Roscoe 1994], in practice it is rare that the corresponding acceptance graph has more states than the original nFSM. On the contrary, the acceptance graph is often significantly smaller than the original specification.

9.3.3.2 Model Checking

Let M_1 and M_2 be two nFSMs and $AG_1 = (Q_1, \Sigma_1, \chi_1, \delta_1, q_{1_0}, F_{C1}, F_{N1})$ and $AG_2 = (Q_2, \Sigma_2, \chi_2, \delta_2, q_{2_0}, F_{C2}, F_{N2})$ be the respective acceptance graphs. In order to verify that M_2 is a valid refinement of M_1 we have to discover whether the behavior of each state q_2 in Q_2 is allowable with respect to a corresponding state q_1 in Q_1 such that q_2 and q_1 have a trace in common. This is done by exploring the Cartesian product of the states of AG_1 and AG_2.

The pseudo code of the implemented model-checking algorithm is given in Figure 9.11. Its main idea can be summarized as follows: We maintain a set of checked state pairs

(checkedPairs) and a set of pending state pairs (pendingPairs). Initially, the former is empty and the latter is the singleton set containing the pair of initial states of AG_1 and AG_2. Until it is empty, we repeatedly inspect pairs from pendingPairs for refinement. For each pair (q_1, q_2), if the refinement holds (will be detailed next), (q_1, q_2) is added to (checkedPairs) while all pairs reachable from (q_1, q_2), which are not already in checkedPairs, are added to pendingPairs. The procedure terminates when pendingPairs becomes the empty set or if the refinement check fails for a given state pair.

```
proc isRefinement(AccGraph refinedAG, AccGraph refiningAG, RefType refType)
returns Boolean {
    set StatePair pendingPairs = (getRoot(ag1), getRoot(ag2))
     set StatePair checkedPairs = ∅
     while pendingPairs ≠ ∅ do
       StatePair currentPair = extractNextPair (pendingPairs)
       Result result := validateStateProperties(currentPair.first,
                                    currentPair.second, refType)
       if result = NOERROR then
         checkedPairs := checkedPairs ∪ currentPair
         foreach nextPair in getReachablePairs(currentPair) do
           if nextPair ∉ checkedPairs then
             pendingPairs := pendingPairs ∪ nextPair
           fi
         od
       else
         return FALSE
       fi
     od
     return TRUE
   }
```

Figure 9.11. State Traversal Algorithm

The refinement check for an individual state pair is given in Figure 9.12. A case distinction is made depending on the refinement type. In the case of *trace equivalence*, a given pair (q_1, q_2) passes the refinement check, if the set of initial events for q_1 and q_2 are the same and if q_1 and q_2 are either both final states or both non-final states. If any of these two conditions fails to check, it is concluded that the trace sets of AG_1 and AG_2 are not identical and hence M_1 and M_2 are not trace equivalent. In the case of *deterministic reduction*, it is verified that the initial events of q_2 form a subset of the initial events of q_1. If q_1 is a final state, then q_2 must be a final state as well. Moreover we verify that the respective acceptance sets of q_1 and q_2 have the same cardinality. If the first two conditions fail to check, it is concluded that the trace set M_2 cannot be a subset of the trace set of M_1. If the third condition fails to check, it is concluded that M_2 has "illegally" restricted a nondeterministic choice of M_1 to provide fewer alternatives.

```
proc validateStateProperties (AccState refinedAccState, AccState
refiningAccState, RefType refType) returns Result {
  switch (refType) {
      case EQUIVALENCE:
        if refiningAccState.initials ≠ refinedAccState.initials then
           return ERROR (PTraceError)
        fi
        if refiningAccState.isFinal ≠ refinedAccState.isFinal then
           return ERROR (PTraceError)
        fi
        return NOERROR

      case DETREDUCTION:
        if refiningAccState.initials ⊄ refinedAccState.initials then
           return ERROR (PTraceError)
        fi
        if !(refiningAccState.isFinal → refinedAccState.isFinal) then
           return ERROR (TraceError)
        fi
        if refiningAccState.AccSet.size ≠ refinedAccState.AccSet.size then
           return ERROR (LivenessError)
        fi
        return NOERROR
    } //end of switch
}
```

Figure 9.12. Refinement Checking Algorithm

10 Conclusion

In this thesis we propose an integrated development methodology for use case and task models. While use cases are the predominant specification medium for functional requirements, task models are employed to capture UI requirements and designs. In current practice, both entities are treated rather independently and are often developed by different teams, which have their own models and lifecycles. This is especially problematic since the lack of integration often results in inconsistent functional and UI design specifications, causing duplication of effort while increasing the maintenance overhead.

To address this shortcoming, we have developed an integrated methodology that can serve as a blueprint for practitioners to derive an iterative and incremental development process, according to which the two artifacts can be refined in a stepwise and integrated manner. For each step, one can verify that the resulting model is a valid refinement of its base model. For this purpose, we defined six refinement relations for use case and/or task models and provided automated tool support.

The integrated development methodology rests upon a formal framework, which defines a two-step mapping from a particular use case or task model notation to a common semantic domain. Such a two-step mapping results in a semantic framework that can be more easily validated, reused and extended. The intermediate semantic domains have been carefully chosen by taking into consideration the intrinsic characteristics of task models and use cases. We selected sets of posets and nFSMs as semantic domains, supporting a true concurrent and interleaving model of concurrency, respectively.

To this end, the contributions of this thesis are as follows:

1. *Enhancement of Use Case and Task Models:* We defined *DSRG-style use case models* and *ECTT task models* as improvements to their respective state-of-the-art counterparts, *Cockburn-style use case models* and *CTT*. Each improvement has been carefully selected to ensure that the intent and nature of each model is preserved.
   - In case of DSRG-style use case models, we introduced *step kinds* and *step types* as distinguishing factors for use case steps. Two different types of inclusions of

sub-use cases have been defined, providing the developer with the flexibility to specify actions depending on the successful or unsuccessful termination of the sub-use case.

- In case of ECTT, as our main contribution, we defined two novel temporal operators: *Stop* and *Resume*, that allow the developer to model error and failure cases, and provide a mechanism to *catch* errors and prevent their propagation. Also, in order to overcome the predominant, yet obsolete, monolithic task-tree structure, we defined ECTT in a modular fashion. Task models are developed in a true top-down manner while taking advantage of encapsulation.

2. *Formalization of Use Case and Task Models:* In current practice, use case and task models are used informally. The absence of a formal semantics hinders the effective verification of well-formedness properties and leaves little room for tool support. As a consequence, ambiguities and inconsistencies may go undetected, and are likely to propagate in subsequent development stages, resulting in higher costs to repair them. To address this shortcoming, we defined a formal semantics for use case and task models sharing two common semantic domains: *Sets of posets* and *nFSMs*. Each semantic domain has its own advantages and shortcomings with respect to tool support, expressiveness and compactness. On the one hand, sets of posets support a true model of concurrency, while nFSMs support an interleaving model, which does not allow for arbitrary refinement of events. On the other hand, sets of posets do not have intrinsic support for modeling nondeterminism. This shortcoming does not occur in the nFSM semantics, which allows for the formalization of all verification problems proposed by this thesis and is suitable for comprehensive tool support.

3. *Refinement between Use Cases and Task Models:* According to the proposed development methodology, use case and task models are incrementally perfected and refined. We demonstrated that depending on the nature of the involved artifacts and their role within the software-development lifecycle, a specific notion of refinement is required. For this purpose, we defined a suite of six refinement. Each refinement

relation has been formalized in the nFSM semantics and, if applicable, in the set of posets semantics.

4. *Validation of the Integrated Development Methodology and Formal Framework:* In order to validate the applicability of the proposed methodology, a small case study was carried out. We formalized and validated the requirements specification (in terms of use case and task models) of an "Invoice Management System."

   By expressing parts of our formal system in Isabelle/HOL, we formally proved key properties such as well-formedness, type consistency, and closure. In order to show that the set of posets semantics and the nFSM semantics coincide, we established a formal correspondence between them and proved that they are trace equivalent.

5. *Prototypical Tool Support for Developers.* The *Use Case and Task Model Verifier* provides developers with tool support to write and validate use case and task model specifications written in the intermediate semantic domains. Using a well-established model-checking algorithm, the tool automates the verification of refinement between two given use case and/or task models. In case of a refinement failure, the shortest possible counter example is generated.

Our research resulted in a methodology that allows UI design to follow as a logical progression from a functional requirements specification. At a minimum, the methodology will provide guidance to developers for the integrated specification of functional and user interface requirements, i.e., the purposes and roles of use case and task models in software development are clearly specified as well as the order in which both kinds of artifacts are to be developed. Ideally, the integrated development methodology could be used as part of a rigorous software development process in which use case and task models are stepwise refined. The transitivity of the refinement relations ensures validity of the resulting final artifacts with respect to a given source specification. The verification of refinement is automated by tools such as the *Use Case – Task Model – Verifier.*

In this thesis the applicability of the proposed development methodology has been partly validated by carrying out a small case study. In the future, we plan to conduct more comprehensive empirical studies. The results obtained will enable us to statistically contrast our development methodology with conventional heterogeneous development approaches. We will also be able to quantify the learning curve and the training effort required to effectively make use of the proposed development and refinement principles.

Another future avenue deals with the extension of the intermediate semantic domains. Currently, they are chosen in a way such that only *regular* sets of scenarios are possible. It is worth pursuing an extension of the intermediate semantic domains in support of e.g., an arbitrary form of recursion in task models, circular invocations of use cases, or the unrestricted use of *concurrent* use case steps and tasks. In all cases, the resulting trace set will no longer be *regular*, and only the set of posets semantics will be able to provide coverage of all syntactic constructs.

Furthermore, we are aiming to extent the proposed semantics to capture state information using Kripke structures. State information is often employed in use case or task models to express and evaluate conditions. For example, the precondition of a use case denotes the set of states in which the use case is to be executed. In addition, every use case extension is triggered by a condition that must hold before the steps defined in the extension are executed. In order to be able to evaluate conditions, the semantic model must provide a means to capture the system state and should be able to map state conditions to the appearance of events.

We are also currently extending the *Use Case and Task Model Verifier* to include additional semantic checks such as livelock detection in use case models. A *livelock* is a phenomenon, where an application performs an infinite sequence of internal actions. From the user's point of view, livelocks are highly undesirable as the user may be able to observe the presence of internal activity and hence hope "eternally" that some output will emerge eventually. We believe that some livelocks are (partly) induced due to erroneous use case specifications, which contain loops of *internal* steps. Clearly, as a requirements specification, the use case model serves as direct input to the subsequent development phases, and it is thus likely that the livelock will propagate to the implementation stage.

References

Abadi, M. and L. Lamport, The Existence of Refinement Mappings, in Theoretical Computer Science, **82** (2), pp. 253-284, 1991.

Alur, R., G. J. Holzmann and D. Peled, An Analyzer for Message Sequence Charts, in Software - Concepts and Tools, **17** (2), pp. 70-77, 1996.

Annett, J. and K. D. Duncan, Task Analysis and Training Design, in Occupational Psychology, **41**, pp. 211-221, 1967.

Armour, F. and G. Miller, Advanced Use Case Modeling, Addison-Wesley, 2001.

Back, R. J., A Calculus of Refinements for Program Derivations., in Acta Informatica, **25** (6), pp. 593-624, 1988.

Baeten, J. C. M. and W. P. Weijland, Process algebra, Cambridge University Press, New York, NY, USA, 1990.

Barnett, M., W. Grieskamp, W. Schulte, N. Tillmann and M. Veanes, Validating use-cases with the AsmL test tool, in Proceedings of *Quality Software 2003*, pp. 238-246, 2003.

Bastide, R. and S. Basnyat, Error Patterns: Systematic Investigation of Deviations in Task Models, in Proceedings of *TaMoDia*, Hasselt, Belgium, pp. 109-122, 2006.

Bergstra, J. A., Handbook of Process Algebra, Elsevier Science Inc, 2001.

Blyth, T. S., Set theory and abstract algebra, Longman, London, 1975.

Bolognesi, T. and E. Brinksma, Introduction to the ISO specification language LOTOS, in Computer Networks and ISDN Systems, **14** (1), pp. 25-59, 1987.

Bomsdorf, B., The WebTaskModel Approach to Web Process Modelling, in Proceedings of *Task Models and Diagrams for User Interface Design* Toulouse, France, pp. 240-253, 2007.

Brinksma, E., G. Scollo and Steenbergen, LOTOS specifications, their implementations, and their tests, in Proceedings of *IFIP Workshop Protocol Specification, Testing, and Verification VI*, pp. 349-360, 1987.

Butler, G., P. Grogono and F. Khendek, A Z Specification of Use Cases, in Proceedings of *APSEC 1998*, pp. 94-101, 1998.

Butler, M., A CSP Approach to Action Systems, PhD Thesis in *Computing Laboratory*, Oxford University, Oxford, 1992.

Card, S., T. P. Moran and A. Newell, The Psychology of Human Computer Interaction, 1983.

Cheung, K. S., T. Y. Cheung and K. O. Chow, A petri-net-based synthesis methodology for use-case-driven system design, in Journal of Systems and Software 79 (6), pp. 772 - 790, 2006.

Clemmensen, T. and J. Norbjerg, Separation in Theory – Coordination in Practice, in Workshop entitled "*Bridging the Gap between Software Engineering and Human-Computer Interaction*", Portland, Oregon, 2003.

Cockburn, A., Writing Effective Use Cases, Addison-Wesley, Boston, 2001.

Constantine, L., R. Biddle and J. Noble, Usage-Centered Design and Software Engineering: Models for Integration, in Workshop entitled "*Bridging the Gaps Between Software Engineering and Human-Computer Interaction*", Portland, Oregon, 2003.

Cost, R. S., C. Ye, W. F. Timothy, L. Yannis and P. Yun, Using Colored Petri Nets for Conversation Modeling, Springer-Verlag, 2000.

De Nicola, R., Extensional Equivalences for Transition Systems, in Acta Informatica, **24**, pp. 211-237, 1987.

Desjardins, AccèsD [Internet], Available from https://accesd.desjardins.com/en/accesd, Accessed: April 2008, Last Update: 2008.

Dijkstra, E. W., A discipline of programming, Prentice-Hall, Englewood Cliffs, N.J., 1976.

Dittmar, A., F. Forbrig, S. Stoiber and C. Stary, Tool Support for Task Modelling - A Constructive Exploration, in Proceedings of *Design, Specification and Verification of Interactive Systems 2004*, July 2004.

Dittmar, A. and P. Forbrig, Higher-Order Task Models, in Proceedings of *Design, Specification and Verification of Interactive Systems 2003*, pp. 187-202, 2003.

Eshuis, R., Symbolic model checking of UML activity diagrams, in ACM Transactions on Software Engineering and Methodology, **15 (1)**, pp. 1-38, 2006.

Eshuis, R. and R. Wieringa, Requirements Level Semantics for UML Statecharts, in Proceedings of *Formal Methods for Open Object-Based Distributed Systems IV*, Stanford, California, 2000.

Fernandes, J., S. Tjell, J. B. Jorgensen and O. Ribeiro, Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net, in

Proceedings of *Sixth International Workshop on Scenarios and State Machines (SCESM'07)*, Minneapolis, MN, IEEE Computer Society, 2007.

Fröhlich, P. and J. Link, Automated Test Case Generation from Dynamic Models, in Proceedings of *ECOOP'00*, Sophia Antipolis and Cannes, France pp. 472-492, 2000.

Garavel, H., F. Lang, R. Mateescu and W. Serwe, CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes, in Proceedings of *19th International Conference on Computer Aided Verification CAV 2007*, Berlin, Germany, 2007.

Gomaa, H., Designing Software Product Lines with UML, Addison-Wesley, 2005.

Grieskamp, W., M. Lepper, W. Schulte and N. Tillmann, Testable Use Cases in the Abstract State Machine Language, in Proceedings of *Second Asia-Pacific Conference on Quality Software*, IEEE Computer Society, 2001.

Harel, D., Statecharts: A Visual Formalism for Complex Systems, in Science of Computer Programming, **8**, pp. 231-274, 1986.

Hennessy, M., Acceptance Trees, in Associations for Computing Machinery, **32** (4), pp. 896-928, 1985.

Hoare, C. A. R., Communicating sequential processes, Prentice/Hall International, Englewood Cliffs, N.J., 1985.

Hopcroft, J. E., R. Motwani and J. D. Ullman, Introduction to automata theory, languages, and computation, 3rd edition, Pearson/Addison Wesley, 2007.

Interactions, I.-I. P. S.-O. S., ISO 8807: LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, 1987.

ITU-T, Recommendation Z.120- Message Sequence Charts, Geneva, 1996.

ITU-T, Recommendation Z.120- Message Sequence Charts, Geneva, 1999.

Jacobson, I., Object-Oriented Software Engineering : A Use Case Driven Approach, ACM Press (Addison-Wesley Pub), New York, 1992.

Jones, C. B., Systematic software development using VDM, Prentice Hall, New York, 1990.

Juristo, H., H. Windl and L. Constantine, Usability Engineering in Software Development, in IEEE Software, **18** (1), 2001.

Katoen, J. P. and L. Lambert, Pomsets for Message Sequence Charts, in Proceedings of *Formale Beschreibungstechniken für verteilte Systeme*, Cottbus, Germany, Shaker Verlag, pp. 197-207, 1998.

Kazman, R., L. Bass and B. John (2004). Bridging the gaps between software engineering and human-computer interaction, Workshop at ICSE 2004. Scotland, UK.

Kazman, R., J. Gunaratne and B. Jerome, Why Can't Software Engineers and HCI Practitioners Work Together?, in Proceedings of *HCI International '03*, Crete, Greece, pp. 504-508, 2003.

Keller, R., Formal Verification of Parallel Programs, in Communications of the ACM, **19**, pp. 561-572, 1976.

Khendek, F. and G. Bochmann, Merging Behavior Specifications, in Formal Methods in System Design, **6**, pp. 259-293, 1995.

Khendek, F., S. Bourduas and D. Vincent, Stepwise Design with Message Sequence Charts, in Proceedings of *Formal Techniques for Networked and Distributed Systems (FORTE)*, Cheju Island, Korea, pp. 19-34, 2001.

Klug, T. and J. Kangasharju, Executable task models, in Proceedings of *4th international workshop on Task models and diagrams*, Gdansk, Poland, pp. 119-122, 2005.

Kryvyy, S. and L. Matvyeyeva, Algorithm of Translation of MSC-specified System into Petri Net, in Fundamenta Informaticae, **79 (3-4)**, pp. 431-445, 2007.

Kujala, S., Linking User Needs and Use Case-Driven Requirements Engineering, chapter in *Human-Centered Software Engineering - Integrating Usability in the Development Process*, Springer, pp. 113-125, 2005.

Kuutti, K., Activity theory as a potential framework for human-computer interaction research, chapter in *Context and consciousness: activity theory and human-computer interaction*, Massachusetts Institute of Technology, pp. 17-44, 1995.

Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), Prentice Hall PTR, 2004.

Leffingwell, D. and D. Widrig, Managing software requirements : a use case approach, Addison-Wesley, Boston, 2003.

Li, L., Translating Use Cases to Sequence Diagrams, in Proceedings of *IEEE ASE 2000* Grenoble, France, pp. 293-296, 2000.

Linz, P., An introduction to formal languages and automata, 4th ed., Jones and Bartlett Publishers, 2006.

Lu, S., C. Paris, K. V. Linden and N. Colineau, Generating UML Diagrams from Task Models, in Proceedings of *CHINZ'03*, Dunedin, New Zealand, pp. 9-14, 2003.

Magee, J. and J. Kramer, Concurrency: State Models & Java Programs, John Wiley & Sons, 1999.

Marr, C., Capturing Conflict and Confusion in CSP, in Proceedings of *Integrated Formal Methods*, Oxford, UK, pp. 413-438, 2006.

Martin, J., Introduction to Languages and the Theory of Computation, 3rd Edition, McGraw-Hill Science, 2002.

Marucci, L., F. Paternó and C. Santoro, Supporting Interactions with Multiple Platforms Through User and Task Models, chapter in *Multiple User Interfaces, Cross-Platform Applications and Context-Aware Interfaces*. London, Wiley, pp. 217-238, 2003.

Mauw, S. and M. A. Reniers, An Algebraic Semantic of Basic Message Sequence Charts, in Computer Journal, **37** (4), 1994.

Mayhew, D., Usability Engineering Lifecycle, Morgan Kauffman, San Francisco, 1999.

Mazurkiewicz, A., Introduction to Trace Theory, chapter in *The book of traces*. V. Diekert and G. Rozenberg. Singapore, World Scientific, 1995.

Merrick, P. and P. Barrow, The Rationale for OO Associations in Use Case Modelling, in Journal of Object Technology, **4** (9), pp. 123-142, 2005.

Milner, R., A calculus of communicating systems, Springer-Verlag, Berlin ; New York, 1980.

Mizouni, R., Formal Composition of Partial System Behaviors, PhD Thesis in *Department of Electrical and Computer Engineering*, Concordia University, Montreal, 2007.

Mizouni, R., A. Salah, S. Kolahi and R. Dssouli, Merging partial system behaviours: composition of use-case automata, in Software, IET, **1** (4), pp. 143-160, 2007.

Morgan, C., The specification statement, in ACM Trans. Program. Lang. Syst., **10** (3), pp. 403-419, 1988.

Mori, G., F. Paternò; and C. Santoro, CTTE: Support for Developing and Analyzing Task Models for Interactive System Design, in IEEE Trans. Softw. Eng., **28** (8), pp. 797-813, 2002.

Nipkow, T., L. Paulson and M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Springer, 2008.

Overgaard, G. and K. Palmkvist, Use Cases Patterns and Blueprints, Addison-Wesley, Indianapolis, 2004.

Park, D., Concurrency and Automata in Infinite Strings, in Lecture Notes in Computer Science **104**, pp. 167-183, 1981.

Paternò, F., Model-Based Design and Evaluation of Interactive Applications, Springer, 2000.

Paternò, F. and C. Santoro, The ConcurTaskTrees Notation for Task Modelling, Technical Report in CNUCE-C.N.R, 2001.

Paternò, F. and C. Santoro, Support for Reasoning about Interactive Systems through Human–Computer Interaction Designers' Representations, in The Computer Journal, **48** (4), pp. 340-357, 2003.

