Common Semantics for Use Cases and Task Models

Daniel Sinnig[1], Patrice Chalin[1] and Ferhat Khendek[2]

[1] Department of Computer Science and Software Engineering,
Concordia University, Montreal, Quebec, Canada
{d_sinnig, chalin}@encs.concordia.ca

[2] Department of Electrical and Computer Engineering,
Concordia University, Montreal, Quebec, Canada
khendek@ece.concordia.ca

Abstract.
In this paper, we introduce a common semantic framework for developing and formally modeling use cases and task models. Use cases are the notation of choice for functional requirements specification and documentation, whereas task models are used as a starting point for user interface design. Based on their intrinsic characteristics we devise an intermediate semantic domain for use cases and for task models, respectively. We describe how the intermediate semantic domain for each model is formally mapped into a common semantic domain which is based on sets of partial order sets. We argue that a two-step mapping results in a semantic framework that can be more easily validated, reused and extended. As a partial validation of our framework we provide a semantics for ConcurTaskTrees (CTT) one of the most popular task model notations as well as our own DSRG use case formalism. Furthermore we use the common semantic model to formally define a satisfiability relation between task model and use case specifications.

Keywords: Use cases, task models, requirements, formal semantics, partial order sets, labeled transition systems.

1 Introduction

User Interface (UI) design and the engineering of functional requirements are generally carried out by different teams using different methodologies, processes and lifecycles [1]. Since both disciplines have their own models and theories, often the respective artifacts are created independently of each other; as a result there arises:

- Duplication in effort during development and maintenance due to redundancies / overlaps in the (independently) developed UI and software engineering models.
- Possible conflicts during implementation as both processes do not have the same reference specification and thus may result in inconsistent designs.

A process allowing for UI design to follow as a logical progression from functional requirements specification does not exist.

Use cases are the artifacts of choice for functional requirements specification and documentation [2] while UI design typically starts with the identification of user tasks, and context requirements [3]. Our primary research goal is to define an integrated methodology for the development of use cases and task models within an overall software process. A prerequisite of this initiative is the definition of a formal framework for handling use case models and task models. The cornerstone for such a formal framework is a common semantic domain for both notations.

Figure 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 based on *sets of partial order sets*. The common semantic model will serve as a reference for tool support and will be the basis for the definition of a satisfiability relation between a use case specification and a task model specification. A definition of the latter is given in this paper.

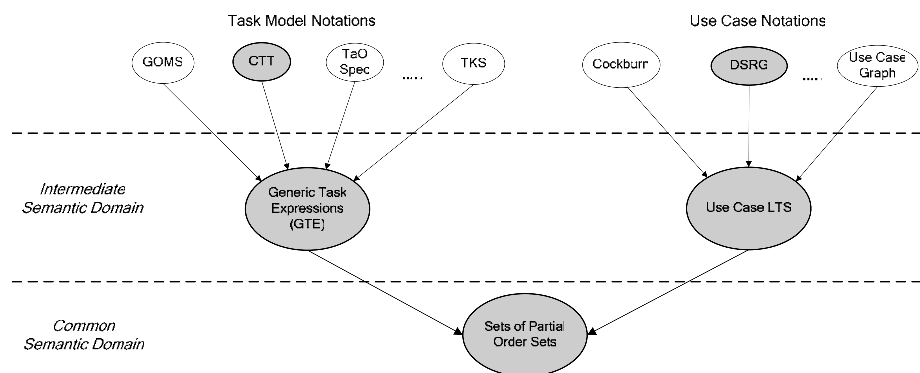

Fig. 1. Two-Step Semantic Mapping

The main reason behind a two-step mapping, rather than a direct mapping, is to provide 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, respectively, so that the mappings to the intermediate semantic domains are straightforward and intuitive: task models are mapped into what we call Generic Task Expressions (GTE); use cases are mapped to Use Case Labeled Transition Systems (UC-LTS). Since the second level mappings to sets of posets 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.

In this paper, we focus on providing concise definitions of both the intermediate semantic domains for use cases and task models and the common semantic model. As concrete examples of mappings, we illustrate how ConcurTaskTree (CTT) [4] specifications and DSRG-style use cases (defined in the next section) are mapped to the intermediate semantic domains. This is followed by a formalization of the second level mappings of GTEs and UC-LTSs into the sets of posets.

The remainder of this paper is organized as follows. In Section 2 we provide necessary background information by reviewing and contrasting use cases and task models. Section 3 discusses related work with respect to the definition of semantics of scenario-based notations. Section 4, formally defines our semantic framework. Finally, in Section 5 we conclude and provide an outlook of future work.

2 Background

In this section we remind the reader of the key characteristics of use cases and task models. For each model we present a particular notation, and an illustrative example. Finally, both models are compared and main commonalities and differences are contrasted.

2.1 Use Case Models

A use case captures the interaction between actors and the system under development. It is organized as a collection of related success and failure scenarios that are all bound to the same goal of the primary actor [5]. Use cases are typically employed as a specification technique for capturing functional requirements. They document the majority of software and system requirements and as such, serve as a contract (of the envisioned system behavior) between stakeholders [2].

Every use case starts with a header section containing various properties of 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 ways 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 constitute 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 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.

Different notations at different degrees of formality have been suggested as a medium to capture use cases. The extremes range from purely textual constructs written in prose language [2] to entirely formal specification written in Z [6], or as Abstract State Machines (ASM) [7, 8]. While the use of narrative languages makes use case modeling an attractive tool to facilitate communication among stakeholders, prose language is well known to be prone to ambiguities and leaves little room for advanced tool support.

Therefore, in this paper we take up a compromise solution, which enforces a formal structure (needed for the definition of formal semantics) but preserves the intuitive nature of use case. In particular, we have developed an XML Schema

(depicted in Figure 2), which acts as a meta model for use cases. As such, it identifies the most important use case elements, defines associated mark-up and specifies existing containment relationships among elements. We refer to use cases that correspond to the schema presented in Figure 2 as "DSRG-style use cases".

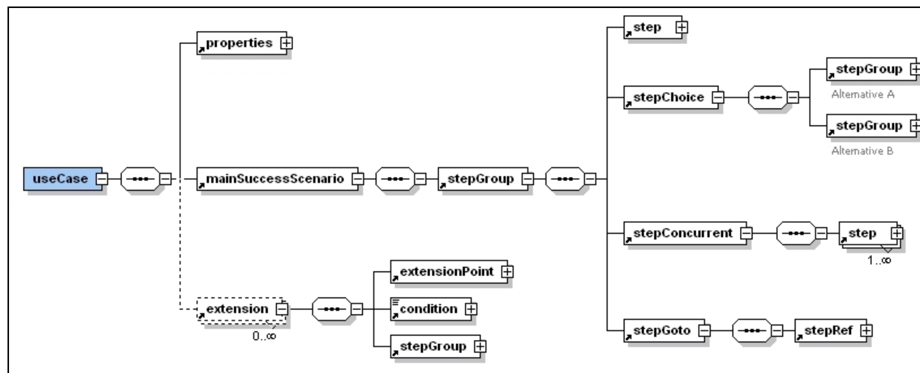

Fig. 2. DSRG Use Case Meta Model

Most relevant for this paper is the definition of the *stepGroup* element as it captures the behavioral information of the use case. As depicted, the *stepGroup* element consists of a sequence of one of the following sub elements:

- The ***step*** element denotes an atomic use case step capturing the primary actor's interactions or system activities.
- The ***stepChoice*** element denotes the alternative composition of two *stepGroup* elements.
- The ***stepConcurrent*** element entails a set of (atomic) *step* elements, whose execution order is not defined.
- The ***stepGoto*** element denotes an arbitrary branching to another *step*.

We note that the *stepGroup* element is part of the *mainSuccessScenario* as well as the *extension* element. The latter additionally contains a condition and a reference to one or many steps stating *why* and *when* the extension may occur.

In order to generate a readable representation of the use case XML document we use XSLT style sheets [9]. Figure 3 depicts the generated HTML presentation of a sub-function level use case for a "Login" function. Note that we will be using the same "Login" example throughout this paper, and for the sake of simplicity, have kept the complexity of the use case to a minimum.

Use Case: Login

Properties

- **Primary Actor:** Customer
- **Goal:** Customer logs into the program.
- **Level:** Sub-function

Main Success Scenario

1. Primary Actor indicates that he/she wishes to log-in to the system.
2. Primary Actor *performs the following in abritrary order.*
   2.1 The Primary Actor provides the user name.
   2.2 The Primary Actor provides the password.
3. Primary Actor confirms the provided data.
4. System authenticates the Primary Actor.
5. System informs the Primary Actor that the Login was successful.
6. System grants access to the Primary Actor based on his/her access levels.

Extensions

4a. The provided username or/and password is/are invalid:
4a1. System informs the Primary Actor that the provided username or/and password is/are invalid
4a2. System denies access to the Primary Actor.

Fig. 3. Generated HTML Presentation of the "Login" Use Case

2.2 Task Models

User task modeling is by now a well understood technique supporting user-centered UI design [4]. In most UI development approaches, the task set is the primary input to the UI design stage. Task models describe the 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 [10], where a human operator carries out activities to change part of the environment (artifacts) in order to achieve a certain goal [11]. Like use cases, task models describe the user's interaction with the system. The primary purpose of task models is to systematically capture the way users achieve a goal when interacting with the system [12]. More precisely, the task model specifies how the user makes use of the system to achieve his/her goal but also indicates how the system supports the user tasks.

Various notations for task models exits. Among the most popular ones are ConcurTaskTrees (CTT) [4], GOMS [13], TaO Spec [14], and TKS [15]. 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. Atomic tasks are also called actions, since they are the tasks that are actually carried out by the user or the system. The execution order of tasks is determined by operators that are defined between peer tasks.

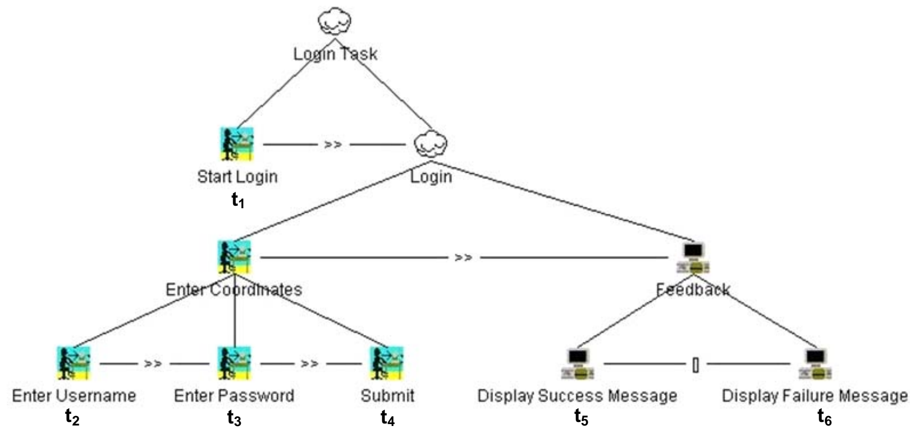

Fig. 4. “Login” Task Model

Figure 4 shows a CTT visualization of the “Login” task model. The figure illustrates the hierarchical break down and the temporal relationships between tasks involved in the “Login” functionality (depicted in the use case of Section 2.1). An indication of task types is given by the symbol used to represent tasks. In CTT the execution order between tasks is defined by temporal operators. Various temporal operators exist; examples include: enabling (>>), choice ([]), iteration (*), and disabling ([>].A complete list of the CTT operators together with an informal definition of their interpretation can be found in [4]. In Section 4.2 we will assign formal semantics to CTT task models by defining a mapping to the intermediate semantic domain of generic task expressions.