Paternò, F., C. Santoro, J. Mäntyjärvi, G. Mori and S. Sansone, Authoring Pervasive MultiModal User Interfaces, in International Journal of Web Engineering and Technology, pp. 235-261, 2008.

Petri, C. A., Fundamentals of a theory of asynchronous information flow, in Proceedings of *IFIP Congress'62*, pp. 386-390, 1962.

Pratt, V. P., Modeling Concurrency with Partial Orders, in International Journal of Parallel Programming, **15** (1), pp. 33-71, 1986.

Pressman, R. S., Software engineering: A practitioner's approach, McGraw-Hill, Boston, Mass., 2005.

Pruitt, J. and J. Grudin, Personas: practice and theory, in Proceedings of *Designing for User Experiences (DUX '03)*, San Francisco, California, pp. 1-15, 2003.

Puerta, A., A Model-Based Interface Development Environment in IEEE Software, **14** (4), pp. 40-47, 1997.

Regnell, B. L., M. Andersson and J. Bergstrand, A Hierarchical Use Case Model with Graphical Representation, chapter in *Proceedings of the IEEE Symposium and Workshop on Engineering of Computer Based Systems*, IEEE Computer Society, pp. 270, 1996.

Roscoe, A. W., Model-checking CSP, chapter in *A Classical Mind: essays in Honour of C.A.R. Hoare*, Prentice-Hall, 1994.

Roscoe, A. W., The Theory and Practice of Concurrency, Prentice-Hall (Pearson), 2005.

Rosenberg, D. and M. Stephens, Use Case Driven Object Modeling with UML: ICONIX Process in Theory and Practice, Addison-Wesley, 2006.

Royce, W., Managing the Development of Large Software Systems, in Proceedings of *IEEE WESCON* Los Angeles, CA, 1970.

Rui, K., Refactoring Use Case Models, PhD Thesis in *Department of Computer Science and Software Engineering*, Concordia University, Montreal, 2007.

Scott, K., The Unified Process Explained Addison-Wesley 2001.

Seffah, A., M. C. Desmarais and M. Metzger, Software and Usability Engineering: Prevalent Myths, Obstacles and Integration Avenues, chapter in *Human-Centered Software Engineering -Integrating Usability in the Software Development Lifecycle*, Springer, 2005.

Sinha, A., A. Paradkar and C. Williams, On Generating EFSM Models from Use Cases, in Proceedings of *SCESM 2007*, Minneapolis, MN, 2007.

Sinnig, D., P. Chalin and F. Khendek, Towards a Common Semantic Foundation for Use Cases and Task Models, in Electronic Notes in Theoretical Computer Science (ENTCS), **183C**, 2006.

Sinnig, D., P. Chalin and F. Khendek, Common Semantics for Use Cases and Task Models, in Proceedings of *Integrated Formal Methods*, Oxford, England, 2007.

Sinnig, D., P. Chalin and F. Khendek, Consistency between Task Models and Use Cases, in Proceedings of *DSV-IS 2007*, Salamanca, Spain, 2007.

Sinnig, D., M. Wurdel, P. Forbrig, P. Chalin and F. Khendek, Practical Extensions for Task Models in Proceedings of *TaMoDia '07*, Toulouse, France Springer, 2007.

Solms, F., Business Process Modeling using URDAD, Technical Report in Solms Consulting, 2005.

Somé, S., Petri Nets Based Formalization of Textual Use Cases, Technical Report in SITE, TR-2007-11, University of Ottawa, 2007.

Souchon, N., Q. Limbourg and J. Vanderdonckt, Task Modelling in Multiple contexts of Use, in Proceedings of *Design, Specification and Verification of Interactive Systems*, Rostock, Germany, pp. 59-73, 2002.

Spivey, J. M., The Z notation : a reference manual, Prentice Hall, Englewood Cliffs, N.J., 1989.

Sutcliffe, A., Convergence or Competition between Software Engineering and Human Computer Interaction, chapter in *Human-Centered Software Engineering -Integrating Usability in the Software Development Lifecycle*. A. Seffah, M. C. Desmarais and M. Metzger, Springer, pp. 71-83, 2005.

UML, Unified Modeling Language [Internet], Available from http://www.uml.org/, Accessed: June 2004, Last Update: 2004.

van den Bergh, J., Coninx, K. , From Task to Dialog Model in the UML, in Proceedings of *TaMoDia 2007*, Toulouse, France, pp. 98-111, 2007.

Wagner, F., R. Schmuki, T. Wagner and P. Wolstenholme, Modeling Software with Finite State Machines: A Practical Approach Auerbach, 2006.

Winskel, G., Events in Computation, PhD Thesis in *Department of Computer Science*, University of Edinburg, Edinburg, 1980.

Wurdel, M., D. Sinnig and P. Forbrig, Task-based Development Methodology for Collaborative Environments, in Proceedings of *TaMoDia 2008*, Pisa, Italy, 2008.

Zheng, T., Validation and Refinement of Timed MSC Specifications, PhD Thesis in *Department of Electrical and Computer Engineering*, Concordia University, Montreal, 2004.

Zuberek, W. M., Timed Petri nets and preliminary performance evaluation, in Proceedings of *7th annual symposium on Computer Architecture*, La Baule, United States, ACM, pp. 88-96, 1980.

Appendix A Symbols and Nomenclature

The purpose of this chapter is to summarize the nomenclatural conventions adopted in this thesis and to define the meaning of frequently used symbols.

Universal Sets are represented by three or more capital letters in italic font. Sub-sets of universal sets are represented by a single capital (possibly Greek) letter. The most important symbols together with their denotations are listed below.

Symbol	**Universal Set**
DSRGUCM	Set of all DSRG use case models
UCNAME	Set of all DSRG-style use case names
USECASE	Set of all DSRG-style use cases
ECTT	Set of all ECTT task models
TASKEXPR	Set of all ECTT task expressions
TASKNAME	Set of all ECTT task names
UCLTS	Set of all use case labeled transition systems
GTM	Set of all generic task models
GTE	Set of all generic task expression
POSET	Set of all posets
SPOSET	Set of all sets of posets
STATE	Set of all states
EVENTNAME	Set of all event names
EVENT	Set of all events
NFSM	Set of all nFSMs
AGRAPH	Set of all acceptance graphs
Symbol	**Subsets of Universal Sets**
$\Sigma \in \mathbb{P}\, EVENTNAME$	Set of event names
$E \in \mathbb{P}\, EVENT$	Set of events
$Q \in \mathbb{P}\, STATE$	Set of states
$T \in \mathbb{P}\, TASKNAME$	Set of atomic generic task expressions

Elements of the abovementioned universal sets can be denoted by **variables** and **constants**. The most important ones are listed below.

Variable	**Description**
$uc \in USECASE$	DSRG-style use case
$D \in DSRGUCM$	DSRG-style use case model

$\upsilon, \varphi, o \in ECTT$	ECTT task expressions
$C \in ECTT$	ECTT task model
$U \in UCLTS$	Use case labeled transition system
$\psi, \rho \in GTE$	Generic task expressions
$G \in GTM$	Generic task model
$e \in EVENT$	Event
$P, R \in SPOSET$	Sets of posets
$M \in NFSM$	Nondeterministic finite state machine
$AG \in NFSM$	Acceptance graph
Constant	**Description**
$\emptyset_{poset} \in POSET$	Empty poset
$e_{poset} \in POSET$	Singleton poset containing event e
$Skip_{nFSM} \in NFSM$	Skip nFSM consisting of only a single (final) state
$e_{nFSM} \in NFSM$	Singleton nFSM consisting of two states (initial and final, respectively) connected by a transition triggered by e

The symbols and signatures of the most important **semantic functions** defined in this thesis are given below.

Semantic Function	Signature
$\mathcal{M}_{DsgrUclts}$	$DSRGUCM \longrightarrow UCLTS$
$\mathcal{M}_{EcttGtm}$	$ECTT \longrightarrow GTM$
$\mathcal{M}_{UcltsSposet}$	$UCLTS \longrightarrow SPOSET$
$\mathcal{M}_{GtmSposet}$	$GTM \longrightarrow SPOSET$
$\mathcal{M}_{UcltsNfsm}$	$UCLTS \longrightarrow NFSM$
$\mathcal{M}_{GtmNfsm}$	$GTM \longrightarrow NFSM$

We use the following notations to define **sets**, **sequences** and **functions**.

Expression	Description
$\emptyset$	Denotes the empty set
$\{e_1, e_2, \ldots, e_n\}$	Denotes the set containing elements $e_1, e_2, \ldots, e_n$
$\{x \in A \mid p(x)\}$	Denotes the set containing all elements from A for which predicate p holds true
$\langle\ \rangle$ or λ	Denote the empty sequence
$\langle e_1, e_2, \ldots, e_n \rangle$	Denotes the sequence of elements $e_1, e_2, \ldots, e_n$
$A \rightarrow B$	Denotes a total function from A to B
$A \nrightarrow B$	Denotes a partial function from A to B

Appendix B Overview of Isabelle Syntax

In this chapter we briefly discuss the Isabelle theorem prover and introduce relevant commands with respect to the Isabelle/HOL theories presented in this thesis. Interacting with Isabelle is done via theory files in which the user can declare types, constants, rules, and functions. Isabelle provides a set of build-in types as well as user defined types, including recursive ones. Rules come in two forms, axioms and theorems or formulas. Functions can be recursive and make use of pattern matching on their associated arguments. There exist two types of recursive functions: (1) primitive recursive functions (`primrec`) that operate exclusively on recursive data types and (2) general purpose recursive functions (`recdef`).

`theorem` or `lemma` are the Isabelle commands for specifying propositions. Proofs for such propositions are performed using proof tactics that allow for the automation of the proving process. Most notably are the tactics `simp`, `auto`, and `blast`. `simp` merely applies simplification rules to a formula. `auto` and `blast` perform simplification first and then use brute force in an attempt to satisfy the proof. A summary of the most important Isabelle/HOL commands is given in Table B.1. A more comprehensive overview of the Isabelle syntax can be found in [Nipkow et al. 2008].

Table B.1 Summary of Isabelle Commands

Keyword	Description
`theory`	Defines a new Isabelle theory
`datatype`	Defines a new HOL datatype
`record`	Defines a new record type by specifying its fields
`definition`	Defines a non-recursive function
`primrec`	Defines a primitive recursive function that allows pattern matching
`types`	Defines a synonym for an existing datatype
`consts`	Declares a constant
`constdefs`	Declares a constant and defines its value
`lemma`	Defines a proposition which is followed by a proof
`apply`	Denotes the application of a tactic to advance a proof of a lemma

Appendix C Rewriting of Disabling and Suspend / Resume

In this section we give formal definitions of the auxiliary operators *deep optionalization* ($\mathcal{O}$) and *interleaved insertion* ($\mathcal{I}$). Both are needed in Definition 5.16 for the rewriting of the ECTT operators *disabling* und *suspend/resume*, respectively.

Intuitively the meaning of the *disabling* operator is defined as follows: Both tasks specified by its operands are enabled concurrently. As soon as the first (sub-) task specified by the second operand is executed, the task specified by the first operand becomes disabled. If the execution of the task(s) specified by the first operand is completed (without interruption) the task(s) specified by the second operand are subsequently executed. In other words, none of the (sub-) tasks of the first operand must necessarily be executed, whereas the execution of the tasks of the second operand is mandatory. Hence, an ECTT task expression including the *disabling* operator can be rewritten as the optional execution of the *deep optionalization* ($\mathcal{O}$) of all tasks involved in the first operand, followed by the execution of the second operand ($\upsilon \; [> \; f = \mathcal{O}[\![\upsilon]\!]_{\mathcal{D}} \gg f$). We note that the definition of the CTT *disabling* operator has been inspired by the disabling operator of the LOTOS process algebra [Interactions 1987]. Yet, the interpretations of both operators are *not* identical. In particular, in LOTOS the subsequent execution of the second operand, after completion of the first one is not allowed.

Definition C.1 (Deep Optionalization). Let υ, φ be ECTT task expressions, n be a task identifier and $\mathcal{D}$ be a finite map of ECTT task definitions. We then define the operator *deep optionalization* ($\mathcal{O}$) inductively as follows:

$$\mathcal{O}[\![n]\!]_{\mathcal{D}} = \begin{cases} \mathcal{O}[\![\mathcal{D}(n)]\!]_{\mathcal{D}}, & \text{if } n \in dom(\mathcal{D}) \\ n, & \text{otherwise} \end{cases}$$

$$\mathcal{O}[\![\upsilon \gg \varphi]\!]_{\mathcal{D}} = [\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}} \gg [\mathcal{O}[\![\varphi]\!]_{\mathcal{D}}]]$$

$$\mathcal{O}[\![\upsilon \; [\,] \; \varphi]\!]_{\mathcal{D}} = [\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}} \; [\,] \; \mathcal{O}[\![\varphi]\!]_{\mathcal{D}}]$$

$$\mathcal{O}[\![\upsilon \parallel \varphi]\!]_{\mathcal{D}} = [\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}] \parallel [\mathcal{O}[\![\varphi]\!]_{\mathcal{D}}]$$

$$\mathcal{O}[\![\upsilon \boxplus \varphi]\!]_{\mathcal{D}} = \mathcal{O}[\![\upsilon \gg \varphi]\!]_{\mathcal{D}} \; [\,] \; \mathcal{O}[\![\varphi \gg \upsilon]\!]_{\mathcal{D}}$$

$$\mathcal{O}[\![\upsilon [> \varphi]\!]_{\mathcal{D}} = \mathcal{O}[\![[\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}] \gg \varphi]\!]_{\mathcal{D}}$$

$$O[\![v \mid > \varphi]\!]_D = O[\![\mathcal{I}[\![v]\!]_D \varphi^*]\!]_D$$

$$O[\![[v]]\!]_D = [O[\![v]\!]_D]$$

$$O[\![v^*]\!]_D = v^* \gg [O[\![v]\!]_D]$$

$$O[\![v^+]\!]_D = v^* \gg [O[\![v]\!]_D]$$

$$O[\![stop(v)]\!]_D = stop(O[\![v]\!]_D)$$

$$O[\![resume(v)]\!]_D = resume(O[\![v]\!]_D)$$

The interpretation of the *suspend/resume* operator is similar to the one of the *disabling* operator. Both tasks specified by its operands are enabled concurrently. At any time the execution of the first operand can be interrupted by the execution of the first (sub-) task of the second operand. An exception to this rule are tasks within the scope of the concurrency operator (||). Such task, although interrupted, may (concurrently) continue their execution[12]. Contrary to the *disabling* operator, the execution of the task specified by the first operand is only suspended and will (once the execution of the second operand is complete) be reactivated from the state reached before the interruption [Paternò 2000]. At this point, the tasks specified by the first operand may continue its execution or may be interrupted again by the execution of the second operand.

In order to model this behavior, we have defined the auxiliary binary operator *interleaved insertion* ($\mathcal{I}$). It "injects" the task specified by its second operand at any possible position in between the (sub-) tasks of the first operand. Using the auxiliary operator it is now possible to rewrite a term containing the *suspend/resume* operator as follows: $v \mid > \varphi = \mathcal{I}[\![v]\!]_D \varphi^*$.

[12] This is not allowed according to the informal CTT semantics.

Definition C.2 (Interleaved Insertion). Let v, φ, o be ECTT task expressions, n be a task identifier and $\mathcal{D}$ be a finite map of ECTT task definitions. We then define the operator *interleaved insertion* ($\mathcal{I}$) inductively as follows:

$$\mathcal{I}[\![n]\!]_{\mathcal{D}}\ o = \begin{cases} \mathcal{I}[\![\mathcal{D}(n)]\!]_{\mathcal{D}}\ o, & if\ n \in dom(\mathcal{D}) \\ o \gg n, & otherwise \end{cases}$$

$$\mathcal{I}[\![v \gg \varphi]\!]_{\mathcal{D}}\ o = \mathcal{I}[\![v]\!]_{\mathcal{D}}\ o \gg \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ o$$

$$\mathcal{I}[\![v\ [\,]\ \varphi]\!]_{\mathcal{D}}\ o = \mathcal{I}[\![v]\!]_{\mathcal{D}}\ o\ [\,]\ \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ o$$

$$\mathcal{I}[\![v \parallel \varphi]\!]_{\mathcal{D}}\ o = \mathcal{I}[\![v]\!]_{\mathcal{D}}\ o \parallel \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ o$$

$$\mathcal{I}[\![v \boxplus \varphi]\!]_{\mathcal{D}}\ o = \mathcal{I}[\![v]\!]_{\mathcal{D}}\ o \gg \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ o\ [\,]\ \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ o \gg \mathcal{I}[\![v]\!]_{\mathcal{D}}\ o$$

$$\mathcal{I}[\![v >\ \varphi]\!]_{\mathcal{D}}\ o = \mathcal{I}[\![[\mathcal{O}[\![v]\!]_{\mathcal{D}}] \gg \varphi]\!]_{\mathcal{D}}\ o$$

$$\mathcal{I}[\![v \mid >\ \varphi]\!]_{\mathcal{D}}\ o = \mathcal{I}[\![\mathcal{I}[\![v]\!]_{\mathcal{D}}\ \varphi^*]\!]_{\mathcal{D}}\ o$$

$$\mathcal{I}[\![[v]]\!]_{\mathcal{D}}\ o = [\mathcal{I}[\![v]\!]_{\mathcal{D}}\ o]$$

$$\mathcal{I}[\![v^*]\!]_{\mathcal{D}}\ o = (\mathcal{I}[\![v]\!]_{\mathcal{D}}\ o)^*$$

$$\mathcal{I}[\![v^+]\!]_{\mathcal{D}}\ o = (\mathcal{I}[\![v]\!]_{\mathcal{D}}\ o)^+$$

$$\mathcal{I}[\![stop(v)]\!]_{\mathcal{D}}\ o = stop(\mathcal{I}[\![v]\!]_{\mathcal{D}}\ o)$$

$$\mathcal{I}([\![resume(v)]\!], o) = resume\big(\mathcal{I}([\![v]\!], o)\big)$$

Appendix D Acceptance Graphs

Appendix D.1 Definitions

Definition D.1 (Acceptance Graph) An *Acceptance Graph AG* is defined as the following tuple: $AG = (Q, \Sigma, \chi, \delta, q_0, F_C, F_N)$, where

$Q \subseteq STATE$ is a finite set states,

$\Sigma \subseteq EVENTNAME$ is a finite set event names,

$\chi: Q \longrightarrow \mathbb{P}(\mathbb{P}(\Sigma))$ is a mapping from Q to a set of subsets of Σ. $\chi(q)$ is called the acceptance set of state q,

$\delta: Q \times \Sigma \longrightarrow Q$ is the transition function, which returns for a given state q_m and a given input symbol a the corresponding state q_n that can be reached from q_m by accepting a,

q_0 is the initial state with $q_0 \in Q$,

$F_C \subseteq Q$ is the set of final states used by the sequential and iterative composition,

$F_N \subseteq Q$ is the set of final states *not* used by the sequential and iterative composition.

Definition D.2 (Well-formedness Requirements). An Acceptance Graph $AG = (Q, \Sigma, \chi, \delta, q_0, F_C, F_N)$ is *well formed*, if the following conditions are satisfied [Hennessy 1985]:

R1. $\forall q \in Q.\ \chi(q) \neq \emptyset.$

R2. $\forall q_m \in Q, \forall \Sigma_A \in \chi(q_m), \exists!\, q_n \in Q.(\forall a \in \Sigma_A.\ (q_m, a, q_n) \in \delta).$

R3. $\forall q_m \in Q,\ \exists\, q_n \in Q, \exists a \in \Sigma_A.(q_m, a, q_n) \in \delta \Longrightarrow \Sigma_A \in \chi(q_m) \wedge a \in \Sigma_A.$

R4. $\forall q \in Q.\Sigma_{A1}, \Sigma_{A2} \in \chi(q) \Longrightarrow (\Sigma_{A1} \cup \Sigma_{A2}) \in \chi(q).$

R5. $\forall q \in Q.(\Sigma_{A1}, \Sigma_{A2} \in \chi(q) \wedge \exists \Sigma_{A3}.\Sigma_{A1} \subseteq \Sigma_{A3} \subseteq \Sigma_{A2}) \Longrightarrow \Sigma_{A3} \in \chi(q).$

We define a set of auxiliary functions that are needed for composing acceptance graphs and for the conversion to and from nFSMs.

Definition D.3 (Extended Transition Function). The *extended transition function* $\delta^*: Q \times \Sigma^* \longrightarrow Q$ is defined in a standard way as:

$$\delta^*(q_i, w) = q_j,$$

where q_j is the state the acceptance graph may be in, having started in state q_i and after having accepted w.

Definition D.4 (Initials). The *Initials* of a given state q are the set of event names accepted by q. We define $initial: Q \longrightarrow \mathbb{P}(\Sigma)$ as follows:

$$initial(q) = \{a \in \Sigma | \exists q_t \in Q.\ (q, a, q_t) \in \delta\}$$

Appendix D.2 Composition Operations for Acceptance Graphs

In this section we define the binary operations *sequential merge* (⊞), *choice merge* (⊗) and *parallel composition* (‖).

Definition D.5 (Sequential Merge). Let $AG_1 = (Q_1, \Sigma_1, \chi_1, \delta_1, q_{1_0}, F_{C1}, F_{N1})$ and $AG_2 = (Q_2, \Sigma_2, \chi_2, \delta_2, q_{2_0}, F_{C2}, F_{N2})$ be two acceptance graphs such that Q_1 and Q_2 are disjoint. We then define the *sequential merge* (⊞) as follows: $AG_1 \boxplus AG_2 = AG$ with $AG = (Q, \Sigma_1 \cup \Sigma_2, \chi, \delta, q_{1_0}, F_C, F_N)$ where

$$Q = \{(q_1, q_2) | q_1 \in Q_1, q_2 \in Q_2\} \cup Q_1 \cup Q_2$$

$$F_C = F_{C2} \cup \{(q_1, q_2) | q_1 \in Q_1 \wedge q_2 \in F_{C2}\}$$

$$F_N = F_{N1} \cup F_{N2} \cup \{(q_1, q_2) | q_1 \in F_{N1} \wedge q_2 \in F_{N2}\}$$

$$\chi\big((q_1, q_2)\big)$$

$$= \begin{cases} \chi_1(q_1) \cup \chi_2(q_2) \cup \chi_{1+2}, & \forall e \in \big(initials_1(q_1) \cup initials_2(q_2)\big).\tau(e) = application \\ \chi_{1+2}, & \forall e \in \big(initials_1(q_1) \cup initials_2(q_2)\big).\tau(e) = interaction \end{cases}$$

with

$$\chi_{1+2} = \{\Sigma_{A1} \cup \Sigma_{A2} | \Sigma_{A1} \in \chi_1(q_1), \Sigma_{A2} \in \chi_2(q_2)\}$$

$$\chi(q) = \begin{cases} \chi_1(q), & q \in Q_1 \\ \chi_2(q), & q \in Q_2 \end{cases}$$

$$\delta(q, a) = \begin{cases} \delta_1(q, a), & q \in Q_{A1} \wedge q \notin F_{C1} \wedge a \in \Sigma_1 \cup \{\lambda\} \\ \delta_2(q, a), & q \in Q_2 \wedge a \in \Sigma_2 \cup \{\lambda\} \\ \delta_2(q_{2_0}, a) & q \in F_{C1} \wedge a \notin initial_1(q) \wedge a \in initial_2(q_{2_0}) \\ \big(\delta_1(q, a), \delta_2(q_{2_0}, a)\big) & q \in F_{C1} \wedge a \in initial_1(q) \wedge a \in initial_2(q_{2_0}) \end{cases}$$

$$\delta((q_1,q_2),a) = \begin{cases} (\delta_1(q_1),\delta_2(q_2)), & a \in initial_1(q_1) \wedge a \in initial_2(q_2) \\ \delta_1(q_1), & a \in initial_1(q_1) \wedge a \notin initial_2(q_2) \\ \delta_2(q_2), & a \notin initial_1(q_1) \wedge a \in initial_2(q_2) \end{cases}$$

Definition D.6 (Choice Merge). Let $AG_1 = (Q_1, \Sigma_1, \chi_1, \delta_1, q_{1_0}, F_{C1}, F_{N1})$ and $AG_2 = (Q_2, \Sigma_2, \chi_2, \delta_2, q_{2_0}, F_{C2}, F_{N2})$ be two acceptance graphs such that Q_1 and Q_2 are disjoint. We then define the operation *choice merge* ($\otimes$) as follows: $AG_1 \otimes AG_2 = AG$ with

$AG = (Q, \Sigma_1 \cup \Sigma_2, \chi, \delta, q_0, F, C), where$

$Q = \{(q_1,q_2) | \; q_1 \in Q_1, q_2 \in Q_2\} \cup Q_1 \cup Q_2$

$q_0 = (q_{1_0}, q_{2_0})$

$F_C = \{(q_1,q_2) | \; q_1 \in F_{C1} \vee q_2 \in F_{C2}\} \cup F_{C1} \cup F_{C2}$

$F_N = \{(q_1,q_2) | \; q_1 \in F_{N1} \vee q_2 \in F_{N2}\} \cup F_{N1} \cup F_{N2}$

$\chi((q_1,q_2))$

$$= \begin{cases} \chi_1(q_1) \cup \chi_2(q_2) \cup \chi_{1+2}, & \forall e \in (initials_1(q_1) \cup initials_2(q_2)).\tau(e) = application \\ \chi_{1+2}, & \forall e \in (initials_1(q_1) \cup initials_2(q_2)).\tau(e) = interaction \end{cases}$$

with

$\chi_{1+2} = \{\Sigma_{A1} \cup \Sigma_{A2} | \; \Sigma_{A1} \in \chi_1(q_1), \Sigma_{A2} \in \chi_2(q_2)\}$

$$\chi(q) = \begin{cases} \chi_1(q), & q \in Q_1 \\ \chi_2(q), & q \in Q_2 \end{cases}$$

$$\delta(q,a) = \begin{cases} \delta_1(q,a), & q \in Q_1 \wedge a \in \Sigma_1 \cup \{\lambda\} \\ \delta_2(q,a), & q \in Q_2 \wedge a \in \Sigma_2 \cup \{\lambda\} \end{cases}$$

$$\delta((q_1,q_2),a) = \begin{cases} (\delta_1(q_1),\delta_2(q_2)), & a \in initial_1(q_1) \wedge a \in initial_2(q_2) \\ \delta_1(q_1), & a \in initial_1(q_1) \wedge a \notin initial_2(q_2) \\ \delta_2(q_2), & a \notin initial_1(q_1) \wedge a \in initial_2(q_2) \end{cases}$$

Definition D.7 (Parallel Composition of two AGs). Let $AG_1 = (Q_1, \Sigma_1, \chi_1, \delta_1, q_{1_0}, F_{C1}, F_{N1})$ and $AG_2 = (Q_2, \Sigma_2, \chi_2, \delta_2, q_{2_0}, F_{C2}, F_{N2})$ be two

acceptance graphs such that Q_1 and Q_2 are disjoint. We then define the operation *parallel composition* (||) as follows: $AG_1 \parallel AG_2 = AG$ with

$AG = (Q, \Sigma_1 \cup \Sigma_2, \chi, \delta, q_0, F, C)$ where

$$Q = \{(q_1, q_2) | \, q_1 \in Q_1, q_2 \in Q_2\} \cup Q_1 \cup Q_2$$

$$q_0 = (q_{1_0}, q_{2_0})$$

$$F_C = \{(q_1, q_2) | \, q_1 \in F_{C1} \wedge q_2 \in F_{C2}\}$$

$$F_N = \{(q_1, q_2) | \, (q_1 \in F_{N1} \wedge q_2 \in F_{N2}) \vee (q_1 \in F_{N1} \wedge q_2 \in F_{C2}) \vee (q_1 \in F_{C1} \wedge q_2 \in F_{N2})\}$$

$$\chi((q_1, q_2)) = \{\Sigma_{A1} \cup \Sigma_{A2} | \, \Sigma_{A1} \in \chi_1(q_1), \Sigma_{A2} \in \chi_2(q_2)\}$$

$$\delta((q_1, q_2), a) \begin{cases} (\delta_1(q_1), q_2), & a \in initial(q_1) \\ (q_1, \delta_2(q_2)), & a \in initial(q_2) \wedge a \notin initial(q_1) \end{cases}$$

Appendix D.3 Conversions between nFSMs and AGs

In this section we define the conversion operations to and from nFSMs as proposed in [Khendek & Bochmann 1995]. Without loss of generality, we assume that the input nFSM is divergence-free, i.e., it does not contain cycles consisting only of lambda transitions. (The construction of divergent nFSMs is not possible with the semantic mappings proposed in this thesis.) We start with a set of auxiliary functions for nFSMs, which are needed for the conversion.

Definition D.8 ($\lambda - closure$ of a State) Let $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ be an nFSM. We define the function $\lambda - closure: Q \longrightarrow \mathbb{P}(Q)$ as follows:

$$\lambda - closure(q) = \{q \in Q | (p, \lambda, q) \in \delta\}$$

Definition D.9 ($\lambda - Closure$ of a Set of States). Let $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ be an nFSM. We define the function $\lambda - Closure: \mathbb{P}(Q) \longrightarrow \mathbb{P}(Q)$ as follows:

$$\lambda - Closure(Q_i) = \bigcup_{q_j \in Q_i} \lambda - closure(q_j)$$

Definition D.10 (Conversion of an nFSM to a Corresponding AG). Let $M = (Q, \Sigma, \delta, q_0, F_C, F_N)$ be a divergence-free nFSM. We define the function $ag: NFSM \longrightarrow AGRAPH$, which generates the corresponding acceptance graph from M as follows: $ag(M) = (Q_A, \Sigma_A, \chi, \delta_A, q_{A_0}, F_{AC}, F_{AN})$ where:

$$Q_A = \mathbb{P}(Q)$$

$$\Sigma_A = \Sigma$$

$$\delta_A(\{q_1, q_2, \dots, q_i\}, a) = \lambda - Closure\big(\delta(q_1, a) \cup \delta(q_2, a) \cup \dots \cup \delta(q_i, a)\big)$$

$$q_{A_0} = \lambda - closure(q_0)$$

$$F_{AC} = \{Q_i \in Q_A \mid q \in Q_i, q \in F_C\}$$

$$F_{AN} = \{Q_i \in Q_A \mid q \in Q_i, q \in F_N\}$$

$$\chi(Q_i) = \left\{\Sigma_{AC} \mid \exists q_j \in Q_i.\ initial(q_j) \subseteq \Sigma_{AC} \subseteq \bigcup_{q_k \in Q_i} initial(q_k)\right\}$$

Definition D.11 (nFSM States Corresponding to an AG State). Let $AG = (Q, \Sigma, \chi, \delta, q_0, F_C, F_N)$ be an acceptance graph and $q \in Q$ be a state. We then define the function $fsmStates: Q \longrightarrow \mathbb{P}(Q)$, which returns the minimal states requires to "encode" q within an nFSM as follows:

$$fsmStates(q) = \left\{\Sigma_j \in \chi(q) \mid \nexists \Sigma_k, \nexists \Sigma_l \in \chi(q).\left(\left(\Sigma_j = \Sigma_k \cup \Sigma_l\right) \vee \left(\Sigma_k \subseteq \Sigma_j \subseteq \Sigma_l\right)\right)\right\}$$

Definition D.12 (Conversion of an AG to a Corresponding nFSM). Let $AG = (Q_A, \Sigma_A, \chi, \delta_A, q_{A_0}, F_{AC}, F_{AN})$ be an acceptance graph. We then define the function $nfsm(AG): AGRAPH \longrightarrow NFSM$, which generates the corresponding nFSM from AG as follows: $nfsm(AG) = (Q, \Sigma, \delta, q_0, F_C, F_N)$ where:

$$\Sigma = \Sigma_A$$

$$Q = \bigcup_{q_{A_i} \in Q_A} (fsmStates(q_{A_i}) \cup \{q_i\})$$

$$q_0 = q_0$$

$$F_C = \bigcup_{q_{A_i} \in F_{AC}} fsmStates(q_{A_i})$$

$$F_N = \bigcup_{q_{A_i} \in F_{AN}} fsmStates(q_{A_i})$$

$\delta = \delta_\lambda \cup \delta_{reg}$ with

$$\delta_\lambda = \bigcup_{q_{A_i} \in Q_A} \{(q_i, \lambda, q_{j_i}) \mid q_{j_i} \in fsmStates(q_{A_i})\}$$

$$\delta_{reg} = \bigcup_{q_{A_i} \in Q_A} \left\{\left(q_{j_i}, a, q_k\right) \mid q_{j_i} \in fsmStates(q_{A_i}), a \in \chi\left(q_{A_j}\right), \left(q_{A_j}, a, q_{A_k}\right) \in \delta_A\right\}$$

Appendix E Proofs

Appendix E.1 Proofs for Chapter 6

Proposition 6.1:

PROOF

Figure E.1 shows the Isabelle/HOL formalizations of a poset and its properties *reflexivity*, *antisymmetry* and *transitivity*.

Reflexivity

definition
reflexive :: ['a set, ('a × 'a) set] => *bool* where
reflexive A r == $r \subseteq A \times A$ & (*ALL x*: *A*. (*x*,*x*) : *r*)

Antisymmetry

definition
antisymmetric :: ['a set, ('a × 'a) set] => *bool* where
antisymmetric A r == $r \subseteq A \times A$ & (*ALL x y*. (*x*,*y*):*r* --> (*y*,*x*):*r* --> *x*=*y*)

Transitivity

definition
transitive :: ['a set, ('a × 'a) set] => *bool* where
transitive A r == $r \subseteq A \times A$ & (*ALL x y z*. (*x*,*y*):*r* --> (*y*,*z*):*r* --> (*x*,*z*):*r*)

We define a poset as *reflexive*, *antisymmetric* and *transitive*

consts *isPoset* :: ('a set × ('a × 'a) set) => *bool*
primrec
isPoset (*A*, *r*) = ((*reflexive A r*) & (*antisymmetric A r*) & (*transitive A r*))

Figure E.1. Isabelle/HOL Formalization of a Poset

The Isabelle/HOL formalizations of the operations *parallel composition*, *sequential composition* and *event hiding* are given in Figure E.2. In case of *parallel composition* and *sequential composition* we assume, without loss of generality, that the event sets of the respective operand posets are disjoint.

Parallel Composition of Posets

consts *par-comp* :: (('a set * ('a * 'a) set) * ('a set * ('a * 'a) set)) => ('a set * ('a * 'a) set)
primrec *par-comp* (*p1*, *p2*) = ((*fst p1* ∪ *fst p2*), (*snd p1* ∪ *snd p2*))

Sequential Composition of Posets

consts *seq-comp* :: (('a set * ('a * 'a) set) * ('a set * ('a * 'a) set)) => ('a set * ('a * 'a) set)
primrec *seq-comp* (*p1*, *p2*) = ((*fst p1* ∪ *fst p2*), {(*x*, *y*). ((*x* ∈ *fst p1*) & (*y* ∈ *fst p2*))} ∪ *snd p1* ∪ *snd p2*)

Event Hiding

consts *event hiding* :: (('a set * ('a * 'a) set) * ('a set)) => ('a set * ('a * 'a) set)
primrec *event-hiding* (*p1*, *E*) = ((*fst p1* − *E*), *snd p1* ∩ {(*x*,*y*). ((*x* ∈ (*fst p1* − *E*)) & (*y* ∈ (*fst p1* − *E*)))})

Event Substitution

consts *event-sub* :: (('a set * ('a * 'a) set) * ('a => 'b)) => ('b set * ('b * 'b) set)
primrec *event-sub* (*p1*, *m*) = ((*m* ' (*fst p1*)), {(*x*, *y*). *EX t*:*fst p1*. *EX s*:*fst p1*. (*x* = *m t*) & (*y* = *m s*) & (*t*,*s*) ∈ *snd p1*})

Figure E.2. Isabelle/HOL Formalizations of Poset Operations

The closure properties for the poset operation are formalized by a set of lemmas. Figure E.3 shows the lemmas and corresponding proofs for the closure of *sequential* and *parallel* composition. ■

Closure of Sequential Composition

Sub-lemma to break down the following two lemmas into suitable sub goals

lemma *compl-impl-rule*: ⟦(*A1* & *B1* & *E*) — *C1*: (*A2* & *B2* & *E*) — *C2*: (*A3* & *B3* & *E*) — *C3*⟧ ⟹ (*A1* & *A2* & *A3* & *B1* & *B2* & *B3* & *E*) — (*C1* & *C2* & *C3*)
apply (*auto*)
done

lemma *seq-comp-closure*: (*isPoset* (*A1*, *r1*) & *isPoset*(*A2*, *r2*) & (*A1* ∩ *A2* = {})) — *isPoset*(*seq-comp* ((*A1*, *r1*), (*A2*, *r2*)))
apply (*simp*)
apply (*rule compl-impl-rule*)
apply (*unfold reflexive-def*)
apply (*blast*)
apply (*unfold antisymmetric-def*)
apply (*blast*)
apply (*unfold transitive-def*)
by (*blast*)

Closure of Parallel Composition

lemma *par-comp-closure*: (*isPoset* (*A1*, *r1*) & *isPoset*(*A2*, *r2*) & (*A1* ∩ *A2* = {})) — *isPoset*(*par-comp* ((*A1*, *r1*), (*A2*, *r2*)))
apply (*simp*)
apply (*rule compl-impl-rule*)
apply (*unfold reflexive-def*)
apply (*blast*)
apply (*unfold antisymmetric-def*)
apply (*blast*)
apply (*unfold transitive-def*)
by *blast*

Figure E.3. Isabelle/HOL Proof of Closure for *Sequential* and *Parallel* Composition

Figure E.4 shows the lemmas and corresponding proofs for the closure of *event hiding* and *event renaming*.

Closure of Event Hiding

Sub lemma to break down the next lemma into suitable sub goals

lemma *simple-impl-rule*: ⟦*A* — *D*; *B* — *E*; *C* — *F*⟧ ⟹ (*A* & *B* & *C*) — (*D* & *E* & *F*)
apply (*auto*)
done

lemma *event-hiding-closure*: (*isPoset* (*A1*, *r1*) — *isPoset*(*event-hiding* ((*A1*, *r1*), *B*)))
apply (*simp*)
apply (*rule simple-impl-rule*)
apply (*unfold reflexive-def*)
apply (*blast*)
apply (*unfold antisymmetric-def*)
apply (*blast*)
apply (*unfold transitive-def*)
by (*blast*)

Closure of Event Substitution

Sub lemma to break down the next lemma into suitable sub goals

lemma *simple-impl-inj*: ⟦(*A* & *I*)— *D*; (*B* & *I*)— *E*; (*C* & *I*)— *F*⟧ —> (*A* & *B* & *C* & *I*) — (*D* & *E* & *F*)
apply (*auto*)
done

lemma *event-sub-closure*: (*isPoset* (*A*, *r*) & *inj-on f A*) — *isPoset* (*event-sub* ((*A*, *r*), *f*))
apply (*simp*)
apply (*rule simple-impl-inj*)
apply (*unfold reflexive-def*)
apply (*blast*)
apply (*unfold antisymmetric-def*)
apply (*unfold inj-on-def*)
apply (*blast*)
apply (*unfold transitive-def*)
apply (*blast*)
done
end

Figure E.4. Isabelle/HOL Proof of Closure of *Event Hiding* and *Event Renaming*

Proposition 6.2:

PROOF

Sub-Goal: $Tr(P \cdot R) = \{x \hat{} y \mid x \in Tr(p) \wedge y \in Tr(R) \wedge p = (E_p, \leq_p) \in P \wedge STOP \notin E_p\} \cup \{x \mid x \in Tr(p) \wedge p = (E_p, \leq_p) \in P \wedge STOP \in E_p\}$

From Definition 6.17 we know that $P \cdot R = \{p_i \cdot r_j \mid p_i \in P, r_j \in R\}$, which is the pairwise sequential composition of the posets entailed in P and R. From Definition 6.7 it follows that the sequential composition of two posets $p \cdot r$ either results in p (if p entails $STOP$) or results in a poset in which all events of p are causally related to events q (while their relative order as defined in p and q is preserved). Hence, we derive the following trace properties for the sequential composition of posets.

$$Tr(p \cdot q) = \begin{cases} Tr(p), & STOP \in name(E_p) \\ \{x \hat{} y \mid x \in Tr(p) \wedge y \in Tr(q)\}, & otherwise \end{cases}$$

By applying this observation to Definition 6.17 and Definition 6.15, we can state that the set of traces of $P \cdot R$ is the union of the set of traces of posets in P that contain $STOP$ $\{x \mid x \in Tr(p) \wedge p = (E_p, \leq_p) \wedge STOP \in E_p\}$ and the sequential composition of traces of posets in P (which do not contain $STOP$) with the set of traces of posets in R $\{x \hat{} y \mid x \in Tr(p) \wedge y \in Tr(R) \wedge p = (E_p, \leq_p) \in P \wedge STOP \notin E_p\}$. ∎

Sub-Goal: $Tr(P \parallel R) = \{x \,|||\, y \mid x \in Tr(P) \wedge y \in Tr(R)\}$

$P \parallel R$ is defined as the pairwise parallel composition of the posets entailed in P and the posets entailed in R (Definition 6.17). According to Definition 6.8 the parallel composition of two posets is the union of the respective event sets and the union of the partial order relations. No synchronization takes place on common events. Consequently, the set of traces of $p \parallel r$ is the set of all interleavings possible with the trace sets of p and r. That is, in order to obtain a trace, event names from p or r may be extracted in any order as long as the predefined partial order of the respective events in p and r is not violated. Such an interleaving of event names is modeled by the $|||$ operator, which was originally introduced for the definition of denotational trace semantics for the parallel, fully interleaved, composition of the two CSP processes [Roscoe 2005]. Since we know

that the set of traces of a set of posets is the union of the sets of traces of all entailed posets (Definition 6.15) we conclude that $Tr(P \parallel R) = \{x \,|||\, y \mid x \in Tr(P) \wedge y \in Tr(R)\}$ ■

Sub-Goal: $Tr(P \# R) = Tr(P) \cup Tr(Q)$

The validity of this statement follows directly from Definition 6.17 and Definition 6.15, according to which $P \# R = P \cup R$ and hence the set of all traces of $P \# R$ is the union of the trace set of P and the trace set of Q. ■

Sub-Goal: $Tr(P^*) = \bigcup_{k=0}^{\infty} Tr(P^k)$, while

$$Tr(P^k) = \begin{cases} \{\langle\ \rangle\}, & k = 0 \\ Tr(P), & k = 1 \\ \{x \hat{} y \mid x \in Tr(p) \wedge p = (E_p, \leq_p) \in P \wedge STOP \notin E_p \wedge y \in Tr(P^{k-1})\}, & k > 1 \end{cases}$$

According to Definition 6.18, the closure operation (*) for sets of posets is defined as follows: $P^* = \bigcup_{k=0}^{\infty} P^k$, with $P^k = \begin{cases} \{\emptyset_{poset}\}, & if\ k = 0 \\ P \cdot P^{k-1}, & k > 0 \end{cases}$

Hence, the set of traces of P^* equals to $Tr(\{\emptyset_{poset}\}) \cup Tr(P) \cup \ldots \cup Tr(P^k) \cup \ldots \cup Tr(P^\infty)$. In order to finish the proof of the sub-goal it is sufficient to show that (1) $Tr(\{\emptyset_{poset}\}) = (\{\langle\ \rangle\}$, (2) $Tr(P^1) = Tr(P)$ and (3) $Tr(P^k) = \{x \hat{} y \mid x \in Tr(p) \wedge p = (E_p, \leq_p) \in P \wedge STOP \notin E_p \wedge y \in Tr(P^{k-1})\}$, for k > 1. While (1) is proven trivially from Definition 6.6, the proof for (2) follows from the fact that $P^1 = P \cdot \{\emptyset_{poset}\} = P$. The validity of (3) follows form the fact $P^k = P \cdot P^{k-1}$ (Definition 6.18) and the conclusion of the first sub-goal, if we take into account that the set of traces of posets in P that contain $STOP$ is already included in the case trace set of P^1. ■

Sub-goals $Tr(close(P)) = Tr(P)$ and $Tr(open(P)) = Tr(P)$

The proof for the sub-goals follows directly from Definition 6.19 and the fact that, according to Definition 6.13, the $STOP$ event is not part of any trace. ■

Appendix E.2 Proofs for Chapter 7

Proposition 7.1:

PROOF:

Sub-goal: $Tr(M_1 \cdot M_2) = \{x \hat{} y \mid x \in Tr(M_1) \wedge y \in Tr(M_2) \wedge \delta^*_{AG_{M_1}}(q_0, x) \in F_{C_1}\} \cup \{x \mid x \in Tr(M_1) \wedge \delta^*_{AG_{M_1}}(q_0, x) \in F_{N_1}\}$

According to Definition 7.11, if two nFSMs M_1 and M_2 are composed sequentially, only the states in F_{C_1} participate in the sequential composition, while the states in F_{N_1} remain unchanged. The new initial state of $M_1 \cdot M_2$ is $q_0 = q_{0_1}$, the new sets of final states are $F_C = F_{C_2}$ and $F_N = F_{N_1} \cup F_{N_2}$ (Note that, due to the conversion to acceptance graphs and back, none of the states in M_1 and M_2 are technically preserved in $M_1 \cdot M_2$, but have been replaced by testing equivalent counterparts [Khendek & Bochmann 1995]). According to Definition 7.4, a trace of an nFSM is a run from q_0 to a state in F_C or F_N. From this follows that the set of all traces of $M_1 \cdot M_2$ is the union of the set of all runs from q_{0_1} to F_{N_1} $\{x \mid x \in Tr(M_1) \wedge \delta^*_{AG_{M1}}(q_0, x) \in F_{N_1}\}$ and the sequential composition of the runs of M_1 from q_{0_1} to F_{C_1} with the set of all accepted runs of M_2 $\{x \hat{} y \mid x \in Tr(M_1) \wedge y \in Tr(M_2) \wedge \delta^*_{AG_{M_1}}(q_0, x) \in F_{C_1}\}$. ■

Sub-goal: $Tr(M_1 \parallel M_2) = \{x \mid\mid\mid y \mid x \in Tr(M_1) \wedge y \in Tr(M_2)\}$

$M_1 \parallel M_2$ constructs the so-called product machine of M_1 and M_2 (Definition 7.11), which defines the interleaved execution of M_1 and M_2. Similar to the parallel composition of posets, a synchronization among common events does not take place. The state set of M is the Cartesian product of the state sets of M_1 and M_2. In any given state pair (q_1, q_2) the product machine either accepts a symbol which M_1 accepts in q_1 or which M_2 accepts in q_2 until a common final state has been reached. Hence, the set of accepted runs of $M_1 \parallel M_2$ is the set of all interleavings of the runs of M_1 and M_2, which corresponds to $\{x \mid\mid\mid y \mid x \in Tr(M_1) \wedge y \in Tr(M_2)\}$. ■

Sub-goal: $Tr(M_1 \# M_2) = Tr(M_1) \cup Tr(M_2)$

According to Definition 7.11, $M_1 \# M_2$ merges the initial states of M_1 and M_2 (and possibly, in case of identical acceptance sets, the respective subsequent states). Since the new sets of final states are defined as $F_C = F_{C_1} \cup F_{C_2}$ and $F_N = F_{N_1} \cup F_{N_2}$ we can conclude that the set of all traces of $M_1 \# M_2$ is the union of the trace set of M_1 and the trace set of M_2. ■

Sub-goal: $Tr(M^*) = \bigcup_{k=0}^{\infty} Tr(M)^k$ while

$$Tr(M)^k = \begin{cases} \{(\ )\}, & k = 0 \\ Tr(M), & k = 1 \\ \{x \hat{} y \mid x \in Tr(M) \wedge y \in Tr(M)^{k-1} \wedge \delta^*_{AG_{M_1}}(q_0, x) \in F_C\}, & k > 1 \end{cases}$$

According to Definition 7.12 the iterative composition of an nFSM M results in an nFSM where the all states in F_C are enriched with the "transitional behavior" of the initial state q_0. Additionally, q_0 is added to the set of final states. Hence, the set of all runs is the union of (1) the empty run ($k = 0$) and (2) the run from q_0 to any of the final states in F_C or F_N ($k = 1$) and (3) the set of runs that traverse any state in F_C (any number of times) and result in any final state ($k > 1$). ■

Sub-goals: $Tr\big(close(M)\big) = Tr(M)$ and $Tr\big(open(M)\big) = Tr(M)$

According to Definition 7.4, all the states in the set $F_C \cup F_N$ are accepting states. Since this set is preserved by the *close* and *open* operation (Definition 7.14) and the set of transitions remains unchanged, we conclude that $Tr\big(close(M)\big) = Tr(M)$ and $Tr\big(open(M)\big) = Tr(M)$ ■

Appendix E.3 Proofs for Chapter 8

Lemma 8.1:

PROOF

Sub-goal: $(\{\alpha_{poset}\} \equiv_{CTR} \alpha_{nFSM})$

With $\{\alpha_{poset}\} = \{(\{\alpha\}, \{(\alpha, \alpha)\})\}$ and $\alpha_{nFSM} = (\{q_0, q_f\}, \{\alpha\}, \{(q_0, \alpha, q_f)\}, q_0, \{q_f\}, \emptyset)$, clearly all three conditions of Definition 8.1 are satisfied. Both specifications have only one trace $\{\alpha\}$. With $\alpha \neq STOP$, the only poset of $\{\alpha_{poset}\}$ does clearly not contain the $STOP$ event and the only final state of α_{nFSM} is in F_C. ∎

Sub-goal: $(\{\emptyset_{poset}\} \equiv_{CTR} Skip_{nFSM})$

Clearly $Tr(\{\emptyset_{poset}\}) = \{\langle\ \rangle\} = Tr(M_{Skip})$. $\{\emptyset_{poset}\}$ contains a single poset, which does not contain $STOP$. Since in M_{Skip} $F_N = \emptyset$, we conclude that conditions (2) and (3) are satisfied. ∎

For the sake of brevity, we shall use, for the subsequent proofs, the following abbreviations for assumptions that follow directly from $P_1 \equiv_{CTR} M_1$ and $P_2 \equiv_{CTR} M_2$.

Table E.1 Statements Following Directly from $P_1 \equiv_{CTR} M_1$ and $P_2 \equiv_{CTR} M_2$

Abbr.	Assumptions, with $i \in \{1,2\}$
$IH1_i$	$Tr(P_i) = Tr(M_i)$
$IH2_i$	$\forall x \in Tr(P_i).\big((\exists p = (E_p, \leq_p) \in P_i.x \in Tr(p) \wedge STOP \in E_p) \Leftrightarrow (\delta_A^*(q_{A0_i}, x) \in F_{AN_i})\big)$
$IH3_i$	$\forall x \in Tr(P_i).\big((\exists p = (E_p, \leq_p) \in P_i.x \in Tr(p) \wedge STOP \notin E_p) \Leftrightarrow (\delta_A^*(q_{A0_i}, x) \in F_{AC_i})\big)$

Lemma 8.2:

PROOF ($P_1 \cdot P_2 \equiv_{CTR} M_1 \cdot M_2$): Let $P = P_1 \cdot P_2$ and $M = (Q, \Sigma, \delta, q_0, F_C, F_N) = M_1 \cdot M_2$ with $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, F_{C1}, F_{N1})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, F_{C1}, F_{C2})$ we prove that $P \equiv_{CTR} M$. We start by proving that P and M are trace equivalent ($Tr(P_1 \cdot P_2) = Tr(M_1 \cdot M_2)$). From Proposition 6.2 we know that the set of all traces of $P_1 \cdot P_2$ is the union $Tr_{P1STOP} \cup Tr_{P1P2}$. From Proposition 7.1 we know that the set of all traces of

$M_1 \cdot M_2$ is the union $Tr_{M_1F_{N1}} \cup Tr_{M_1M_2}$. In order to prove that $Tr(P) = Tr(M)$, we show that $Tr_{P1STOP} = Tr_{M_1F_{N1}}$ and that $Tr_{P1P2} = Tr_{M_1M_2}$.

Tr_{P1STOP} is the set of all traces of posets in P_1 that contain the $STOP$ event. $Tr_{M_1F_{N1}}$ is the set of all runs of M_1 from q_{0_1} to a state in F_{N1}. Since $p \cdot q = p$, if p contains the $STOP$ event and states in F_{N1} do not participate in the sequential composition with M_2 but remain final states of $M_1 \cdot M_2$ we conclude that traces in Tr_{P1STOP} and $Tr_{M_1F_{N1}}$ are preserved in $P_1 \cdot P_2$ and $M_1 \cdot M_2$. From $IH2_1$ we know that for each trace in Tr_{P1STOP} there is a run in M_1 from q_{0_1} to F_{N1} and vice versa (see Figure E.5). Hence we conclude that $Tr_{P1STOP} = Tr_{M_1F_{N1}}$.

Tr_{P1P2} is the set of all traces that are the result of the pairwise concatenation of the traces of posets in P_1 that *do not* contain $STOP$ ($Tr_{P1.NOSTOP}$) with the set of traces of P_2 ($Tr(P_2)$). Similarly $Tr_{M_1M_2}$ is the set of traces resulting from the pairwise concatenation of the runs of M_1 from q_0 to a state in F_{C1} with the set of all runs of M_2. From $IH3_1$ and $IH1_2$ it follows that the set of traces that participate in the computation of Tr_{P1P2} and $Tr_{M_1M_2}$ are identical (see Figure E.5). We conclude that $Tr_{P1P2} = Tr_{M_1M_2}$ and that $Tr(P_1 \cdot P_2) = Tr(M_1 \cdot M_2)$.

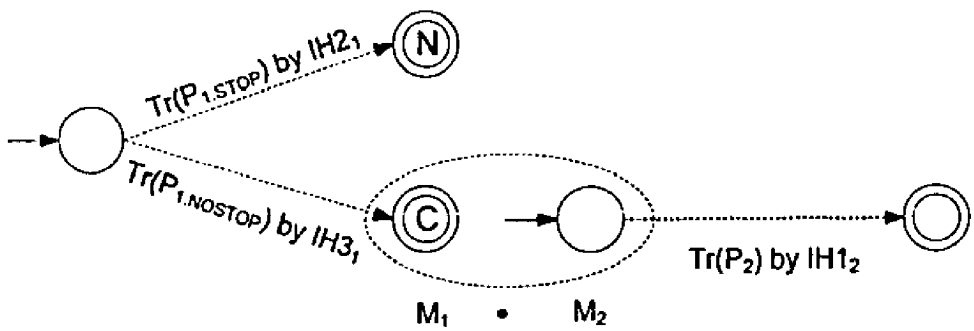


Figure E.5. Correspondence between Seq. Composition of Sets of Posets and nFSMs

We continue to show that conditions (2) and (3) are satisfied and start with the implication from left to right ($\Rightarrow$).

It follows from Definition 6.7 and Definition 6.17 that the sequential composition of sets of posets results into the following sub-sets $P = P_1 \cdot P_2 = P_{STOP.P_1} \cup P_{STOP.P_1P_2} \cup P_{P_1P_2}$. $P_{STOP.P_1}$ contains exactly the posets from P_1 that contain $STOP$. $P_{STOP.P_1P_2}$ is the set of

posets resulting from the pairwise sequential composition of the posets of P_1 that do not contain $STOP$ and the posets of P_2 that do contain $STOP$. $P_{P_1P_2}$ is the result of the pairwise sequential composition of the posets of P_1 and P_2, which do not contain $STOP$. To satisfy condition (2) we show that for each trace of $P_{STOP.P_1}$ and $P_{STOP.P_1P_2}$ there exist a run in M from q_0 to F_N.

According to Definition 7.11 the sequential composition of nFSMs merges the states in F_{C1} states with q_{0_2} while the states in F_{N1} remain unmodified ($F_{N1} \subseteq F_N$). Hence, all runs from q_{0_1} to F_{N1} in M_1 are also runs from q_0 to F_N in M. From $IH2_1$ we conclude that for each trace in $P_{STOP.P_1}$ there exist a run in M from q_0 to F_N. It also follows from Definition 7.11 that $F_{N2} \subseteq F_N$. Hence, for each run from q_{0_2} to F_{N2} in M_2 there must be a (set of) corresponding runs in M from q_0 to F_N, where each run is prefixed by a run from q_0 to F_{C1} in M_1. From $IH3_1$ and $IH2_2$ we conclude that for each trace of a poset in $P_{STOP.P_1P_2}$ there exist a run from q_0 to F_N in M.

To satisfy condition (3) we show that for each trace of a poset in $P_{P_1P_2}$ there exist a run in M from q_0 to a state in F_C. This proposition is discharged in a similar way to the previous case by taking into consideration that $F_C = F_{C2}$. Hence, for each run from q_{0_2} to a state in F_{C2} in M_2 there must be a (set of) corresponding runs in M from q_0 to F_C. The proof of the proposition follows directly from $IH3_1$ and $IH3_2$.

We complete the proof by showing that the implication from right to left ($\Leftarrow$) holds. Since, $F_C = F_{C1}$ and $F_N = F_{N1} \cup F_{N2}$, the set of traces of M can be split up as follows: $Tr(M) = \delta_{q_0F_{N1}} \cup \delta_{q_0F_{N2}} \cup \delta_{q_0F_C}$. $\delta_{q_0F_{N1}}$ represents the set of runs from q_0 to a state in F_{N1}. $\delta_{q_0F_{N2}}$ is the set of runs from q_0 through a state in F_{C1} (which was merged with q_{0_2}) to a state in F_{N2}. $\delta_{q_0F_C}$ is the set of runs from q_0 through a state in F_{C1} to a state in F_C. To satisfy condition (2) we show that for $\delta_{q_0F_{N1}}$ and $\delta_{q_0F_{N2}}$ there exists a set of posets in P which have the same set of traces and contain the $STOP$ event. According to Definition 6.7 and Definition 6.17 P retains all posets of P_1 that contain the $STOP$ element. From $IH2_1$ we conclude that there exist a sub-set of P which has the same set of traces as $\delta_{q_0F_{N1}}$ and whose members contains the $STOP$ element.

We also know (from Definition 6.7 and Definition 6.17) that P is the result of the pairwise sequential composition of posets in P_1 that do not contain the $STOP$ element with all posets in P_2. From $IH3_1$ and $IH2_2$ we conclude that there exist a sub-set of P which has the same set of traces and where each poset contains $STOP$. Condition 3 is discharged in a similar way by concluding from $IH3_1$ and $IH3_2$ that there exist a sub-set of P which has the same set of traces as $\delta_{q_0 F_C}$ and whose members do not contain $STOP$.

■

Lemma 8.3:

PROOF $(P_1 \parallel P_2 \equiv_{CTR} M_1 \parallel M_2)$: Let $P = P_1 \parallel P_2$ and $M = (Q, \Sigma, \delta, q_0, F_C, F_N) = M_1 \parallel M_2$ with $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, F_{C1}, F_{N1})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, F_{C1}, F_{C2})$ we prove that $P \equiv_{CTR} M$. We start by proving that P and M are trace equivalent. From Proposition 6.2 and Proposition 7.1 we know that the set of all traces of P and M are sets of all interleavings of the traces of P_1 and P_2 and M_1 and M_2, respectively. From $IH1_1$ and $IH1_2$ we conclude that $Tr(P) = Tr(M)$.

We continue to show that conditions (2) and (3) are satisfied and start with the implication from left to right ($\Longrightarrow$). According to Definition 6.8 and Definition 6.17, the parallel composition of sets of posets is defined as the pairwise parallel composition of the respective posets of P_1 and P_2. Hence P can be decomposed into the following sub-sets $P = P_{NOSTOP} \cup P_{STOP.P_1} \cup P_{STOP.P_2} \cup P_{STOP.P_1P_2}$. P_{NOSTOP} is the sets of posets resulting from the pairwise parallel composition of posets taken from P_1 and P_2 that do not contain the $STOP$ element. $P_{STOP.P_1}$ and $P_{STOP.P_2}$ are the set of posets resulting from the composition of posets where either the first operand ($P_{STOP.P_1}$) or the second operand ($P_{STOP.P_2}$) contains the $STOP$ element. $P_{STOP.P_1P_2}$ results from the parallel composition of posets, which both contain $STOP$. Clearly, except for P_{NOSTOP} all posets in P contain the $STOP$ element.

In order to satisfy condition (2) we show that for each trace in $P_{STOP.P_1} \cup P_{STOP.P_2} \cup P_{STOP.P_1P_2}$ there exist a run in M to a state in F_N.

We start with $P_{STOP.P_1}$. The set of traces in $P_{STOP.P_1}$ is the set of all interleavings of traces of posets in P_1 that contain $STOP$ and traces of posets in P_2 that do not contain $STOP$.

According to Definition D.7, $F_N = \big((F_{C1} \cup F_{N1}) \times (F_{C2} \cup F_{N2})\big) - (F_{C1} \cup F_{C2})$. Clearly, we have $F_{N1} \times F_{C2} \subseteq F_N$. Similarly to the trace set of $P_{STOP.P_1}$, we know from Definition D.7 that the set of all runs in M to a state in $F_{N1} \times F_{C2}$ is the set if all interleavings of runs from the initial state of M_1 to a state F_{N1} and runs from the initial state in M_2 to a state in F_{C2}. This allows us to conclude from $IH2_1$ and $IH3_2$ that for each trace in $P_{STOP.P_1}$ there is a corresponding run in M from the initial state to a state in $F_{N1} \times F_{C2} \subseteq F_N$. The cases for $P_{STOP.P_2}$ and $P_{STOP.P_1P_2}$ are discharged accordingly from $IH3_1$ and $IH2_2$ or $IH2_1$ and $IH2_2$, respectively.

P_{NOSTOP} contains the posets resulting from the pairwise parallel composition of the posets in P_1 and P_2 that do not contain the $STOP$ event. Hence, in order to satisfy condition (3) we show that for each trace in P_{NOSTOP} there exist a run in M from the initial state to a state in F_C. According to Definition D.7 $F_C = F_{C1} \times F_{C2}$ and the set of all runs in M from the initial state to a state in F_C corresponds to the set of all interleavings of runs in M_1 from the initial state to a state in F_{C1} and runs in M_2 from the initial state to a state in F_{C2}. This allows us to conclude from $IH3_1$ and $IH3_2$ that for each trace in P_{NOSTOP} there is a corresponding run in M from the initial state to a state in F_C.

We complete the proof by proving that the implication from right to left ($\Leftarrow$) holds. Since, $F_C = F_{C1} \times F_{C2}$ and $F_N = (F_{C1} \times F_{N2}) \cup (F_{N1} \times F_{C2}) \cup (F_{N1} \times F_{N2})$ the set of traces of M can be split up as follows: $Tr(M) = (\delta_{q_0F_{C1}} ||| \delta_{q_0F_{C2}}) \cup (\delta_{q_0F_{C1}} ||| \delta_{q_0F_{N2}}) \cup (\delta_{q_0F_{N1}} ||| \delta_{q_0F_{C2}}) \cup (\delta_{q_0F_{N1}} ||| \delta_{q_0F_{N2}})$. $(\delta_{q_0F_{C1}} ||| \delta_{q_0F_{C2}})$ represents the set of interleaved runs of M_1 and M_2 from q_{0_1} to F_{C1} and q_{0_2} to F_{C2}, respectively. $(\delta_{q_0F_{C1}} ||| \delta_{q_0F_{N2}})$, $(\delta_{q_0F_{N1}} ||| \delta_{q_0F_{C2}})$ and $(\delta_{q_0F_{N1}} ||| \delta_{q_0F_{N2}})$ are the set of interleaved runs of M_1 and M_2 from q_{0_1} to F_{C1} and q_{0_2} to F_{N2}, *or* from q_{0_1} to F_{N1} and q_{0_2} to F_{C2},. *or* from q_{0_1} to F_{N1} and q_{0_2} to F_{N2}.

In order to satisfy condition (2) we show that for each trace set $(\delta_{q_0F_{C1}} ||| \delta_{q_0F_{N2}})$, $(\delta_{q_0F_{N1}} ||| \delta_{q_0F_{C2}})$, and $(\delta_{q_0F_{N1}} ||| \delta_{q_0F_{N2}})$ there exists a sub-set in P, in which each poset contains the $STOP$ element and which has the same set of traces. The case $(\delta_{q_0F_{C1}} ||| \delta_{q_0F_{N2}})$ can be discharged from $IH3_1$ and $IH2_2$ while taking into account the fact that the parallel composition of two posets will result in a poset containing $STOP$, if

at least one operand posets contains *STOP*. The cases for $(\delta_{q_0F_{N1}} ||| \delta_{q_0F_{C2}})$ and $(\delta_{q_0F_{N1}} ||| \delta_{q_0F_{N2}})$ are discharged in a similar manner by drawing conclusions from $IH2_1$, $IH3_2$ and $IH2_1$, $IH2_2$, respectively.

In order to satisfy condition (3) we show that there exists a sub-set in P, in which all posets do not contain the *STOP* element and which has the same set of traces as $(\delta_{q_0F_{C1}} ||| \delta_{q_0F_{C2}})$. According to Definition 6.8 and Definition 6.17 a poset in P does not contain the *STOP* element, if it is the result of the pairwise composition of posets from P_1 and P_2 that also do not contain the *STOP* element. Taking this into consideration we can conclude from $IH3_1$ and $IH3_2$ that there exist a sub-set of P which has the same set of traces as $(\delta_{q_0F_{C1}} ||| \delta_{q_0F_{C2}})$ and does not contain the *STOP* element. ■

Lemma 8.4:

PROOF $(P_1 \# P_2 \equiv_{CTR} M_1 \# M_2)$: Let $P = P_1 \# P_2$ and $M = (Q, \Sigma, \delta, q_0, F_C, F_N) = M_1 \# M_2$ with $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, F_{C1}, F_{N1})$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, F_{C1}, F_{C2})$ we prove that $P \equiv_{CTR} M$. We start by proving that P and M are trace equivalent. From Proposition 6.2 and Proposition 7.1 we know that $Tr(P) = Tr(P_1) \cup Tr(P_2)$ and that $Tr(M) = Tr(M_1) \cup Tr(M_2)$. From $IH1_1$ and $IH1_2$ we conclude that $Tr(P) = Tr(M)$.