2.3 Use Cases vs. Task Models: A Comparison

In the previous two sections, the main characteristics of use cases and task models were discussed. In this section, we compare both models and outline noteworthy differences and commonalities.

Both, use cases and task models, belong to the family of scenario-based notations and as such capture sets of usage scenarios of the system. On the one hand, a use case specifies system behavior by means of a main success scenario and any corresponding extensions. On the other hand, a task model specifies system interaction within a single “monolithic” task tree. In theory, both notations can be used to describe the same information. In practice however, use cases are mainly employed to document functional requirements whereas task models are used to describe UI requirements/design details. Based on this assumption we identify three main differences which are pertinent to their purpose of application:

1. Use cases capture requirements at a higher level of abstraction whereas task models are more detailed. Hence, the atomic actions of the task model are often lower level UI details that are irrelevant (actually contraindicated [2]) in the

context of a use case. We note that due to its simplicity, within our example, this difference in the level of abstraction is not explicitly visible.

2. Task models concentrate on aspects that are relevant for UI design and as such, their usage scenarios are strictly depicted as input-output relations between the user and the system. Internal system interactions (i.e. involvement of secondary actors or internal computations) as specified in use cases are not captured.
3. If given the choice, a task model may only implement a subset of the scenarios specified in the use case. Task models are geared to a particular user interface and as such must obey its limitations. E.g. a voice user interface will most likely support less functionality than a fully-fledged graphical user interface.

3 Related Work

For scenario-based notations, the behavioral aspects of a system (capturing the ordering and relations between the events) represent the important features to describe. While several different formalisms have been proposed for scenario-based notations, in what follows we briefly discuss three prominent approaches, namely: process algebras, partial orders and graph structures.

Process Algebra has been widely used to define *interleaving* semantics of scenario-based notations [17-19]. 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) [20, 18]. This work is a continuation of preliminary research work by Mauw and Reniers [17]. In more recent work, Xu et. al. also suggest a process algebraic semantics for use case models, with the overall goal of formalizing use case refactoring [19]. In their approach, scenarios are represented as basic MSCs. The authors assign meaning to a particular use case scenario (episode) by partially adapting the ITU MSC semantics.

Formalisms suitable for the definition of *non-interleaving* semantics are based on partial orders. For example, Zheng et. al. propose a non-interleaving semantics for timed MSC 2000 [21, 22] based on timed labeled partial order sets (lposets). Partial order semantics for (regular, un-timed) MSCs has been proposed by Alur [23], and Katoen and Lambert [24]. Alur et. al. propose a semantics for a subset of MSCs that restricts MSC event types to message events only.

Mizouni et. al. propose use case graphs as an intermediate notation for use cases [25]. Use case graphs are directed, potentially cyclic graphs whose edges represent use case steps and nodes represent system states. This allows for a natural representation of the order in which actions are to be performed. Structural operational semantics for CTT task models are defined in [26]. In particular Paternò defines a set of inference rules to map CTT terms into labeled transition systems.

The semantic framework proposed in this paper is inspired by the lposet approach proposed in [22]. Similar to the approach in [22], our semantic framework is based on sets of partial order sets. The main motivation for this choice was the quest for a true, non-interleaving, model of concurrency. System behavior is represented as causally inter-related events based on a partial order relation. Events, that are not causally related, are seen as concurrent. In addition, similar to the work in [25], we employ

labeled graph structures (Use Case LTS) as an intermediate notation for use cases. Preliminary results towards the definition of a common semantic model for use cases and task models were reported in [27]. In this paper we complete and define our framework as a two-step mapping process, provide a formal semantics for all CTT expressions, and formalize the mapping from DSRG-style use cases to partial order sets using the intermediary notation of Use Case LTS.

4 Semantics for Use Cases and Task Models

In the previous section we have studied key characteristic of use cases and task models and reviewed relevant related work. In this section we re-employ this information to define a common formal semantics for use cases and task models. We start with the definition of the intermediate semantic domains. Then we define the common semantic model based on sets of partial order sets and specify the corresponding mappings from the intermediate domains. We conclude the section by providing a formal definition of a *satisfiability* relation based on the common semantic model.

4.1 Intermediate Semantic Domain for Use Cases

In this section we define an intermediate semantic domain, UC-LTS, for use cases and specify how DSRG-style use cases are transformed into UC-LTS.

Definition 1: (UC-LTS). *A use case labeled transition system* (UC-LTS) is defined by the tuple (S, Q, q_0, F, T), where:

S is the set of labels of atomic use case steps.
Q is a set of states.
$q_0 \in Q$ is the initial state.
$F \subseteq S$ is the set of final states.
$T = Q \times 2^S \times Q$ is the set of transitions.

We have defined UC-LTS in order to capture easily and intuitively the nature of use cases. A use case primarily describes the 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. Sometimes, the execution order of two or more steps is not important or just abstracted away for the purpose of the description. In UC-LTS the execution of a step is denoted by a labeled transition, from a source state to a target state. The transition labels serve as references to the corresponding steps in the original use case. The execution order of use case steps is modeled using transition sequences, where the target state of a transition serves as a source state of the following transition.

Contrary to LTSs, the labels in the UC-LTS are sets. For a given transition, if this set contains more than one label, then no specific execution order exists between the corresponding use case steps. This partial order semantics reflects better the nature of use cases.