We continue to show that conditions (2) and (3) are satisfied and start with the implication from left to right ($\Longrightarrow$). According to Definition 6.17 P can be decomposed into $P = P_{STOP.P_1} \cup P_{NOSTOP.P_1} \cup P_{STOP.P_2} \cup P_{NOSTOP.P_2}$, where $P_{STOP.P_1}$ and $P_{STOP.P_2}$ are sub-sets of P_1 and P_2 consisting of posets which contain the *STOP* event and $P_{NOSTOP.P_1}$ and $P_{NOSTOP.P_2}$ are sub-sets of P_1 and P_2 which only consist of posets which do not contain *STOP*. We show that for each trace of $P_{STOP.P_1}$ and $P_{STOP.P_2}$ there exist a run in M from q_0 to F_N and for each trace of $P_{NOSTOP.P_1}$ and $P_{NOSTOP.P_2}$ there exist a run from q_0 to F_C. The former case is discharged by $IH2_1$ and $IH2_2$ and the fact that M is constructed by merging the initial states of M_1 and M_2 with $F_N = F_{N1} \cup F_{N2}$. The latter case is discharged in a similar manner from $IH3_1$ and $IH3_2$ and the fact $F_C = F_{C1} \cup F_{C2}$.

We continue by discharging the implication from right to left ($\Longleftarrow$). Since $F_N = F_{N1} \cup F_{N2}$ and $F_C = F_{C1} \cup F_{C2}$ the set of all traces of M can be split up as follows: $Tr(M) =$

$\delta_{q_0F_{N1}} \cup \delta_{q_0F_{N2}} \cup \delta_{q_0F_{C1}} \cup \delta_{q_0F_{C2}}$, where $\delta_{q_0F_{N1}}$ and $\delta_{q_0F_{N2}}$ represent the sets of runs from q_0 to F_{N1} and q_0 to F_{N2}, respectively. Analogously $\delta_{q_0F_{C1}}$ and $\delta_{q_0F_{C2}}$ represent the sets of runs from q_0 to F_{C1} and q_0 to F_{C2}, respectively. From $IH2_1$ and $IH2_2$ and the fact that P is the union of the posets of P_1 and P_2 we conclude that there exist sub-sets of P which implement the same traces as $\delta_{q_0F_{N1}}$ and $\delta_{q_0F_{N2}}$and whose entailed posets contain the $STOP$ event. The case for $\delta_{q_0F_{C1}}$and $\delta_{q_0F_{C2}}$is discharged similarly by taking into account $IH3_1$ and $IH3_2$. ■

Lemma 8.5:

PROOF ($P_1^* \equiv_{CTR} M_1^*$): Let $P = P_1^*$ and $M = (Q, \Sigma, \delta, q_0, F_C, F_N) = M_1^*$ with $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, F_{C1}, F_{N1})$ we prove that $P \equiv_{CTR} M$. We start by proving that P and M are trace equivalent. Let $P_{1_{STOP}}$ be the posets of P_1 that contain $STOP$ and $P_{1_{NOSTOP}}$ the posets of P_1 that do not contain $STOP$, then $Tr_{P1^n{STOP}}$ is the set of all traces of $P_{1_{NOSTOP}} \cdot_1 P_{1_{NOSTOP}} \cdot_2 \cdots \cdot_{n-1} P_{1_{NOSTOP}} \cdot_n P_{1_{STOP}}$ and $Tr_{P1^n{NOSTOP}}$ is the set of all traces of $P_{1_{NOSTOP}} \cdot_1 P_{1_{NOSTOP}} \cdot_2 \cdots \cdot_n P_{1_{NOSTOP}}$. Let M_1F_N be the nFSM obtained from M by setting $F_C = \emptyset$ and M_1F_C be the nFSM obtained from M by setting $F_C = \emptyset$, then $Tr_{M1^n{F_{N1}}}$is the set of all traces of $M_1F_C \cdot_1 M_1F_C \cdot_2 \cdots \cdot_{n-1} M_1F_C \cdot_n M_1F_N$ and $Tr_{M1^n{F_{C1}}}$ is the set of all traces of $M_1F_C \cdot_1 M_1F_C \cdot_2 \cdots \cdot_n M_1F_C$. From Proposition 6.2 and Proposition 7.1 it follows that the set of all traces of P and M are decomposable as depicted in the table below:

Table E.2 Breakdown of Sets of Traces of P and M

$Tr(P) = \{(\ )\} \cup$	$Tr(M) = \{(\ )\} \cup$
$Tr_{P1^1{STOP}} \cup Tr_{P1^1{NOSTOP}} \cup$	$Tr_{M1^1{F_{N1}}} \cup Tr_{M1^1{F_{C1}}} \cup$
$Tr_{P1^2{STOP}} \cup Tr_{P1^2{NOSTOP}} \cup$	$Tr_{M1^2{F_{N1}}} \cup Tr_{M1^2{F_{C1}}} \cup$
$Tr_{P1^3{STOP}} \cup Tr_{P1^3{NOSTOP}} \cup$	$Tr_{M1^3{F_{N1}}} \cup Tr_{M1^3{F_{C1}}} \cup$
...	...
∞	∞

Using the breakdown of the trace set we prove by induction that $Tr(P) = Tr(M)$.

BASE ($n = 1$)

From $IH1_1$, $IH2_1$ and $IH3_1$ it follows that $Tr_{P1^1STOP} = Tr_{M1^1F_{N1}}$ and $Tr_{P1^1NOSTOP} = Tr_{M1^1F_{C1}}$.

INDUCTION ($n \rightarrow n + 1$)

We assume that $Tr_{P1^nSTOP} = Tr_{M1^nF_{N1}}$ and $Tr_{P1^nNOSTOP} = Tr_{M1^nF_{C1}}$ and show that $Tr_{P1^{n+1}STOP} = Tr_{M1^{n+1}F_{N1}}$ and $Tr_{P1^{n+1}NOSTOP} = Tr_{M1^{n+1}F_{C1}}$.

From Proposition 6.2 we know that $Tr_{P1^{n+1}STOP}$ and $Tr_{P1^{n+1}NOSTOP}$ are constructed by stepwise concatenating $Tr_{P1^1NOSTOP}$ with Tr_{P1^nSTOP} and $Tr_{P1^nNOSTOP}$, respectively. Similarly, it follows from Proposition 7.1 that $Tr_{M1^{n+1}F_{N1}}$ and $Tr_{M1^{n+1}F_{C1}}$ are the results of stepwise concatenating $Tr_{M1^1F_{C1}}$ with $Tr_{M1^nF_{N1}}$ and $Tr_{M1^nF_{C1}}$, respectively. Since $Tr_{P1^1NOSTOP} = Tr_{M1^1F_{C1}}$(base case), $Tr_{P1^1STOP} = Tr_{M1^1F_{N1}}$(base case) and $Tr_{P1^nNOSTOP} = Tr_{M1^nF_{C1}}$(induction hypothesis) it follows that $Tr_{P1^nNOSTOP} \cdot Tr_{P1^1STOP} = Tr_{M1^nF_{C1}} \cdot Tr_{M1^1F_{N1}}$ and $Tr_{P1^nNOSTOP} \cdot Tr_{P1^1NOSTOP} = Tr_{M1^nF_{C1}} \cdot Tr_{M1^1F_{C1}}$ and we can conclude that P and M are trace equivalent. ∎

We continue to show that conditions (2) and (3) are satisfied and start with the implication from left to right ($\Rightarrow$). According to Definition 6.18 the result of the iterative composition of sets of posets can be decomposed into the following sub-sets of P.

$$\{\emptyset_{poset}\} \cup$$
$$P_{1STOP} \cup P_{1NOSTOP} \cup$$
$$P_{1NOSTOP} \cdot P_{1STOP} \cup P_{1NOSTOP} \cdot P_{1NOSTOP} \cup$$
$$P_{1NOSTOP} \cdot P_{1NOSTOP} \cdot P_{1STOP} \cup P_{1NOSTOP} \cdot P_{1NOSTOP} \cdot P_{1NOSTOP} \cup$$
$$\ldots\ldots$$
$$(P_{1NOSTOP})^{n-1} \cdot P_{1STOP} \cup (P_{1NOSTOP})^n \cup$$
$$\ldots\ldots$$
$$\infty$$

We show by induction that for each trace of $(P_{1NOSTOP})^{n-1} \cdot P_{1STOP}$, with $n \geq 1$ there exist a run in M from q_0 to F_N and for each trace in $(P_{1NOSTOP})^n$, with $n \geq 1$ there exist a run from q_0 to F_C

BASE ($n = 1$)

By definition, $(P_{1NOSTOP})^0 \cdot P_{1STOP} = P_{1STOP}$ and $(P_{1NOSTOP})^1 = P_{1NOSTOP}$. Since M^* does not remove any transitions from M while letting F_N unchanged and adding q_0 to F_C, but not removing any states from F_C, it follows from $IH2_1$ that for each trace in P_{1STOP} there exists a run in M from q_0 to F_N. It follows from $IH3_1$ that for each trace in $P_{1NOSTOP}$ there exists a run in M from q_0 to F_C.

INDUCTION ($n \rightarrow n + 1$)

We assume that for each trace in $(P_{1NOSTOP})^{n-1} \cdot P_{1STOP}$ there exists a run in M from q_0 to a state in F_N and that for each trace in $(P_{1NOSTOP})^n$ there exists a run in M from q_0 to a state in F_C. To satisfy condition (2) we show that for each trace in $(P_{1NOSTOP})^n \cdot P_{1STOP}$ there exists a run in M from q_0 to a state in F_N. To satisfy condition (3) we show that for each trace in $(P_{1NOSTOP})^{n+1}$ there exists a run in M from q_0 to a state in F_C. From the induction hypothesis we know that for each trace in $(P_{1NOSTOP})^n$ there exists a run from q_0 to F_C. We also know that for each trace in P_{1STOP} and $P_{1NOSTOP}$ there exists a run from q_0 to F_N and from q_0 to F_N accordingly. Since according to Definition 7.12 all states in F_C adopt the behavior of the initial state of M_1 we conclude that for each trace in $(P_{1NOSTOP})^n \cdot P_{1STOP}$ and $(P_{1NOSTOP})^n \cdot (P_{1NOSTOP}) = (P_{1NOSTOP})^{n+1}$ there exists a run from q_0 to F_N and from q_0 to F_C respectively. ■

We continue the proof by discharging the implication from right to left ($\Leftarrow$). According to Proposition 7.1 the runs of M can be broken down as indicated on the right hand side of Table E.2. The run $\langle\ \rangle$ results in the initial state q_0 which (according to Definition 7.12) is part of the set F_C. Clearly, the corresponding poset $\{\emptyset_{poset}\}$ is part of P (according to Definition 6.18). In what follows, to satisfy condition (2), we show by induction that for each trace in $Tr_{M1^nF_{N1}}$ there exists a poset in P which has a corresponding trace and contains the $STOP$ element. To satisfy condition (3) we show that for each trace in $Tr_{M1^nF_{C1}}$ there exist a poset in P which implements the same trace and does not contain $STOP$.

BASE ($n = 1$): The base case can be easily discharged from the $IH1_2$ and $IH2_3$ and the fact that $P_1 \subset P$ (Definition 6.18).

INDUCTION ($n \rightarrow n+1$)

We assume that for each trace in $Tr_{M1^n F_{C1}}$ there exists a poset in P which implements the same trace and does not contain the $STOP$ element. We prove that the above holds for $Tr_{M1^{n+1} F_{C1}}$ and that for each trace in $Tr_{M1^{n+1} F_{N1}}$ there exists a poset which has the same trace and does contain $STOP$.

From the inductive proof of trace equivalence we know that there must exist a sub-set of P, which we shall call P^n_{NOSTOP}, such that $Tr(P^n_{NOSTOP}) = Tr(Tr_{M1^n F_{C1}})$ and each entailed poset does not contain $STOP$. Clearly $P^n_{NOSTOP} \cdot P_1 \subset P$. Since we can rewrite P_1 as $P_1 = P_{1STOP} \cup P_{1NOSTOP}$, we have $P^n_{NOSTOP} = P^n_{NOSTOP} \cdot P_{1STOP} \cup P^n_{NOSTOP} \cdot P_{1STOP}$. It follows from Definition 6.7 and Proposition 6.2 that $P^n_{NOSTOP} \cdot P_{1STOP}$ has the same set of traces as $Tr_{M1^{n+1} F_{N1}}$ and each entailed poset contains $STOP$. It also follows that $P^n_{NOSTOP} \cdot P_{1NOSTOP}$ has the same set of traces as $Tr_{M1^{n+1} F_{C1}}$ and each entailed poset does not contain $STOP$. ∎

Lemma 8.6:

PROOF

Sub-goal: $(close(P_1) \equiv_{CTR} close(M_1))$: Let $P = close(P_1)$ and $M = close(M_1)$ with $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, F_{C1}, F_{N1})$ we prove that $P \equiv_{CTR} M$. According to Proposition 6.2 and Proposition 7.1, we have $Tr(P_1) = Tr(P)$ and $Tr(M_1) = Tr(M)$. Hence, under the assumption $IH1_1$ it follows that $Tr(P) = Tr(M)$ and condition (1) is clearly satisfied.

We continue to show that conditions (2) and (3) are satisfied. According to Definition 6.11 and Definition 6.19 the close operation on sets of posets adds the $STOP$ event to each entailed poset. Hence, every trace of P will originate from a poset containing the $STOP$ event. Similarly the close operation on nFSMs removes each state from F_C and adds it to F_N. Hence, each run of the nFSM will result into a state in F_N. From $IH1_1$ it follows that $Tr(P) =_{CTR} Tr(M)$. ∎

Sub-goal: $(open(P_1) \equiv_{CTR} open(M_1))$: Let $P = open(P_1)$ and $M = (Q, \Sigma, \delta, q_0, F_C, F_N) = close(M_1)$ with $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, F_{C1}, F_{N1})$ we prove that

$P \equiv_{CTR} M$. This case is proven in the same manner as the previous case, acknowledging the fact that open is dual to close on sets of posets as well nFSMs. ■

Lemma 8.7:

PROOF: According to step 1 of the *LTS_to_SPO* algorithm the initial interpretation of *SPO* is defined as follows:

$$SPO(q_k, q_n) = \begin{cases} \{X, \{(X, \{(l, l) \mid l \in X\})\}\} & \exists X \in \mathbb{P}(\Sigma), \exists Q_n \in \mathbb{P}(Q). q_n \in Q_n \wedge (q_k, X, Q_n) \in \delta \\ \{\emptyset_{poset}\} & otherwise \end{cases}$$

We start by showing that:

(1) $\mathcal{P}(X) = Tr(\{(X, \{(l, l) \mid l \in X\})\})$

Clearly, if X is a singleton set {e}, we have $\mathcal{P}(\{e\}) = \{\langle e \rangle\} = Tr(\{e_{poset}\})$. If X contains multiple elements then $\mathcal{P}(X)$ returns the set of all possible event sequences over X. $SPO(X)$ returns a set of posets $\{P\}$, where $P = (X, \{(l, l) \mid l \in X\})$. Since all events of P are causally unrelated, it follows from Definition 6.13 that the set of all traces of P is the set of all event sequences over X. Hence $\mathcal{P}(X) = Tr(\{(X, \{(l, l) \mid l \in X\})\})$. Assuming the initial interpretation of *SPO* (as presented above), we derive the following statement.

(2) $\forall q_k, q_n \in Q, \forall X \in \mathbb{P}(\Sigma), \exists Q_n \in \mathbb{P}(Q).$

$$[((q_k, X, Q_n) \in \delta) \Rightarrow (\mathcal{P}(X) = Tr(SPO(q_k, q_n)))]$$

According to Proposition 6.2 the resulting trace set of the sequential composition of P_1 and P_2 equals to $Tr(P \cdot R) = \{x \hat{} y \mid x \in Tr(p) \wedge y \in Tr(R) \wedge p = (E_p, \leq_p) \in P \wedge STOP \notin E_p\} \cup \{x \mid x \in Tr(p) \wedge p = (E_p, \leq_p) \wedge STOP \in E_p\}$. If P_1, however does not consists of posets containing *STOP*[13], the set of all traces equals to the set resulting from the pairwise sequential composition of the traces of P_1 and P_2 ($\{x \hat{} y \mid x \in Tr(p) \wedge y \in Tr(R)\}$) and we conclude that $Tr(P_1) \cdot Tr(P_2) = Tr(P_1 \cdot P_2)$. Clearly this observation can be generalized to the sequential composition involving more than two operands; i.e., under the assumption that $P_1, P_2, \ldots, P_n$ do not contain posets containing *STOP* it follows that:

[13] This is not possible according to the *LTS_to_SPO* algorithm

(3) $Tr(P_1) \cdot Tr(P_2) \cdot \ldots \cdot Tr(P_n) = Tr(P_1 \cdot P_2 \cdot \ldots \cdot P_n)$

Using (3), the definition of $Tr(U_{SPOSET-INIT})$ can be rewritten as:

$$Tr(U_{SPOSET-INIT}) = \bigcup \{Tr(SPO(q_0, q_1)) \cdot Tr(SPO(q_1, q_2)) \cdot \ldots \cdot Tr(SPO(q_{n-1}, q_n)) \mid \delta(q_{i-1}, X_i) = q_i \text{ for } i \in \{1, .., n\} \text{ and } q_n \in F\}$$

Using (1) and (2) we further rewrite $Tr(U_{SPOSET-INIT})$ to:

$$Tr(U_{SPOSET-INIT}) = \bigcup \{\mathcal{P}(X_1) \cdot \mathcal{P}(X_2) \cdot \ldots \cdot \mathcal{P}(X_n) \mid \delta(q_{i-1}, X_i) = q_i \text{ for } i \in \{1, .., n\} \text{ and } q_n \in F\}$$

Since the transitions of $U_{SPOSET-INIT}$ are clearly identical to U we conclude that $Tr(U) = Tr(U_{SPOSET-INIT})$. ◙

Lemma 8.9:

Steps 9 and 10 of the *LTS_to_SPO* algorithm postulate the derivation of the following set of posets from a minimized generalized UC-LTS: $P_{result} = P_{00}^{*} \cdot (P_{01} \# P_{02} \# \ldots \# P_{0n})$. Clearly, the set of traces of $U_{SPOSET-MIN}$ can be broken down as follows:

$$\begin{aligned} Tr(U_{SPOSET-MIN}) = \quad & Tr(P_{01}) \cup Tr(P_{02}) \cup \ldots \cup Tr(P_{0n}) \cup \\ & Tr(P_{00} \cdot P_{01}) \cup Tr(P_{00} \cdot P_{02}) \cup \ldots \cup Tr(P_{00} \cdot P_{0n}) \cup \\ & Tr(P_{00} \cdot P_{00} \cdot P_{01}) \cup Tr(P_{00} \cdot P_{00} \cdot P_{02}) \cup \ldots \cup Tr(P_{00} \cdot P_{00} \cdot P_{0n}) \cup \\ & \ldots \\ & Tr(P_{00} \cdot P_{00} \cdot \ldots \cdot P_{01}) \cup Tr(P_{00} \cdot P_{00} \cdot \ldots \cdot P_{02}) \cup \ldots \cup Tr(P_{00} \cdot P_{00} \cdot \ldots \cdot P_{0n}) \end{aligned}$$

Since none of the involved posets can contain the *STOP* event, it is valid to factor out the sequential compositions of P_{00}'s and to simplify the equation using the iteration operator as follows: $Tr(U_{SPOSET-MIN}) = Tr(P_{00}^{*} \cdot P_{01}) \cup Tr(P_{00}^{*} \cdot P_{02}) \cup \ldots \cup Tr(P_{00}^{*} \cdot P_{0n})$. From Proposition 6.2 we know that the set of traces of an alternative composition of sets of posets equals to the union of the set of traces of the involved operands. Hence,

$Tr(U_{SPOSET-MIN}) = Tr\big((P_{00}^{*} \cdot P_{01}) \# (P_{00}^{*} \cdot P_{02}) \# \ldots \# (P_{00}^{*} \cdot P_{0n})\big)$. We can further simplify the equation by factoring out the P_{00}^{*}'s and obtain $Tr(U_{SPOSET-MIN}) = Tr\big(P_{00}^{*} \cdot (P_{01} \# P_{02} \# P_{03})\big)$. Clearly the set of traces of P_{result} equals to the set of traces of $U_{SPOSET-MIN}$. ■

Lemma 8.11:

PROOF: M is obtained by replacing all *singleton set* transitions of U_{flat} by *event name* transitions. According to Definition 7.4, the set of traces of M is defined as $Tr(M) = \{w \mid \delta^{*}(q_0, w) \cap (F_C \cup F_N) \neq \emptyset\}$. From the fact that the initial and final states of U_{flat} and M are the same (Definition 7.20), it follows that $w = \langle e_1, e_2, \ldots e_n \rangle$ is a run of M, if and only if $\langle \{e_1\}, \{e_2\}, \ldots, \{e_n\} \rangle$ is a run of U_{flat}. Clearly, $\mathcal{P}(\{e_1\}) \cdot \mathcal{P}(\{e_2\}) \cdot \ldots \cdot \mathcal{P}(\{e_n\}) = \{\{\langle e_1 \rangle\} \cdot \{\langle e_2 \rangle\} \cdot \ldots \cdot \{\langle e_n \rangle\}\} = \{\langle e_1, e_2, \ldots e_n \rangle\}$ and we conclude that $Tr(U_{flat}) = Tr(M)$. ■

Appendix F Scenario Specification and Refinement Language

The syntax of the Scenario Specification and Refinement Language (SSRL) is defined by the following grammar in Extended Backus-Naur Form (EBNF)

```
(* Grammar of Scenario Specification and Refinement Language *)
(* V.1.0 *)

****Start SSRL****

<USL> ::= <DECLSECTION> <USECASEDEFSECTION> <TASKDEFSECTION>
<REFMAPSECTION> <ASSERTSECTION>

****Event Definitions****

<DECLSECTION> ::= "EVENTS:" (<DEF>)+
<DEF> ::= <EVENTNAME> ":" <TYPE> ";"
<EVENTNAME> ::= <ID>
<TYPE> ::= "INTERACTION" | "SYSTEM" | "INTERNAL"

****UC-JIS Definitions****

<USECASEDEFSECTION> ::= "USECASEDEFS:" (<UCDEF>)+

<UCDEF> ::= <UCNAME> "=" <UCLTS> ";"
<UCLTS> ::= "{" <TRANSITIONS> "}", <FINALSTATES>
<TRANSITIONS> ::= <TRANSITION> ("," <TRANSITION>)*
<TRANSITION> ::= "(" <STATE> "," "{" <EVENTS> "}" "," <STATE> ")"
<FINALSTATE> ::= "{" <STATES> "}"
<STATES> ::= <STATE> ("," <STATE>)*
<STATE> ::= <ID>
<EVENTS> ::= <EVENT> ("," <EVENT>)*
<EVENT> ::= <ID>
<UCNAME> ::= <ID>

****GTM Definitions****

<TASKDEFSECTION> ::= "TASKDEFS:" (<GTE>)+
<GTE> ::= <TASKNAME> "=" <GTEEXPR> ";"
<GTEEXPR> ::= <GTEEXPR> " >>" <ENABLED>
<ENABLED> ::= <ENABLED> "-||" <PARALLEL>
<PARALLEL> ::= <PARALLEL>"-" []" <CHOICE>
<CHOICE> ::= <TASKID>  | <ITERATION> | <OPTIONAL> | <STOP> | <RESUME> |
             "("<GTEEXPR>")"

<ITERATION> ::= "("<GTEEXPR>")*"
<OPTIONAL> ::= "["< GTEEXPR >"]"
<STOP> ::= "STOP" "("<GTEEXPR>")"
<RESUME ::= "RESUME" "("<GTEEXPR>")"
<TASKID> ::= <ID>
<TASKNAME> ::= <ID>

****Refinement Mappings****

<REFMAPSECTION> ::= "REFMAPS:" (<REFMAP>)+
<REFMAP> ::= <REFMAPID> "=" "{" <MAPS> "{" ";"
<MAPS> ::= <MAP> ("," <TRANSITION>)*
<MAP> ::= "(" "{" <EVENTS> "}" "," <EVENT> ")"
<REFMAPID> ::= <ID>

****Assertions****

<ASSERTSECTION> ::= "ASSERT:" (<ASSERTION>)+
<ASSERTION> ::= <SPECNAME> <RELATION> <SPECNAME> "WITH" <REFMAPID>";"
<SPECNAME> ::= <ID>
<RELATION> ::= "EQU" | "DET"

****Auxiliary Definitions****

<ID> ::= <LETTER> (<LETTER> | <DIGIT>)*
<LETTER> ::= [" ","a"-"z","A-"Z"]
<DIGIT> ::= ["0"-"9"]
```

Appendix G Case Study: Invoice Management System

In this chapter we specify the functional (use case model) and user interface (UI) requirements (task model) of the "Invoice Management System" (IMS). Using the formal semantic framework presented in this thesis, the use case model and task model are transformed into the semantic domains. It is proven that the task model is a valid refinement of the use case model.