In what follows we illustrate how use cases in DSRG style are transformed to the intermediate UC-LTS form. As the mapping turns out to be quite straightforward we will only sketch out the main translation principles. Given a UC-LTS consisting of a single state q_0 and a DSRG-style use case specification, iterate through the steps of *stepGroup* of the Main Success Scenario. For each found element, perform the following (depending on the type), using q_0 as a starting state:

- ***Step*****:** Create a new state q_{new} and define the following transition: (q_{last}, {label}, q_{new}) where q_{last} is the last state that has been created and 'label' is a (unique) identifier of the currently visited use case step. If there exists an extension for the currently visited step then, using q_{new} as a starting state, recursively repeat the same procedure for each step defined in the *stepGroup* of the extension.
- ***stepChoice*****:** For each of the two entailed *stepGroup* elements recursively re-perform this procedure with q_{last} as a starting state.
- ***stepConcurrent*****:** Create a new state q_{new} and define the following transition: (q_{last}, L, q_{new}) where q_{last} is the last state that has been created and L is the set of labels of all the step elements entailed in the *stepConcurrent* element. If there exist an extension for the *stepConcurrent* element then, using q_{new} as a starting state, recursively repeat the same procedure for each step defined in the *stepGroup* of the extension.
- ***stepGoto:*** Continue with the target step referenced in *stepGoto* element. If the target step has been already visited then replace q_{last} with the target step and update all transition definitions that included q_{last}, accordingly.

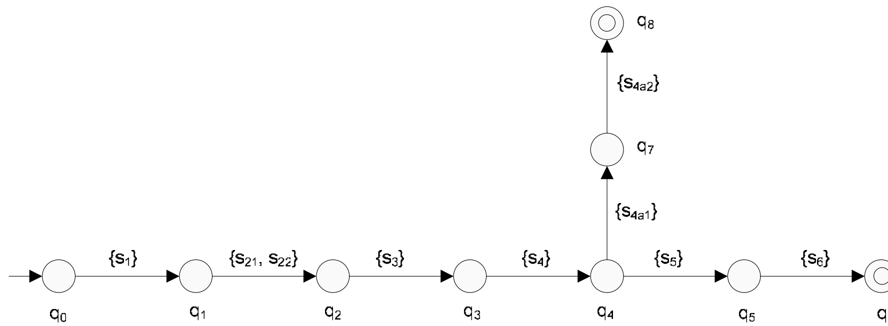

Fig. 5. Intermediate UC-LTS corresponding to the "Login" Use Case

Figure 5 illustrates the UC-LTS generated from the use case of Figure 3. Note that the transition between state q_1 and state q_2 has been annotated with labels of two use case steps, denoting the concurrent execution of use case step 2.1 and step 2.2. It is also to be noted that starting from state q_4 two transitions are defined, denoting the execution of step 5 in the main success scenario and alternatively the execution of step 4a1 defined in extension 4.

4.2 Intermediate Semantic Domain for Task Models

In this section we define an intermediate semantic domain for task models called Generic Task Expressions (GTE) and specify how a CTT specification (possibly including "Disabling" and "Suspend / Resume") is mapped into a corresponding GTE specification. In Section 2.2 we noted that tasks are commonly decomposed into subtasks and sub-subtasks until an atomic level is reached. For the definition of GTE we adopted the same paradigm and define a task expression as either an atomic action or a composition of (sub) task expression.

Definition 2: (Generic Task Expression). A generic task expression T is recursively defined as follows:

(1) An atomic action α is a generic task expression ($\alpha \in$ T)

(2) If ψ and ρ are generic task expressions (ψ, $\rho \in$ T) then

ψ^{Opt},

ψ^{Rep},

ψ _|| ρ,

ψ _[] ρ,

ψ _>> ρ,

are also generic task expressions.

Please note that the operator precedence is reflected by the order of their enumeration in Definition 2. Operators listed at a higher position have a higher precedence than operators listed at a lower position. Intuitively the meaning of the operators is as follows: The binary operators _>>, _||, and _[] denote the sequential, concurrent or alternative composition of two generic task expressions. The unary operators 'Opt' and 'Rep' denote the optional or the iterative (zero to infinitely many times) execution of a generic task expression.

In what follows we demonstrate how CTT task models are mapped to GTE. More precisely, we assign for each CTT task expression a corresponding denotation expressed in GTE. At the atomic level, we define that *CTT leaf tasks* correspond to *atomic GTE expressions (α).* At the composite level, CTT expressions entailing basic operators are mapped in a one-to-one manner to the corresponding GTE expressions. As depicted in Table 1, the only exception is the "Order_Independency" operator which is translated into the shallow interleaving of its operands. In order to illustrate the basic mapping, let us use again the "Login" task model from Section 2.2. According to the definitions of Table 1 the CTT specification is mapped into the following GTE specification: t_1 _>> t_2 _>> t_3 _>> t_4 _>> (t_5 _[] t_6).

Unfortunately, the mappings of the complex binary operators *disabling* and *suspend/resume* are not straightforward and require a pre-processing of their operands.

Table 1. Mappings of Basic CTT Operators into GTE.

CTT Expression			GTE Expression
t_l >> t_r	(Enabling)	=	t_l _>> t_r
t_l ||| t_r	(Concurrency)	=	t_l _|| t_r

t_l [] t_r	(Choice)	=	t_l _[] t_r
t*	(Iteration)	=	t^{Rep}
(t)	(Optional)	=	t^{Opt}
t_l |+| t_r	(Order Indepen.)	=	(t_l _>> t_r) _[] (t_r _>> t_l)

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, a term including the *disabling* operator can be rewritten as the "optionalization" of all tasks involved in the first operand, followed by the execution of the second operand.

For the purpose of "optionalizing" the first operand we have defined the unary auxiliary operator *Deep Optionalization* ({}). As inductively defined in Table 2, the application of the operator defines every subtask of its target task expression as optional. However if the subtasks are executed, they have to be executed in their pre-defined order. The final mapping of the disabling operator to an AGT expression, using the Deep Optionalization operator can be found in Table 3. We note that the definition of the CTT *disabling* operator has been inspired by the disabling operator of the LOTOS process algebra [28]. Yet, the interpretation of both operators is *not* identical. In particular, in LOTOS the subsequent execution of the second operand, after completion of the first one is not allowed.

Table 2. Inductive Definitions of "Deep Optionalization" and "Interleaved Insertion"

(Unary) Deep Optionalization {}			(Binary) Interleaved Insertion ⊕		
$\{\alpha\}$	=	α[1]	$\alpha \oplus t_i$	=	t_i _>> α
$\{t_l$ _>> $t_r\}$	=	$(\{t_l\}$ _>> $(\{t_r\})^{Opt})^{Opt}$	$(t_l$ _>> $t_r) \oplus t_i$	=	$(t_l \oplus t_i)$ _>> $(t_r \oplus t_i)$
$\{t_l$ _|| $t_r\}$	=	$(\{t_l\})^{Opt}$ || $(\{t_r\})^{Opt}$	$(t_l$ _|| $t_r) \oplus t_i$	=	$(t_l \oplus t_i)$ _|| $(t_r \oplus t_i)$
$\{t_l$ _[] $t_r\}$	=	$(\{t_l\} + \{t_r\})^{Opt}$	$(t_l$ _[] $t_r) \oplus t_i$	=	$(t_l \oplus t_i)$ _[] $(t_r \oplus t_i)$
$\{t^{Opt}\}$	=	$(\{t\})^{Opt}$	$(t^{Opt} \oplus t_i)$	=	$(t \oplus t_i)^{Opt}$
$\{t^{Rep}\}$	=	t^{Rep} _>> $(\{t\})^{Opt}$	$(t^{Rep} \oplus t_i)$	=	$(t \oplus t_i)^{Rep}$

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. In contrast to *disabling*, however, 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 [4]. At this point, the task specified by the first operand may continue its execution or may be interrupted again by the execution of the second operand.

[1] α denotes an atomic action

Table 3. Mappings of Disabling and Suspend/Resume into GTE.

CTT Expression			GTE Expression
t_l [> t_r	(Disabling)	=	$(\{t_l\})^{Opt}$ _>> t_r
t_l |> t_r	(Suspend/Resume)	=	$t_l \oplus (t_r^{Rep})$

In order to model this behavior, we have defined the auxiliary binary operator *Interleaved Insertion* (⊕). As defined in Table 2 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 define a mapping from a *suspend/resume* CTT expression to a corresponding GTE expression (Table 3).

4.3 Common Semantic Domain Based on Sets of Posets

This section defines the second-level mapping of our semantic framework. We start by providing necessary definitions. Next we present a semantic function that maps GTE specifications into the common semantic domain. Finally we specify an algorithm that generates a set of posets from a UC-LTS.

4.3.1 Notations and Definitions

The common semantic domain of our framework is based on sets of partial order sets (posets). In what follows we provide definitions of the involved formalisms and specify a set of operations needed for the semantic mapping. It is also in this section, where we propose a notion of refinement between two sets of posets specifications.

Definition 3: (Poset). A partially ordered set (poset) is a tuple $(E,\leq)$, where
E is a set of events, and
$\leq \subseteq E \times E$ is a partial order relation (reflexive, anti-symmetric, transitive) defined on E. This relation specifies the causal order of events.
We will use the symbol $\varnothing_{poset}$ to denote the empty poset with $\varnothing_{poset} = (\varnothing,\varnothing)$. Further we will use the symbol e_{poset} to denote a poset containing a single event e ($e_{poset} = (\{e\}, \{(e,e)\})$).

In order to be able to compose posets we define the following operations:

Definition 4: (Binary Operations on Posets). The binary operations: *sequential composition* (.) and *parallel composition* (||) of two posets p and q are defined as[2]:
Let $p = (E_p, \leq_p)$ and $q = (E_q, \leq_q)$ with $E_p \cap E_q = \varnothing$ then:
$p.q = (E_p \cup E_q, (\leq_p \cup \leq_q \cup \{(e_p, e_q) \mid e_p \in E_p \text{ and } e_q \in E_q\})^*)$
$p \| q = (E_p \cup E_q, \leq_p \cup \leq_q)$

[2] Note that R* denotes the reflexive, transitive closure of R.

We define semantics for GTE and UC-LTS using the following operations over sets of posets.

Definition 5.1: (Binary Operators on Sets of Posets). For two sets of posets P and Q, *sequential composition* (.), *parallel composition* (||), and *alternative composition* (#) are defined as follows:

$P \,.\, Q = \{ p_i \,.\, q_j \mid p_i \in P \text{ and } q_j \in Q \}$
$P \,//\, Q = \{ p_i \,\|\, q_j \mid p_i \in P \text{ and } q_j \in Q \}$
$P \,\#\, Q = P \cup Q$

Definition 5.2: (Repeated Sequential Composition). The *repeated sequential composition* of a set of posets P is defined as:

$P^0 = \{\varnothing_{\text{poset}}\}$
$P^n = P^{n-1}.\ P$ for $n > 0$
$P^* = P\ .\ P.\ \cdots$

Definition 5.3: (Iterated Alternative Sequential Composition). The *iterated alternative sequential composition* of a set of posets P is defined as:

$P_\#^0 = \{\varnothing_{\text{poset}}\}$
$P_\#^n = P^0 \# P^1 \# \ldots \# P^n$
$P_\#^* = P^0 \# P^1 \# \ldots$

Also fundamental to our model is the notion of a *trace*. A trace corresponds to one particular scenario defined in the original use case or task model specification. In the following we define the set of traces for a given poset, and for a given set of posets.

Definition 6: (Trace). A *trace* t of a poset $p = (E, \leq)$ is defined as a (possibly infinite) sequence of events from E such that

$\forall$ (i, j in the index set of t) $\bullet$ $i < j \Rightarrow \neg(t(j) \leq t(i))$ and
$\bigcup t(i) = E$
where $t(i)$ denotes the ith event of the trace.

Definition 7: (Set of All Traces of a Poset). The set of all traces of a poset p is defined as:

$tr(p) = \{ t \mid t \text{ is a trace of } p \}$.

Definition 8: (Set of All Traces of a Set of Posets). The set of all traces of a set of posets P is defined as:

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

Using the set of all traces as a basis, we can define refinement among two sets of posets through trace inclusion.