Appendix G.1 IMS DSRG-style Use Case Model

An overview (in form of a use case diagram) of the "IMS" use case model is given in Figure G.1.

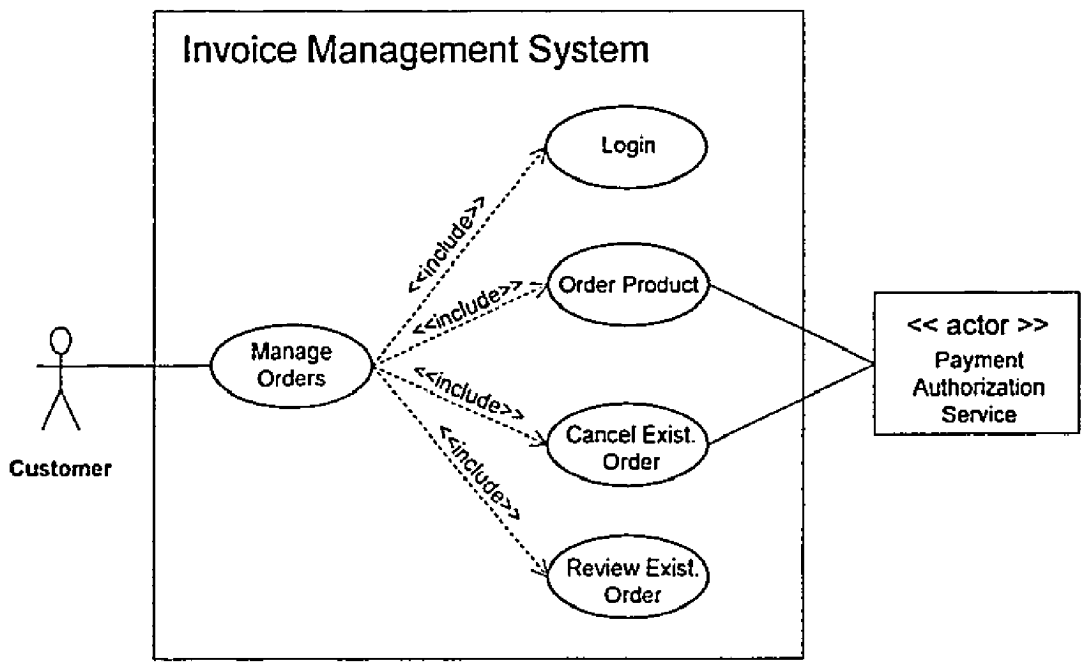


Figure G.1. "IMS" Use Case Diagram

In what follows, we provide the full details of all involved use cases. For the sake of readability the use cases are given both, prose and abstract syntax (DSRG-style use case model).

Appendix G.1.1 IMS Use Case Model in Textual Format

Use case: Manage Orders

Properties

Goal: Primary Actor manages his/her orders
Primary Actor: Customer
Goal Level: Summary
Precondition: None

Main Success Scenario

1. Primary Actor logs into the Invoice Management System (Extinclude)
2. Primary Actor *chooses one of the following* (Choice)
   - Order a Product (Include)
   - Review an Existing Order (Include)
   - Cancel an Existing Order (Include)
3. *Use case resumes at step 2.* (Goto)

Extensions

1a. Login terminated unsuccessfully:

1a1. *Use case ends unsuccessfully* (Failure)

2a. Primary Actor decides to quit the Invoice Management System:

2a1. Primary Actor indicates to quit Invoice Management System. (Atom. interaction, 'inQU')

2a2. *Use case ends successfully* (Success)

Use Case: Login

Properties

Goal: Primary Actor logs into the system
Primary Actor: Customer
Goal Level: Sub-function
Precondition: None

Main Success Scenario

1. Primary Actor indicates his/her desire to login into the system. (Atom. interaction, 'inLO')
2. System prompts Primary Actor for login coordinates. (Atom. application, 'prLO')
3. *Primary Actor performs the following steps in any order:* (Concurrent)
   - Primary Actor provides username. (Primitive, 'prUN')
   - Primary Actor provides password. (Primitive, 'prPW')
4. Primary Actor submits the input. (Atom. interaction, 'smIN')
5. System authenticates the Primary Actor. (Atom. internal, 'vaIN')
6. System informs Primary Actor that login was successful. (Atom. application, 'inLS')
7. *Use case ends successfully.* (Success)

Extensions

3a. Primary Actor decides to cancel the use case:

3a1. Primary Actor indicates to cancel the use case. (Atom. interaction, 'inCL')

3a2. *Use case ends unsuccessfully.* (Failure)

5a: System fails to authenticate Primary Actor:

5a1. System informs Primary Actor that login was denied. (Atom. application, 'inLF')
5a2. *Use case resumes at step 2.* (Goto)

Use case: Order Product

Properties

Goal: Primary Actor places an order for a specific product.
Primary Actor: Customer
Goal Level: User-goal
Precondition: Primary Actor is logged into the system.

Main Success Scenario

1. Primary Actor indicates that he/she wishes to Order a Product. (Atom. interaction. "inOP")
2. Primary Actor specifies the desired product category. (Atom. interaction. "spCA")
3. System displays search results that match the Primary Actor's supplied criteria. (Atom. application. "diSR")
4. Primary Actor selects a product and identifies the desired quantity. (Atom. interaction. "slPQ")
5. System validates that the product is available in the requested quantity. (Atom. internal. "vaPQ")
6. System displays the purchase summary. (Atom. application. "diPS")
7. Primary Actor *chooses one of the following* (Choice)

   7A.1. Primary Actor elects to pay by credit card and submits account information. (Atom. interaction. "paCC")

   OR

   7B.1 Primary Actor elects to pay by debit card and submits account information. (Atom. interaction. "paDB")

   7B.2 The Primary Actor provides his/her PIN number. (Atom. interaction. "prPI")
8. System interacts with the Payment authorization system to carry out the payment. (Atom. internal. "vaPA")
9. System informs Primary Actor that order is confirmed. (Atom. application. "inCO")
10. *Use case ends successfully.* (Success)

Extensions

4a. Primary Actor is not satisfied with the search results:

4a1. Primary Actor indicates to do another product search. (Atom. interaction. "inPS")

4a2. *Use case resumes at step 2.* (Goto)

4b. Primary Actor decides to cancel the use case:

4b1. Primary Actor indicates to cancel the use case. (Atom. interaction. "inCA")

4b2. *Use case ends unsuccessfully.* (Failure)

5a. The desired product is not available in sufficient quantities:

5a1. System informs Primary Actor that product unavailable in desired quantity. (Atom. application."inIQ")

5a2. *Use case ends unsuccessfully.* (Failure)

7a. Primary actor decides to cancel the use case:

7a1. Primary Actor indicates to cancel the use case. (Atom. interaction. "inCA")

7a2. *Use case ends unsuccessfully.* (Failure)

8a. The payment was not authorized:

8a1. System informs Primary Actor that payment was not authorized. (Atom. application. "inPF")

8a2. *Use case resumes at step 7.* (Goto)

Use case: Review an Existing Order

Properties

Goal: Primary Actor reviews an existing order.
Primary Actor: Customer
Goal Level: User-goal
Precondition: Primary Actor is logged into the system.

Main Success Scenario

1. Primary Actor indicates that he/she wishes to review an existing order. (Atom. interaction. "inRO")
2. System prompts Primary Actor to provide the confirmation number. (Atom. application. "prCN")

3. Primary Actor provides confirmation number. (Atom. interaction. "prCN")
4. System retrieves order with given confirmation number. (Atom. internal. "riOR")
5. System displays purchase summary of the order. (Atom. application. "diSU")
6. *Use case ends successfully.* (Success)

Extensions

3a. Primary Actor decides to cancel the use case:

3a1. Primary Actor indicates to cancel the use case. (Atom. interaction. "caRO")
3a2. *Use case ends unsuccessfully.* (Failure)

4a. The provided confirmation number is invalid:

4a1. System informs Primary Actor that confirmation number is invalid. (Atom. application. "diNI")
4a2. *Use case ends unsuccessfully.* (Failure)

Use case: Cancel an Existing Order

Properties

Goal: Primary Actor cancels an existing order.
Primary Actor: Customer
Goal Level: User goal
Precondition: Primary Actor is logged into the system.

Main Success Scenario

1. Primary Actor indicates that he/she wishes to cancel an existing order. (Atom. interaction. "trCO")
2. System prompts Primary Actor to provide the confirmation number. (Atom. application. "pmCN")
3. Primary Actor provides confirmation number. (Atom. interaction. "prCN")
4. System retrieves order with given confirmation number. (Atom. internal. "riOR")
5. System prompts for confirmation. (Atom. application. "pmCF")
6. Primary Actor confirms cancelation of the order. (Atom. interaction. "cfCO")
7. System informs Primary Actor that the order has been canceled. (Atom. application. "inOC")
8. *Use case ends successfully.* (Failure)

Extensions

3a. Primary Actor decides to cancel the use case:

3a1. Primary Actor indicates to cancel the use case. (Atom. interaction. "caRO")
3a2. *Use case ends unsuccessfully.* (Failure)

4a. The provided confirmation number is invalid:

4a1. System informs Primary Actor that confirmation number is invalid. (Atom. application. "diNI")
4a2. *Use case ends unsuccessfully.* (Failure)

4b. The order cannot be cancelled (see Business Rule 1):

4b1. System informs Primary Actor that order cannot be cancelled. (Atom. application. "diCC")
4b2. Use case ends unsuccessfully. (Failure)

6a. Primary Actor decides to *not* cancel the order:

6a1. Primary Actor indicates to deny the cancelation of the order. (Atom. interaction. "caCA")
6a2. *Use case ends unsuccessfully.* (Failure)

Business Rules

Rule 1: An order can only be canceled, if it has not been shipped, yet.

Appendix G.1.2 IMS Use Case Model in Abstract Syntax

Formally we define the "IMS" use case model as follows:

$D_{IMS} = (n_0, U)$ where,

n_0 = "Manage Orders"

$$U = \left\{ \begin{array}{c} \text{"Manage Orders"} := \text{ManageOrderUC}, \\ \text{"Login"} := \text{LoginUC}, \\ \text{"Order Product"} := \text{OrderProductUC}, \\ \text{"Review Order"} := \text{ReviewOrderUC}, \\ \text{"Cancel Order"} := \text{CancelOrderUC} \end{array} \right\}, \text{where the use cases}$$

ManageOrderUC, LoginUC, OrderProductUC, ReviewOrderUC and CancelOrderUC are defined in the following.

```
ManageOrdersUC :: UseCase
"ManageOrdersUC == (|
    Name = ''Manage Orders'',
    Properties = {|
      Goal = ''Primary Actor manages his/her orders'',
      PrimaryActor = ''Customer'',
      GoalLevel = SUMMARY,
      Precondition = ''None''  |),
    MainSuccessScenario = [
       ExtInclude ''MO.s1'' ''Login'' ''MO.e1'',
       Choice ''MO.s2''
         [
           [Include ''MO.s2A1'' ''Order Product''],
           [Include ''MO.s2A2'' ''Review Order''],
           [Include ''MO.s2A3'' ''Cancel Order'']
         ] {''MO.e2''},
       Goto ''MO.s3'' ''MO.s2''
    ],
    Extensions = {
      ('Extension 1e')
     (|
      ID = ''MO.e1'',
      Condition = '''',
      ExtensionScenario =
       [Failure ''MO.s1a1'']
     |),
      ('Extension 2e')
     (|
      ID = ''MO.e2'',
      Condition = ''Primary Actor decides to quit the IMS'',
      ExtensionScenario =
        [
          Atom ''MO.s2a1'' INTERACTION ''inQU'' {},
          Success ''MO.s2a2''
        ]
     |)}|)"
```

Figure G.2. Abstract Syntax of "Manage Orders" Use Case

```
LoginUC :: UseCase
 "LoginUC == (|
   Name = ''Login'',
   Properties = (|
    Goal = ''Primary Actor logs into the system'',
    PrimaryActor = ''Customer'',
    GoalLevel = SUBFUNCTION,
    Precondition = ''None''
   |),
   MainSuccessScenario = [
      Atom ''LO.s1'' INTERACTION ''inLO'' {},
      Atom ''LO.s2'' APPLICATION ''prCO'' {},
      Concurrent ''LO.s3'' {''prUN'', ''prPW''} {''LO.e1''},
      Atom ''LO.s4'' INTERACTION ''smIN'' {},
      Atom ''LO.s5'' INTERNAL ''vaIN'' {''LO.e2''},
      Atom ''LO.s6'' APPLICATION ''inLS'' {},
      Success ''LO.s7''
   ],
   Extensions = {
     (''Extension 3a'')
    (|
     ID = ''LO.e1'',
     Condition = ''Primary Actor decides to cancel the use case'',
     ExtensionScenario =
       [
        Atom ''LO.s3a1'' INTERACTION ''inCL'' {},
        Failure ''LO.s3a2''
       ]
    |),
     (''Extension 5a'')
    (|
     ID = ''LO.e2'',
     Condition = ''System fails to authenticate Primary Actor'',
     ExtensionScenario =
       [
        Atom ''LO.s5a1'' APPLICATION ''inLF'' {},
        Goto ''LO.s5a2'' ''LO.s2''
       ]
    |)}|)"
```

Figure G.3. Abstract Syntax of "Login" Use Case

```
constdefs
OrderProductUC :: UseCase
 "OrderProductUC == (|
    Name = ''Order Product'',
    Properties = (|
     Goal = ''Primary actor places an order for a specific product'',
     PrimaryActor = ''Customer'',
     GoalLevel = USERGOAL,
     Precondition = ''Primary Actor is logged into the system.''
    |),
    MainSuccessScenario = [
       Atom ''OP.s1'' INTERACTION ''trOP'' {},
       Atom ''OP.s2'' INTERACTION ''spCA'' {},
       Atom ''OP.s3'' APPLICATION ''diSR'' {},
       Atom ''OP.s4'' INTERACTION ''slPQ'' {''OP.e1'', ''OP.e2''},
       Atom ''OP.s5'' INTERNAL    ''vaPQ'' {''OP.e3''},
       Atom ''OP.s6'' APPLICATION ''diPS'' {},
       Choice ''OP.s7'' |
        [Atom ''OP.s7A1'' INTERACTION ''paCC'' {} ],
        [
         Atom ''OP.s7B1'' INTERACTION ''paDB'' {},
         Atom ''OP.s7B2'' INTERACTION ''prPI'' {}
        ] ] {''OP.e4''},
       Atom ''OP.s8'' INTERNAL    ''vaPA'' {''OP.e5''},
       Atom ''OP.s9'' APPLICATION ''inCO'' {},
       Success ''OP.s10''
    ],
    Extensions = (
      (* Extension 4a' *)
     (|
      ID = ''OP.e1'',
      Condition = ''Primary Actor is not satisfied with search results'',
      ExtensionScenario =
       [ Atom ''OP.s4a1'' INTERACTION ''inPS'' {},
         Goto ''OP.s4a2'' ''OP.s2'']
     |),
      (* Extension 4b' *)
     (|
      ID = ''OP.e2'',
      Condition = ''Primary Actor decides to cancel the use case'',
      ExtensionScenario =
       [ Atom ''OP.s4b1'' INTERACTION ''inCA'' {},
         Failure ''OP.s4b2'']
     |),
      (* Extension 5a' *)
     (|
      ID = ''OP.e3'',
      Condition = ''The product is unavailable in sufficient quantities'',
      ExtensionScenario =
       [ Atom ''OP.s5a1'' APPLICATION ''inIQ'' {},
         Failure ''OP.s5a2'' ]
     |),
      (* Extension 7a' *)
     (|
      ID = ''OP.e4'',
      Condition = ''Primary Actor decides to cancel the use case'',
      ExtensionScenario =
       [ Atom ''OP.s7a1'' INTERACTION ''inCA'' {},
         Failure ''OP.s7a2'' ]
     |),
      (* Extension 8a' *)
     (|
      ID = ''OP.e5'',
      Condition = ''The payment was not authorized'',
      ExtensionScenario =
       [ Atom ''OP.s8a1'' APPLICATION ''inPF'' {},
         Goto ''OP.s8a2'' ''OP.s7'' ]
     |)
    )
|)"
```

Figure G.4. Abstract Syntax of "Order Product" Use Case

```
constdefs
CancelOrderUC :: UseCase
 "CancelOrderUC == (|
    Name = ''Cancel Order'',
    Properties = (|
      Goal = ''Primary Actor cancels an existing order. '',
      PrimaryActor = ''Customer'',
      GoalLevel = USERGOAL,
      Precondition = ''Primary Actor is logged into the system.''
    |),
    MainSuccessScenario = [
       Atom ''CO.s1'' INTERACTION ''trCO'' {},
       Atom ''CO.s2'' APPLICATION ''pmCN'' {},
       Atom ''CO.s3'' INTERACTION ''prCN'' {''CO.e1''},
       Atom ''CO.s4'' INTERNAL ''riOR'' {''CO.e2'', ''CO.e3''},
       Atom ''CO.s5'' APPLICATION ''pmCF'' {},
       Atom ''CO.s6'' INTERACTION ''cfCO'' {''CO.e4''},
       Atom ''CO.s7'' APPLICATION ''inOC'' {},
       Success ''CO.s8''
    ],
    Extensions = {
      (''Extension 3a'')
     (| ID = ''CO.e1'',
        Condition = ''Primary Actor decides to cancel the use case'',
        ExtensionScenario =
          [
           Atom ''CO.s3a1'' INTERACTION ''caCO'' {},
           Failure ''CO.s3a2''
          ]
     |),
      (''Extension 4a'')
     (| ID = ''CO.e2'',
        Condition = ''The provided confirmation number is invalid'',
        ExtensionScenario =
          [
           Atom ''CO.s4a1'' APPLICATION ''diNI'' {},
           Failure ''CO.s4a2''
          ]
     |),
      (''Extension 4b'')
     (| ID = ''CO.e3'',
        Condition = ''The order cannot be cancelled'',
        ExtensionScenario =
          [
           Atom ''CO.s4b1'' APPLICATION ''diCC'' {},
           Failure ''CO.s4b2''
          ]
     |),
      (''Extension 6a'')
     (| ID = ''CO.e4'',
        Condition = ''Primary Actor decides to not cancel the order'',
        ExtensionScenario =
          [
           Atom ''CO.s6a1'' INTERACTION ''caCA'' {},
           Failure ''CO.s6a2''
          ]
     |)
    }
|)"
```

Figure G.5. Abstract Syntax of "Cancel Order" Use Case

```
constdefs
ReviewOrderUC :: UseCase
 "ReviewOrderUC == (|
   Name = ''Review Order'',
   Properties = (|
     Goal = ''Primary Actor reviews an existing order.'',
     PrimaryActor = ''Customer'',
     GoalLevel = USERGOAL,
     Precondition = ''Primary Actor is logged into the system.''
   |),
   MainSuccessScenario = [
      Atom ''RO.s1'' INTERACTION ''trRO'' {},
      Atom ''RO.s2'' APPLICATION ''pmCN'' {},
      Atom ''RO.s3'' INTERACTION ''prCN'' {''RO.e1''},
      Atom ''RO.s4'' INTERNAL ''riOR'' {''RO.e2''},
      Atom ''RO.s5'' APPLICATION    ''diSU'' {},
      Success ''s6''
   ],
   Extensions = [
     (|
      ID = ''RO.e1'',
      Condition = ''Primary Actor decides to cancel the use case'',
      ExtensionScenario =
       [
         Atom ''RO.s3a1'' INTERACTION ''caRO'' {},
         Failure ''RO.s3a2''
       ]
     |),
     (|
      ID = ''RO.e2'',
      Condition = ''The provided confirmation number is invalid'',
      ExtensionScenario =
       [
         Atom ''RO.s4a1'' APPLICATION ''diNI'' {},
         Failure ''RO.s4a2''
       ]
     |)
   ]
|)"
```

Figure G.6. Abstract Syntax of "Review Existing Order" Use Case

Appendix G.2 IMS ECTT Task Model

In this section we specify the UI requirements for the "IMS" system in form of an ECTT task model. Similar to the previous section, for the sake of readability, the task definitions are giving in both, graphical format and abstract syntax.

Appendix G.2.1 Graphical Task Definitions

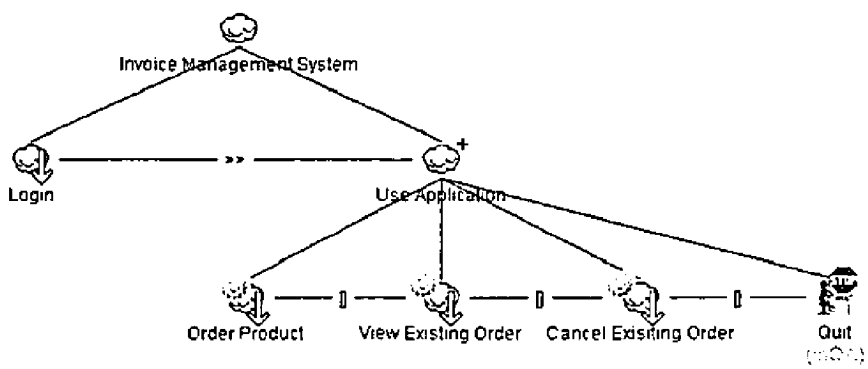


Figure G.7. Top-Level ECTT Task Definition of "IMS" System

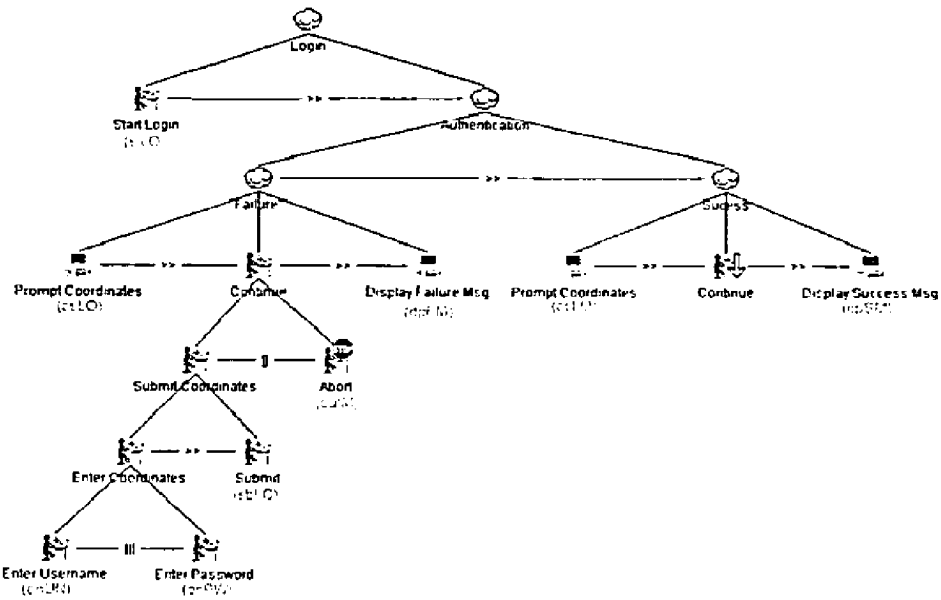


Figure G.8. "Login" ECTT Task Definition

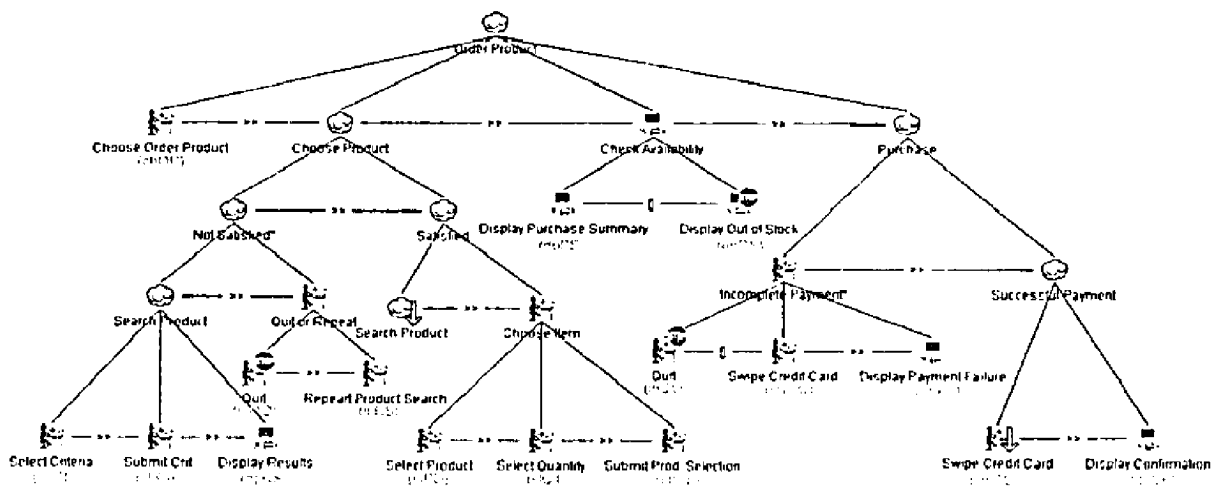


Figure G.9. "Order Product" Task Definition

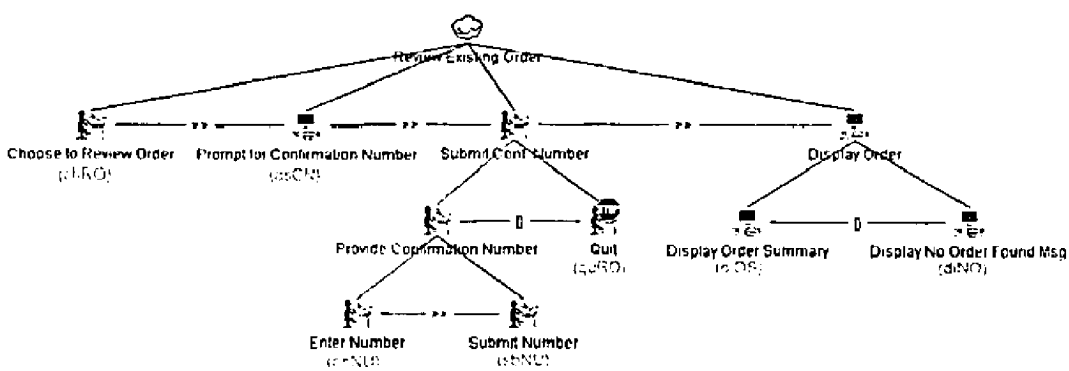


Figure G.10. "Review Existing Order" Task Definition

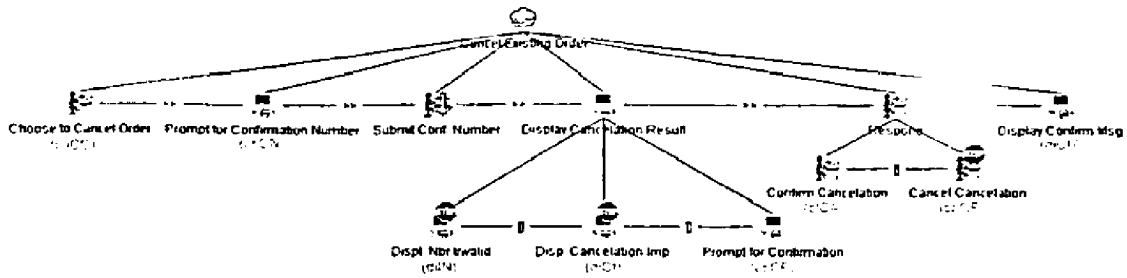


Figure G.11. "Cancel Existing Order" Task Definition

Appendix G.2.2 Formalization of IMS ECTT Model

We formally define the abstract ECTT syntax for the IMS task model as follows:

C_{IMS} = ("Invoice Management System", $\mathcal{D}, \tau$), with
$\mathcal{D}$ = {
"Invoice Management System" := "Login" ≫ ("Use Application")$^+$
"Use Application" := *resume* ("Order Product")[ ]
resume ("View Existing Order") [ ]
resume("Cancel Existing Order") [ ]
stop ("inQA")
"Login" := "stLO" ≫ "Authentication"
"Authentication" := ("Failure")* ≫ "Success"
"Failure" := "asLO" ≫ "Continue" ≫ "dpFM"
"Continue" := "Submit Coordinates"[ ] *stop*("quIM")
"Submit Coordinates" := Enter Coordinates" ≫ "sbLC"
"Enter Coordinates" := "enUN" ‖ "enPW"
"Success" := "asLO" ≫ "Continue" ≫ "dpSM"
"Order Product" := "chOP" ≫ "Choose Product" ≫ "Check Availability" ≫ "Purchase"
"Choose Product" := ("Not Satisfied")* ≫ "Satisfied"
"Not Satisfied" := "Search Product" ≫ "Quit or Repeat"
"Search Product" := "slCR" ≫ "sbCR" ≫ "dpRS"
"Quit or Repeat" := *stop*("inQO") [ ] "inRS"
"Satisfied" := ("Search Product" ≫ "Choose Item")
"Choose Item" := "slPD" ≫ "slQT" ≫ "sbPS"
"Check Availability" := "dpPS" [ ] *stop*("dpOS")
"Purchase" := ("Incomplete Payment")* ≫ "Successful Payment"
"Incomplete Payment" := *stop*("inQO")[ ] ("swCC" ≫ "dpPF")
"Successful Payment" := "swCC" ≫ "dpCF"
"Review Existing Order" := "chRO" ≫ "asCN" ≫ "Submit Conf. Number"
≫ "DisplayOrder"
"Submit Conf. Number" := "Provide Confirmation Number" [ ] "quRO"
"Provide Confirmation Number" := "enNU" ≫ "sbNU"
"DisplayOrder" := "diOS" [ ] "diNO"
Cancel Existing Order := "chCO" ≫ "asCN" ≫ "Submit Conf. Number"
≫ "Display Cancelation Result" ≫ "Respond" ≫ "diCM"
"Display Cancelation Result" := *stop*("diIN") [ ] *stop* ("diCI) [ ] "asCF"
"Respond" := *stop* ("cfCA")[ ] "caCF"

$$\tau(t) = \begin{cases} abstract, & if\ t \in \{\text{"InvoiceManagementSystem", " Login", "Use Application", "View Existing Order", "Cancel Existing Order", "Order Product", "Choose Product", "Make Decision", "Not Satisfied", "Redo", "Purchase", "Feedback", "Payment Failure", "Authentication", "Failure", "Success",}\} \\ interaction, & if\ t \in \{\text{"inQA", "chOP", "slCR", "sbCR", "inQO", "inRS", "Satisfied", slPD, slQT,sbPS,Proceed,swCC, slLO",Enter Coordinates", "enUN", "enPW", "Continue", "sbLC", "quIM", "chRO", "enNU", "sbNU", "Submit Conf. Number", "Provide Conf. Number" "quRO", "chCO", "Respond", "cfCA", "caCF"}\} \\ application, & if\ t \in \{\text{"dpRS", "CheckAvailability", "dpRS", "dpOS", "dpCF", "dpPF", "asLO", "dpFM", "dpSM", "asCN" "Display Order", "diOS", "diNO", "asCF", "diCM" "Display Cancelation Result", "dilN", "diCl"}\} \end{cases}$$

Appendix G.3 Intermediate Semantic Domain: UC-LTS

In this section we map the "IMS" DSRG-style use case model to the intermediate semantic domain of UC-LTSs. As an intermediate result of the mapping process, we obtain the UC-LTSs depicted in Figure G.12. States, that belong to the same equivalence class (~) are circled by a dashed line. Next, the equivalent states of the various UC-LTSs are merged and we obtain the consolidated UC-LTS given in Figure G.13.

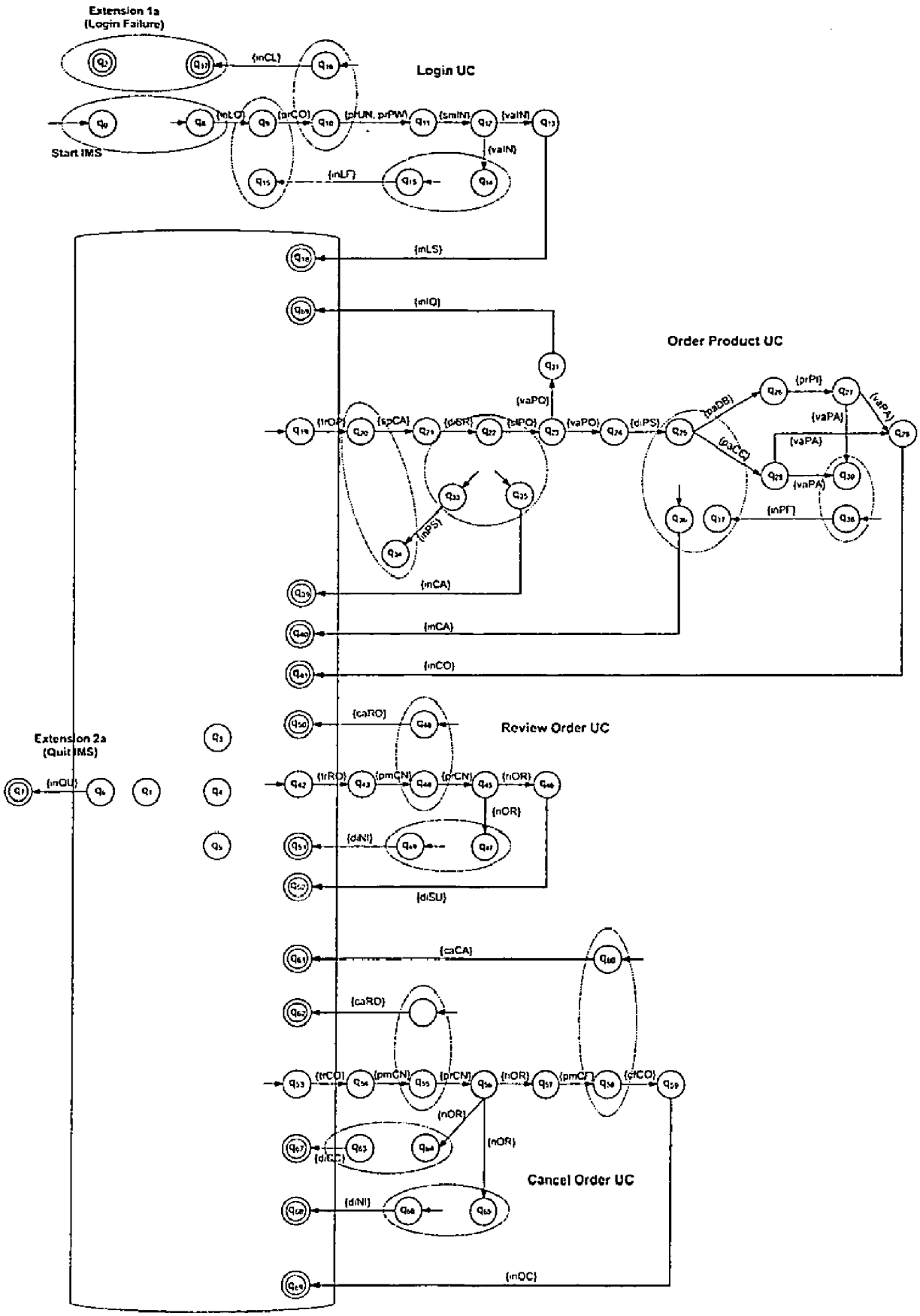


Figure G.12. UC-LTSs of the "IMS" Use Case Model

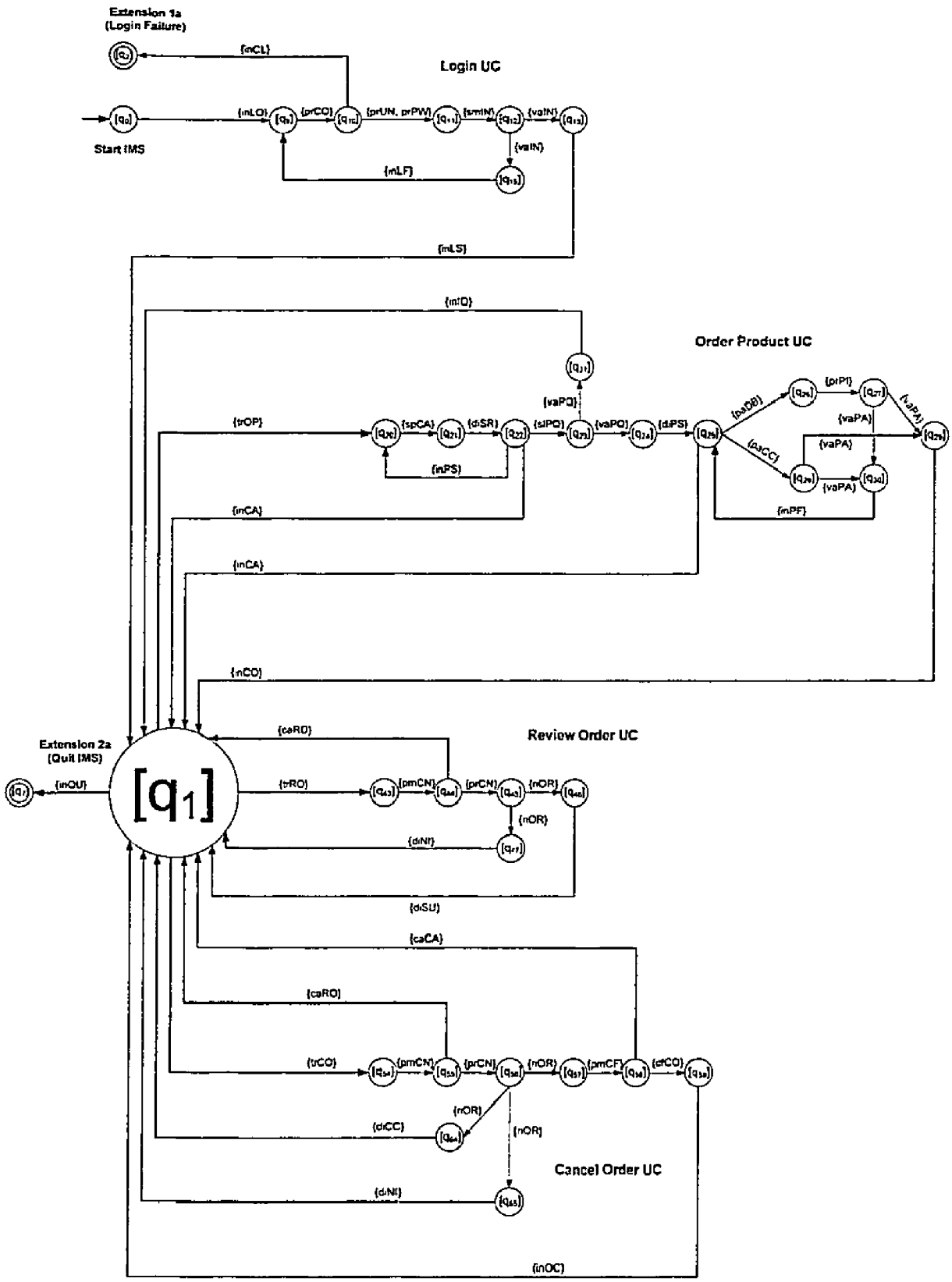


Figure G.13. Merged UC-LTS of the "IMS" Use Case Model

Appendix G.4 Intermediate Semantic Domain: GTM

In this section we map the "IMS" ECTT Model to the intermediate semantic domain of Generic Task Models.

$G_{IMS} = (T, \psi, \tau)$ with

$$T = \left\{ \begin{array}{l} \text{"stLO","asLO", "enUN", "enPW, "sbLC", "quIM", "dpFM", "dpSM", "dpRS", "dpOS", "dpCF", "dpPF",} \\ \text{"chOP",slCR",sbCR",inQO",inRS",slPD", slQT",sbPS",swCC",dpRS","chRO", "asCN", "enNU",} \\ \text{"sbNU", "quRO", "diOS","diNO", "chCO", "dilN", "diCI", "asCF", "cfCA", "caCF", "diCM", "inQA"} \end{array} \right\}$$

$\psi =$

"stLO" $\gg$ ("asLO" $\gg$ ((("enUN" $\|$ "enPW") $\gg$ "sbLC") [ ] *stop*("quIM") $\gg$ "dpFM"))$^{\bullet}$ $\gg$

("asLO" $\gg$ ((("enUN" $\|$ "enPW") $\gg$ "sbLC") [ ] *stop*("quIM") $\gg$ "dpSM")) $\gg$

(

resume("chOP" $\gg$ ("slCR" $\gg$ "sbCR" $\gg$ "dpRS" $\gg$ (*stop*("inQO") [ ] "inRS"))$^{\bullet}$ $\gg$

"slCR" $\gg$ "sbCR" $\gg$ "dpRS" $\gg$ "slPD" $\gg$ "slQT" $\gg$ "sbPS" $\gg$
("dpPS" [ ] *stop*("dpOS")) $\gg$ (*stop*("inQO") [ ] ("swCC" $\gg$ "dpPF"))$^{\bullet}$ $\gg$
"swCC" $\gg$ "dpCF")

[ ]

resume("chRO" $\gg$ "asCN" $\gg$ (("enNU" $\gg$ "sbNU") [ ] *stop* ("quRO")) $\gg$

("diOS" [ ] "diNO"))

[ ]

resume("chCO" $\gg$ "asCN" $\gg$ (("enNU" $\gg$ "sbNU") [ ] *stop* ("quRO")) $\gg$
(*stop*("dilN") [ ] *stop* ("diCI) [ ] "asCF") $\gg$ (*stop* ("cfCA") [ ] "caCF") $\gg$ "diCM")

[ ]

stop ("inQA")

)$^{\bullet}$ $\gg$

resume("chOP" $\gg$ ("slCR" $\gg$ "sbCR" $\gg$ "dpRS" $\gg$ (*stop*("inQO") [ ] "inRS"))$^{\bullet}$ $\gg$

"slCR" $\gg$ "sbCR" $\gg$ "dpRS" $\gg$ "slPD" $\gg$ "slQT" $\gg$ "sbPS" $\gg$
("dpPS" [ ] *stop*("dpOS")) $\gg$ (*stop*("inQO")[ ] ("swCC" $\gg$ "dpPF"))$^{\bullet}$ $\gg$
"swCC" $\gg$ "dpCF")

[ ]

resume("chRO" $\gg$ "asCN" $\gg$ (("enNU" $\gg$ "sbNU") [ ] *stop* ("quRO")) $\gg$

("diOS" [ ] "diNO"))

[ ]

resume("chCO" $\gg$ "asCN" (("enNU" $\gg$ "sbNU") [ ] *stop* ("quRO")) $\gg$
(*stop*("dilN") [ ] *stop* ("diCI) [ ] "asCF") $\gg$ (*stop* ("cfCA") [ ] "caCF") $\gg$ "diCM")

[ ]

stop ("inQA")

)

$$\tau(t) = \begin{cases} interaction, & if\ t \in \left\{ \begin{array}{c} \text{"stLO"},\text{"enUN"},\text{"enPW"},\text{"sbLC"},\text{"quIM"},\text{"inQA"},\text{"chOP"},\text{"slCR"}, \\ \text{"sbCR"},\text{"inQO"},\text{"inRS"},\text{"slPD"},\text{"slQT"},\text{"sbPS"},\text{"swCC"}, \\ \text{"chRO"},\text{"enNU"},\text{"sbNU"},\text{"quRO"},\text{"chCO"},\text{"cfCA"},\text{"caCF"} \end{array} \right\} \\ application, & if\ t \in \left\{ \begin{array}{c} \text{"asLO"},\text{"dpFM"},\text{"dpSM"},\text{"dpRS"},\text{"dpRS"},\text{"dpOS"},\text{"dpCF"}, \\ \text{"asCN"},\text{"diNO"},\text{"diOS"},\text{"diIN"},\text{"diCI"},\text{"asCF"},\text{"diCM"} \end{array} \right\} \end{cases}$$

Appendix G.5 Set of Posets Semantics

The set of posets representations of the "IMS" use case and task model are given in this section.