Definition 9: (Refinement). A set of posets Q is a refinement of a set of posets P if, and only if

$$Tr(Q) \subseteq Tr(P)$$

The refining specification is more restricted (in terms of possible orderings of events) than the refined specification. Or, in other words, the refining specification has less partial orders than the refined specification. In Section 4.4 we will re-use the definition of refinement to specify a satisfiability relation between two task model or use case specifications.

4.3.2 Mapping GTE Specifications to Sets of Posets

This section specifies how a generic task expression is mapped into a corresponding set of posets. For this purpose we define a (compositional) semantic function in the common denotational style. As given in Definition 10, an atomic generic task expression (denoted by α) is mapped into a set containing a single poset, which in turn consists of a single element only. Composite task expressions are represented by sets of posets, which are composed using the composition operators, defined in the previous section.

Definition 10: Let t, t_1, t_2 be abstract task expressions, then the mapping to sets of partial order sets is defined as follows:

$$\mathcal{M}[\![\alpha]\!] = \{\alpha_{poset}\}$$
$$\mathcal{M}[\![t_1 \ _{>>} \ t_2]\!] = \mathcal{M}[\![t_1]\!] \,.\, \mathcal{M}[\![t_2]\!]$$
$$\mathcal{M}[\![t_1 \ _|| \ t_2]\!] = \mathcal{M}[\![t_1]\!] \,||\, \mathcal{M}[\![t_2]\!]$$
$$\mathcal{M}[\![t_1 \ _[] \ t_2]\!] = \mathcal{M}[\![t_1]\!] \,\#\, \mathcal{M}[\![t_2]\!]$$
$$\mathcal{M}[\![t^{Opt}]\!] = \mathcal{M}[\![t]\!] \,\#\, \{\varnothing_{poset}\}$$
$$\mathcal{M}[\![t^{Rep}]\!] = \mathcal{M}[\![t]\!]^{*}_{\#}$$

In what follows we illustrate the application of the semantic function by applying it to the “Login” generic task expression of the previous section. The overall application of

$$\mathcal{M}(t_1 \ _{>>} \ t_2 \ _{>>} \ t_3 \ _{>>} \ t_4 \ _{>>} \ (t_5 \ _[] \ t_6)$$

can be further decomposed, by successively applying the definition of _>> and _[]. As a result, we obtain the following expression:

$$\mathcal{M}(t_1).\ \mathcal{M}(t_2).\ \mathcal{M}(t_3).\ \mathcal{M}(t_4).(\ \mathcal{M}(t_5) \# \mathcal{M}(t_6)).$$

By mapping the atomic tasks into the corresponding sets of posets and by performing the required set compositions we obtain the following:

$$\{(\{t_1, t_2, t_3, t_4, t_5\}, \{(t_1, t_2), (t_2, t_3), (t_3, t_4), (t_4, t_5)\}^*),$$
$$(\{t_1, t_2, t_3, t_4, t_6\}, \{(t_1, t_2), (t_2, t_3), (t_3, t_4), (t_4, t_6)\}^*)\}$$

The first poset denotes the scenario of a successful login and the second poset represents a scenario of login failure.

4.3.3 Transforming UC-LTS to Sets of Posets

In this section we demonstrate how UC-LTS specifications (as defined in Section 4.1) are mapped into the common semantic model. For this purpose we have devised an algorithm that generates a set of posets from a given UC-LTS specification. Table 4 gives the corresponding pseudo code. We 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 [29]. However, as described in the following, instead of step-wise composition of regular expressions, we compose sets of posets.

Table 4. Algorithm Transforming a UC-LTS to a Set of Posets

```
(1)   var tt:SPOSET[][] with all array elements initialized to {∅_sposet}
      for each transition (q_s, X, q_e) in T do
           tt[q_s, q_e] := {(X, id X)}  where id X = {(l, l) | l ∈ X)}
      od
-------------------------------------------------------------------------
(2)   for each state q_i in Q – (F ∪ {q_0}) do
-------------------------------------------------------------------------
(3)        for each pair of states q_n and q_k with n ≠ i & k ≠ i and
                X, Y ∈ 2^S such that (q_n, X, q_i) ∈ T and (q_i, Y, q_k) ∈ T  do
-------------------------------------------------------------------------
(4)             var tmp:SPOSET
                tmp := tt[q_n, q_i] . tt[q_i, q_i]*_# . tt[q_i, q_k]
-------------------------------------------------------------------------
(5)             if ∃ V ∈ 2^S such that (q_n, V, q_k) ∈ T then
                     tmp := tmp # tt[q_n, q_k]
                endif
-------------------------------------------------------------------------
(6)             T := T ∪ { (q_n, ∅, q_k) }
-------------------------------------------------------------------------
(7)             tt[q_n, q_k] := tmp
           od
-------------------------------------------------------------------------
(8)        Q = Q – {q_i}
      od
-------------------------------------------------------------------------
(9)   var result:SPOSET := ∅
      for each q_f in F do
           if result = ∅ then
                result := tt[q_0, q_f]
           else
                result := result # tt[q_0, q_f]
           endif
      od
-------------------------------------------------------------------------
(10)  if ∃ W ∈ 2^S such that (q_0, W, q_0) ∈ T then
           result := result # tt[q_0, q_0]
      endif
-------------------------------------------------------------------------
      return result
```

The procedure starts (1) with the creation of the transition table (a two-dimensional array (‘tt’)) populated with all transitions of the given UC-LTS specification. Indexed by a source and a target state a table cell contains a set of posets constructed from the label(s) associated to the representative transition. In most cases the set of posets will contain a single poset, which in turn consists of a single element representing *one* use case step. Only, if multiple labels were associated with the transition, 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.

The core part of the algorithm consists of two nested loops. The outer loop (2) iterates through all states of the 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, we perform the following: Compute (and temporarily store) the sequential composition of the following three sets of posets (4):

1. Set of posets associated to the incoming transition
2. Result of the *iterated alternative sequential composition* (Definition 5.2) of the poset 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 $\emptyset_{sposet}$.
3. Set of posets associated to the outgoing transition.