Appendix G.5.1 Set of Posets Formalization of "IMS" Use Case Model

```
-Login Use Case
[inLO] · ([prLO] · ([prUN] ‖ [prPW]) · [smIN, vaIN, inLF])* · [prLO, inCL] #
[inLO] · ([prLO] · ([prUN] ‖ [prPW]) · [smIN, vaIN, inLF])* ·
        ([prLO] · ([prUN] ‖ [prPW]) · [smIN, vaIN, inLS]) ·
 (
  -Order Product Use Case
  (
  [trOP] · [spCA, diSR, inPS]* · [spCA, diSR, inCA] #
  [trOP] · [spCA, diSR, inPS]* · [spCA, diSR, slPQ, vaPQ, inIQ] #
  [trOP] · [spCA, diSR, inPS]* · [spCA, diSR, slPQ, vaPQ, diPS] ·
    ([paDB, prPI, vaPA, inPF] # [paCC, vaPA, inPF])* ·
        [inCA] #
        [paDB, prPI, vaPA, inCO] #
        [paCC, vaPA, inCO]
  )
  #
  -Review Order Use Case
  (
  [trRO, pmsCN, caRO] #
  [trRO, pmCN, prCN, riOR, diNI] #
  [trRO, pmCN, prCN, riOR, diSU]
  )
  #
  -Cancel Order Use Case
  (
  [trCO, pmCN, quRO] #
  [trCO, pmCN, prCN, riOR, diCC] #
  [trCO, pmCN, prCN, riOR, diNI] #
```

```
    [trCO, pmCN, prCN, riOR, pmCF, caCA] #
    [trCO, pmCN, prCN, riOR, pmCF, cfCO, inOC]
   )
  )* · [inQU]   Quit IMS
```

Figure G.14. Set of Posets Formalization of "IMS" Use Case Model

Appendix G.5.2 Set of Posets Formalization of the "IMS" Task Model

```
 Login TM
[stLO] · ([asLO] · ([enUN] ‖ [enPW]) · [sbCR, dpFM])* · [asLO, quIM, STOP] #
[stLO] · ([asLO] · ([enUN] ‖ [enPW]) · [sbCR, dpFM])* ·
        ([asLO] · ([enUN] ‖ [enPW]) · [sbCR, dpSM]) ·
  (
    Order Product TM
    (
    [chOP] · [slCR, sbCR, dpRS, inRS]* · [slCR, sbCR, dpRS, inQO] #
    [chOP] · [slCR, sbCR, dpRS, inRS]* · [slCR, sbCR, dpRS, slPD, slQT, sbPS,dpOS ] #
    [chOP] · [slCR, sbCR, dpRS, inRS]* · [slCR, sbCR, dpRS, slPD, slQT, sbPS,dpPS] · [swCC, dpPF]*
                · [inQO] #
    [chOP] · [slCR, sbCR, dpRS, inRS]* · [slCR, sbCR, dpRS, slPD, slQT, sbPS,dpPS] · [swCC, dpPF]*
                · [swCC, dpCF]
    )
    #
    Review Existing Order TM
    (
    [chRO, asCN, quRO] #
    [chRO, asCN, enNU, sbNU, diNO] #
    [chRO, asCN, enNU, sbNU, diOS]
    )
    #
    Cancel Existing Order TM
    (
    [chCO, asCN, quRO] #
    [chCO, asCN, enNU, sbNU, diIN] #
    [chCO, asCN, enNU, sbNU, diCI] #
    [chRO, asCN, enNU, sbNU, asCF, caCF] #
    [chRO, asCN, enNU, sbNU, asCF, cfCF, diCM]
    )
  )* · [inQO, STOP]   Quit IMS
```

Figure G.15. Set of Posets Formalization of "IMS" Task Model

Appendix G.6 nFSM Semantics

The nFSM representations of the "IMS" use case and task model are given in this section.

Appendix G.6.1 nFSM Formalization of the "IMS" Use Case Model

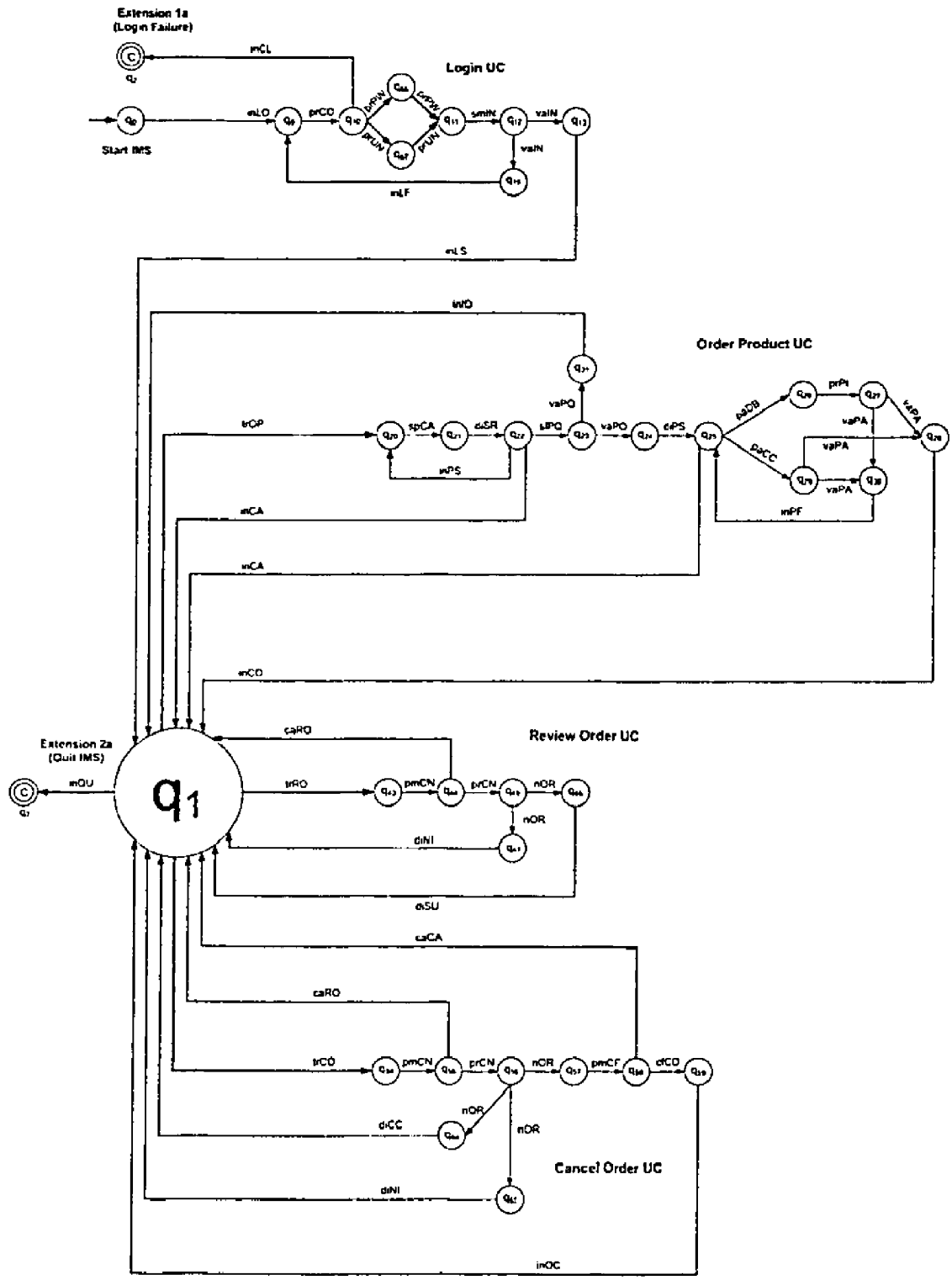


Figure G.16. nFSM Formalization of "IMS" Use Case Model

Appendix G.6.1 nFSM Formalization of the "IMS" Task Model

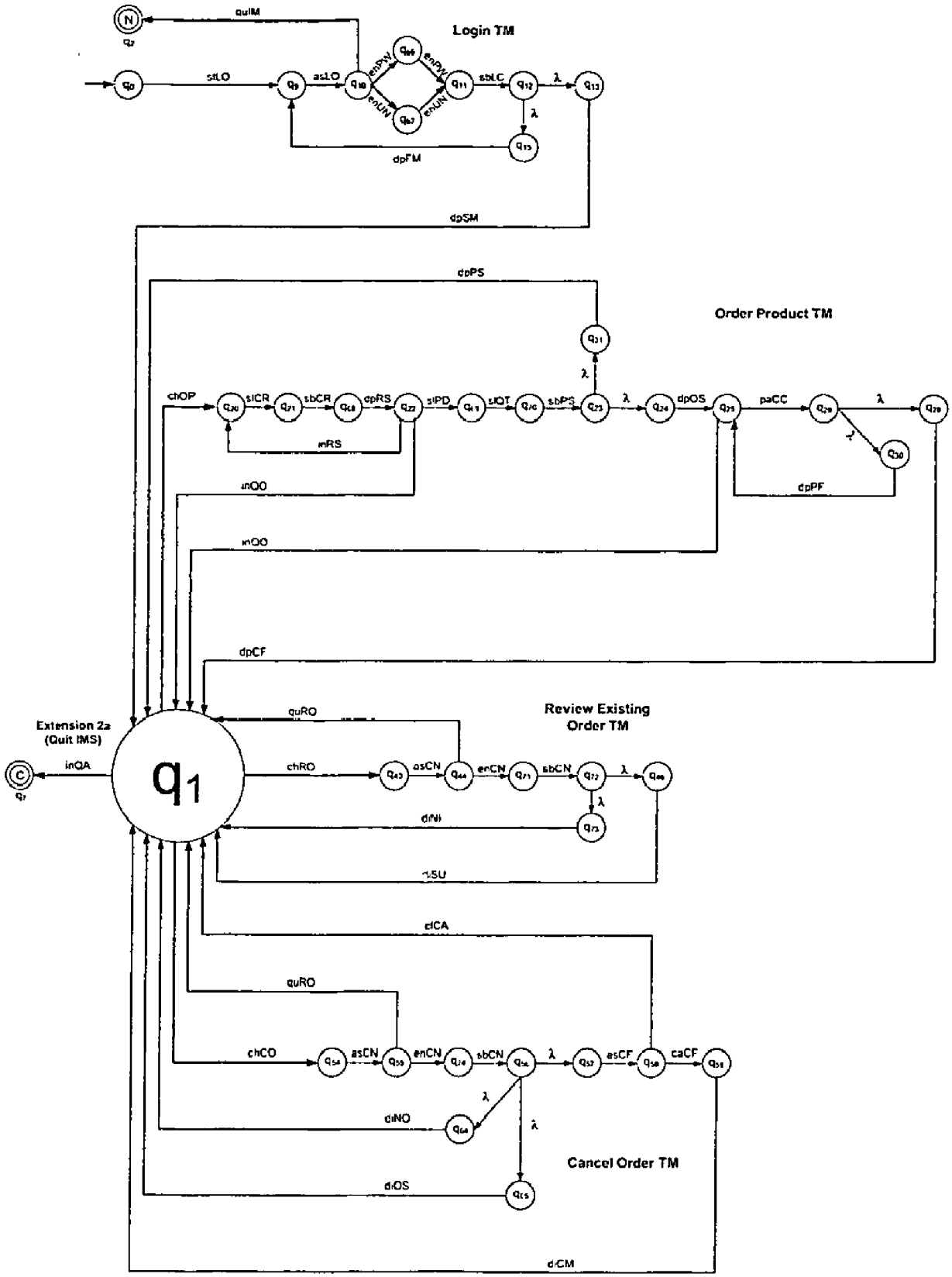


Figure G.17. nFSM Formalization of "IMS" Task Model

Appendix G.7 Refinement Verification

Using the *Use Case – Task Model Verifier* we prove that the "IMS" task model is a valid refinement of the "IMS" use case model. For this purpose, we specify the corresponding UC-LTS and GTM specifications in SSRL as follows (Figure G.18 – Figure G.20):

```
EVENTS:
  //Events in "IMS" UC-LTS
  inQU: INTERACTION; //Quit IMS
  //Login
  inLO: INTERACTION; prCO: APPLICATION; prUN: INTERACTION;
  prPW: INTERACTION; smIN: INTERACTION; vaIN: INTERNAL;
  inLS: APPLICATION;
  //Order Product
  trOP: INTERACTION; spCA: INTERACTION; diSR: APPLICATION;
  inPS: INTERACTION; inCA: INTERACTION; slPQ: INTERACTION;
  vaPQ: INTERNAL; diPS: APPLICATION; paDB: INTERACTION;
  paCC: INTERACTION; prPI: INTERACTION;  vaPA: INTERNAL;
  inCO: APPLICATION; inPF: APPLICATION; inIQ: APPLICATION;
  //Review an Existing Order
  trRO: INTERACTION; pmCN: APPLICATION; prCN: INTERACTION;
  riOR: INTERNAL; diSU: APPLICATION; caRO: INTERACTION;
  diNI: APPLICATION;
  //Cancel an Existing Order
  trCO: INTERACTION; pmCN: APPLICATION; prCN: INTERACTION;
  riOR: INTERNAL; pmCF: APPLICATION; cfCO: INTERACTION;
  inOC: APPLICATION; caRO: INTERACTION; diNI: APPLICATION;
  diCC: APPLICATION; caCA: INTERACTION;
  //Events in "IMS" Generic Task Model
  inQA: INTERACTION; //Quit IMS
  //Login
  stLO: INTERACTION; asLO: APPLICATION; enUN: INTERACTION;
  enPW: INTERACTION; quIM: INTERACTION; dpFM: APPLICATION;
  dpSM: APPLICATION;
  //Order Product
  chOP: INTERACTION; slCR: INTERACTION; sbCR: INTERACTION;
  dpRS: APPLICATION; inQO: INTERACTION; inRS: INTERACTION;
  slPD: INTERACTION; slQT: INTERACTION; sbPS: INTERACTION;
  dpPS: APPLICATION; dpOS: APPLICATION; swCC: INTERACTION;
  dpPF: APPLICATION; dpCF: APPLICATION;
  //Review Existing Order
  chRO: INTERACTION; asCN: APPLICATION; enNU: INTERACTION;
  sbNU: INTERACTION; quRO: INTERACTION; diOS: APPLICATION;
  diNO: APPLICATION;
  //Cancel Existing Order
  chCO: INTERACTION; asCN: APPLICATION; diIN: APPLICATION;
  diCI: APPLICATION; asCF: APPLICATION; cfCA: INTERACTION;
  caCF: INTERACTION; diCM: APPLICATION;
```

Figure G.18. Event Declaration Section for "IMS" UC-LTS and GTM

```
USECASEDEFS:
  IMSUCLTS = (
  //Login
  {(q0, {inLO}, q9), (q9, {prCO}, q10), (q10, {prUN, prPW}, q11),
  (q11, {smIN}, q12), (q12, {vaIN}, q13), (q13, {inLS},q1),
  (q10, {inCL}, q2), (q12, {vaIN}, q15), (q15, {inLF}, q9),
  //Order Product
  (q1, {trOP}, q20), (q20, {spCA}, q21), (q21, {diSR}, q22),
  (q22, {slPQ}, q23), (q23, {vaPQ}, q24), (q24, {diPS}, q25),
  (q25, {paDB}, q26), (q25, {paCC}, q29), (q26, {prPI}, q27),
  (q27, {vaPA}, q28), (q29, {vaPA}, q28), (q28, {inCO}, q1),
  (q22, {inPS}, q20), (q22, {inCA}, q1), (q23, {vaPQ}, q31),
  (q31, {inIQ}, q1), (q25, {inCA}, q1), (q27, {vaPA}, q30),
  (q29, {vaPA}, q30), (q13, {inPF}, q6),
  //Review Existing Order
  (q1, {trRO}, q43), (q43, {pmCN}, q44), (q44, {prCN}, q45),
  (q45, {riOR}, q46), (q46, {diSU}, q1), (q44, {caRO}, q1),
  (q45, {riOR}, diN1),
  //Cancel Existing Order
  (q1, {trCO}, q54), (q54, {pmCN}, q55), (q55, {prCN}, q56),
  (q56, {riOR}, q57), (q57, {pmCF}, q58), (q58, {cfCO}, q59),
  (q59, {inOC}, q1), (q55, {caRO}, q1), (q56, {riOR}, q65),
  (q65, {diNI}, q1), (q56, {riOR}, q64), (q64, {diCC}, q1),
  (q58, {caCA}, q1), (q1, {inQU}, q7)},
  {q7, q2}); //Set of finalStates of the UC-LTS
```

Figure G.19. UC-LTS Specification of "IMS" Use Case Model

```
TASKDEFS:
  IMSGTM =
  //Login
  stLO >> (asLO >> (((enUN || enPW) >> sbLC) [] STOP (quIM) >> dpFM))*>>
          (asLO >> (((enUN || enPW) >> sbLC) [] STOP (quIM) >> dpSM)) >>
  ( //Iteration through main functionality (0 or many times)
  //Order Product
   RESUME (chOP >> (slCR >> sbCR >> dpRS >> (STOP (inQO) [] inRS))* >>
           slCR >> sbCR >> dpRS >> slPD >> slQT >> sbPS >> (dpPS []
           STOP (dpOS)) >> (STOP (inQO) [] (swCC >> dpPF))* >>
           swCC >> dpCF)
  []
   //Review Existing Order
   RESUME (chRO >> asCN >> ((enNU >> sbNU) [] STOP (quRO)) >>
          (diOS [] diNO))
   []
   //Cancel Existing Order
   RESUME (chCO >> asCN >> ((enNU >> sbNU) [] STOP (quRO)) >>
          (STOP (diIN) [] STOP (diCI) [] asCF) >>
          (STOP (cfCA) [] caCF) >> diCM)
   []
   STOP (inQA) //Quit IMS
   )* >>
   //Repetition of main functionality
   //Order Product
   RESUME (chOP >> (slCR >> sbCR >> dpRS >> (STOP (inQO) [] inRS))* >>
           slCR >> sbCR >> dpRS >> slPD >> slQT >> sbPS >> (dpPS []
           STOP (dpOS)) >> (STOP (inQO) [] (swCC >> dpPF))* >>
           swCC >> dpCF)
   []
   //Review Existing Order
   RESUME (chRO >> asCN >> ((enNU >> sbNU) [] STOP (quRO)) >>
          (diOS [] diNO))
   []
   //Cancel Existing Order
   RESUME (chCO >> asCN >> ((enNU >> sbNU) [] STOP (quRO)) >>
          (STOP (diIN) [] STOP (diCI) [] asCF) >>
          (STOP (cfCA) [] caCF) >> diCM)
   []
   STOP (inQA) //Quit IMS
```

Figure G.20. GTM Specification of "IMS" Task Model

The refinement mapping and the assertion that the "IMS" task model is a deterministic reduction of the "IMS" use case model are given in Figure G.21.

```
REFMAPS:
  //Refinement Mapping from IMSGTM to IMSUCTS
  IMSUCtoTM = {
  //Login
  ({stLO}, inLO), ({asLO}, prCO), ({quIM}, inCL), ({enUN}, prUN),
  ({enPW}, prPW), ({sbLC}, smIN), ({dpFM}, inLF), ({dpSM}, inLS),

  //Order product
  ({chOP},trOP),({slCR,sbCR},spCA),({dpRS},diSR),({inQO},inCA),
  ({inRS},inPS), ({slPD, slQT,sbPS},slPQ),({dpPS},inIQ),({dpOS},diPS),
  ({swCC},paCC),({dpPF},inPF),({dpCF},inCO),

  //Revise Existing Order
  ({chRO}, trRO), ({asCN}, pmCN), ({quRO}, caRO), ({enCN, sbCN}, prCN),
  ({diNO}, diNI), ({diOS}, diSU),

  //Cancel Existing Order
  ({chCO}, trCO), ({diIN}, diCC), ({diCI}, diNI),({asCF}, pmCF),
  ({cfCA}, caCA), ({caCF}, cfCO), ({diCM}, inOC),

   //Quit IMS
   ({inQA}, inQU)};
ASSERT:
   /*Assert that the IMS GTM is a deterministic reduction of the
     IMS UC-LTS. */
   IMSUCLTS DET IMSGTM WITH IMSUCtoTM;
```

Figure G.21. Refinement Mapping and Assertion for "IMS" UC-LTS and GTM

The result of the verification is captured in Figure G.22. As expected, under the assumption that the "IMS" task model is a requirements artifact, it is a valid refinement of the use case model.

```
Verification Result:

IMSUCLTS DET IMSGTM WITH IMSUCtoTM: Valid Refinement: (159 state pairs
explored)
```

Figure G.22. Positive Verification Result

INDEX