Next we examine whether there exists a transition from the source state of the incoming transition to the target state of the outgoing transition. If yes (5), the temporary stored set of posets is overwritten by the choice composition of the set of posets denoted by the found existing transition and the former "value" of the temporary store. Then (6) we add a new transition from the source state of the incoming transition to the target state of the outgoing transition. In addition (7) we populate the corresponding cell in the transition table with the temporary stored set of posets.

Back in the outer loop, we eliminate (8) the currently visited state from the UC-LTS and proceed with the next state. Once the 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 (9) we perform a choice composition of the sets of posets indexed by all the transitions from the initial state to a final state. Second, if the initial state additionally contains a self loop (10) then we add the set of posets denoted by that self loop to the before-mentioned choice composition.

If we apply our algorithm to the example "Login" UC-LTS of section 4.1 we obtain the following set of posets:

$\{$
$(\{s_1, s_{21}, s_{22}, s_3, s_4, s_5, s_6\}, \{(s_1, s_{21}), (s_1, s_{22}), (s_{21}, s_3), (s_{22}, s_3), (s_3, s_4), (s_4, s_5), (s_5, s_6)\}^*),$
$(\{s_1, s_{21}, s_{22}, s_3, s_4, s_{4a1}, s_{4a2}\}, \{(s_1, s_{21}), (s_1, s_{22}), (s_{21}, s_3), (s_{22}, s_3), (s_3, s_4), (s_4, s_{4a1}), (s_{4a1}, s_{4a2})\}^*)$
$\}$

The first poset represents the main success scenario in the original "Login" use case whereas the second poset represents the scenario where extension 4a ("The provided username or/and password is/are invalid") is taken. We note that the events e_{21} and e_{22} are not related by the partial order relation. Hence, a valid trace (see Definition 6) can contain e_{21} and e_{22} in any order. This correlates to the original use case specification where the primary actor may perform step 2.1 and step 2.2 in arbitrary order.

4.4 Satisfiability between Use Cases and Task Models

The common semantic domain defined in the previous sections is the essential basis for the formal definition of a satisfiability relation between two specifications. Such a notion of satisfiability applies equally well between artifacts of a similar nature (e.g. two use cases) as it does between use cases and task models. Our definition of satisfiability is as follows: *A specification 'X' satisfies another specification 'Y' if every scenario of 'X' is also a valid scenario of 'Y'.*

Within our semantic framework, a scenario of a use case or task model corresponds to a trace (Definition 6) in the corresponding set of posets. Hence a task model or use case specification satisfies another specification if the set of all traces (Definition 8) of the former is a subset of the set of all traces of the latter. One precondition for the application of the definition is that both sets of posets are based on the same event 'alphabet'. This can be achieved by renaming the events of the refined specification to their corresponding counterparts in the refining specification. Moreover, if a task model specification is compared with a use case specification, all events representing internal use case steps need to be removed. As pointed out in Section 2.3 task models focus on aspects that are relevant for UI design and as such abstract from internal system interactions.

For illustration purposes, we will formally determine whether the specification of the "Login" task model satisfies the specification of the "Login" use case. As a first step we need to unify the event alphabets. In the case of the "Login" use case steps 4, 4a1 and 6 represent internal (UI irrelevant) system interactions and hence are to be deleted. Moreover, the events representing use case steps must be renamed after the events representing the corresponding tasks in the task model.

Table 5. Mappings of Disabling and Suspend/Resume into GTE.

Set of Posets representing "Login" UC (after Event Mapping)
$\{(\{t_1, t_2, t_3, t_4, t_5\}, \{(t_1, t_2), (t_1, t_3), (t_2, t_4), (t_3, t_4), (t_4, t_5)\}^*),$ $(\{t_1, t_2, t_3, t_4, t_6\}, \{(t_1, t_2), (t_1, t_3), (t_2, t_4), (t_3, t_4), (t_4, t_6)\}^*)\}$
Set of Posets representing the "Login" Task Model
$\{(\{t_1, t_2, t_3, t_4, t_5\}, \{(t_1, t_2), (t_2, t_3), (t_3, t_4), (t_4, t_5)\}^*),$ $(\{t_1, t_2, t_3, t_4, t_6\}, \{(t_1, t_2), (t_2, t_3), (t_3, t_4), (t_4, t_6)\}^*)\}$

As depicted by Table 5, it can be easily seen that every trace of the set of posets representing the task model is also a trace of the set of posets (after the event mapping) of the use case. Hence, according to the definition above, we can conclude that the "Login" task model satisfies the "Login" use case.

5 Conclusion and Future Work

In this paper we have presented a common semantic framework for use cases and task models. The main motivation behind our research is the need for an integrated development methodology where task models are developed as logical progressions

from use case specifications. Our semantic framework is based on a two-step mapping from a particular use case or task model notation to the common semantic domain of *sets of partial order sets*. We argue that 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, respectively. In particular we defined a Use Case Labeled Transition System as an intermediate semantic domain for use cases. It was demonstrated that UC-LTS allow for a natural representation of the order in which actions are to be performed. In the case of task models we defined generic task expressions (GTE) as an intermediate semantic domain. Similar to tasks, a generic task expression is hierarchically composed of sub-task expressions using a set of standard operators. Hence the mapping from a concrete task model to GTE remains straightforward and intuitive. In order to (partially) validate our approach we used the framework to define a semantics for CTT task models, including complex operators such as "disabling" and "suspend/resume". We also demonstrated how DSRG-style use cases are mapped into a set of partially order sets. Finally we used our semantic framework to provide a formal definition of satisfiability between use case and task model specifications. According to the definition, a use case or task model specification satisfies another specification if every scenario of the former is also a valid scenario of the latter.

Thus far, we concentrated on capturing sets of usage scenarios. As future work, we are aiming at further extending our semantic framework. One such extension is the introduction of different event types. The main motivation for such an extension is that in task modeling (e.g. CTT), one often distinguishes between different task *types*. Examples are: "data input", "data output", "editing", "modification", or "submit". As a consequence, rules to further restrict the definition of a valid trace may need to be defined. An example of such a rule may be the condition that an event of type "data input" must always be followed by a corresponding event of type "submit". Another extension of the semantic model deals with the capturing of state information. State information is often employed in a use case to express and evaluate conditions. For example the pre-condition 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 means to capture the notion of the state and should be able to map state conditions to the appearance of events.

Further avenues deal with the extension of the proposed definition of a satisfiability relation for use case and task model specifications. Such an extended definition may take into account different event types and the refinement of state conditions. Moreover, we envision that refinements, and proofs of satisfiability, can ideally be aided by tools, supporting the verification. We are currently investigating how our approach can be translated into the specification languages of existing model checkers and theorem provers.

Acknowledgments. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) in the form of a Postgraduate Scholarship for D. Sinnig and Discovery Grants for P. Chalin and F. Khendek.

References

1. Seffah, A., Desmarais, M. C., Metzger, M., Software and Usability Engineering: Prevalent Myths, Obstacles and Integration Avenues, chapter in *Human-Centered Software Engineering—Integrating Usability in the Software Development Lifecycle*, Springer.
2. Cockburn, A., Writing effective use cases, Addison-Wesley, Boston, 2001.
3. Pressman, R. S., Software engineering: a practitioner's approach, McGraw-Hill, Boston, Mass., 2005.
4. Paternò, F., Model-Based Design and Evaluation of Interactive Applications, Springer, 2000.
5. Larman, C., Applying UML and patterns : an introduction to object-oriented analysis and design and the unified process, Prentice Hall PTR, Upper Saddle River, NJ, 2002.
6. Butler, G., Grogono, P., Khendek, F., A Z Specification of Use Cases. In *Proceedings of APSEC 1998*, pp. 94-101, 1998.
7. Grieskamp, W., Lepper, M., Schulte, W., Tillman, N., Testable use cases in the abstract state machine language. In *Proc. APAQS'01*, Asia-Pacific Conference on Quality Software, 2001.
8. Barnett, M., Grieskamp, W., Schulte, W., Tillmann, N., Veanes, M., Validating Use Cases with the AsmL Test Tool in *Proceedings of QSIC 2003* (Third International Conference on Quality Software), November 2003.
9. XSLT, XSL Transformations Version 2.0 [Internet], Available from http://www.w3.org/TR/xslt20/. Accessed: Dec. 2006. Last Update: Nov. 2006.
10. 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.
11. Dittmar, A., Forbrig, P., Higher-Order Task Models, in *Proceedings of Design, Specification and Verification of Interactive Systems 2003*, pp. 187-202, Funchal, Madeira Island, Portugal.
12. Souchon, N., Limbourg, Q., Vanderdonckt, J., Task Modelling in Multiple contexts of Use, in *Proceedings of Design, Specification and Verification of Interactive Systems*, Rostock, Germany, pp. 59-73, 2002.
13. Card, S., Moran, T. P., Newell, A., "The Psychology of Human Computer Interaction", 1983.
14. Dittmar, A., Forbrig, P., Stoiber, S., Stary, C., Tool Support for Task Modelling - A Constructive Exploration, in Proceedings of *Design, Specification and Verification of Interactive Systems* 2004, July 2004.
15. Johnson, P., Johnson, H., Waddington, R., Shouls, A., Task Related Knowledge Structures: Analysis, Modelling and Application. Jones, D.M. and Winder, R. (Eds). People and Computers IV, Manchester, Cambridge University Press. pp. 35-62, 1988.
16. Sinnig, D., Chalin, P., Khendek, F., Consistency between Task Models and Use Cases, to Appear in Proceedings of *Design, Specification and Verification of Interactive Systems*, Salamanca, Spain, March 2007.
17. Mauw, S., Reniers, M. A., An Algebraic Semantic of Basic Message Sequence Charts, in Computer Journal, 37, 1994.
18. ITU-T, Recommendation Z.120- Message Sequence Charts, Geneva, 1996.
19. Xu, J., Yu, W., Rui, K., Butler, G., Use Case Refactoring: A Tool and a Case Study, in Proceedings of APSEC 2004, Busan, Korea, pp. 484-491, 2004.
20. Baeten, J. C. M., Weijland, W. P., Process algebra, Cambridge University Press, New York, NY, USA, 1990.
21. ITU-T, Recommendation Z.120- Message Sequence Charts, Geneva, 1999.

22. Zheng, T., Khendek, F., Time consistency of MSC-2000 specifications, in Computer Networks, Vol. 42, No. 3, Elsevier, June 2003.
23. Alur, R., Holzmann, G. J., Peled, D., An Analyzer for Message Sequence Charts, in Software - Concepts and Tools, 17, pp. 70-77, 1996.
24. Katoen, J. P., Lambert, L., Pomsets for Message Sequence Charts, in *Proceedings of FBT-VS 1998*, Cottbus, Germany, Shaker Verlag, pp. 197-207, 1998.
25. Mizouni, R., Salah, A., Dssouli, R., Parreaux, B., Integrating Scenarios with Explicit Loops, in *Proceedings of NOTERE*, 2004, Essaidia Morocco, 2004.
26. Paternò F., Santoro, C., The ConcurTaskTrees Notation for Task Modelling, Technical Report at CNUCE-C.N.R., May, 2001.
27. Sinnig, D., Chalin, P., Khendek, F., Towards a Common Semantic Foundation for Use Cases and Task Models, to appear in *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2007.
28. Brinksma, E., Scollo, G., Steenbergen, C., LOTOS specifications, their implementations, and their tests, in *Proceedings of IFIP Workshop Protocol Specification, Testing, and Verification VI*, pp. 349-360, 1987.
29. Linz, P., An introduction to formal languages and automata, Jones and Bartlett Publishers, Sudbury, Mass., 1997.