Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 183 (2007) 73–88

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Towards a Common Semantic Foundation for Use Cases and Task Models

Daniel Sinnig, [a,1] [,2] Patrice Chalin[a,3] and Ferhat Khendek[b,4]

[a] *Department of Computer Science and Software Engineering*
Concordia University
Montreal, Canada

[b] *Department of Electrical and Computer Engineering*
Concordia University
Montreal, Canada

Abstract

Use cases are the notation of choice for functional requirements documentation, whereas task models are used as a starting point for user interface design. In this paper, we motivate the need for an integrated development methodology in order to narrow the conceptual gap that exists between software engineering and user interface design. A prerequisite is the definition of a common semantic framework. With respect to the definition of a suitable semantic domain, we discuss core requirements and review related work. A preliminary approach based on (sets of) partially ordered sets is presented. A mapping from CTT task models and use case graphs to the before-mentioned formalism is proposed.

Keywords: Use Cases, Task Models, Scenarios, Semantics, Posets

1 Introduction

Unfortunately in current practice, functional requirements specification and UI design are neither harmonized nor coordinated. Instead of having a unique process, where UI design follows as a logical progression from functional requirements specification, both entities are treated rather independently. In particular, it has been noted that most UI design methods are not very well integrated with standard software engineering practices [23]. In fact, UI design and the engineering of functional requirements are often carried out by different people following different processes.

[1] This work is partially supported by the National Sciences and Engineering Research Council of Canada.

[2] Email: d_sinnig@encs.concordia.ca

[3] Email: chalin@encs.concordia.ca

[4] Email: khendek@ece.concordia.ca

doi:10.1016/j.entcs.2007.01.062

There exists a relatively large conceptual gap between software engineering and UI development with both disciplines having their own models, lifecycles and theories. The following two issues follow directly as a result of this lack of integration:

- Developing UI-related models and software engineering models independently neglects existing overlaps, gives rise to redundancies and increases the maintenance overhead.
- Deriving the implementation from UI-related models and software engineering models towards the end of the lifecycle is problematic as both processes do not commence from the same specification and thus may result in inconsistent designs.

In this paper, we present preliminary results from an ongoing research project which has as a main goal: bridging the conceptual gap between software engineering and UI design by formally integrating use cases and task models. While use cases are the method of choice for the purpose of functional requirements documentation [4], UI design typically starts with the identification of user tasks, and environmental requirements [20]. None-the-less, use cases and task models share many similarities. We demonstrate this (in part) by the presentation of a common semantic framework for both models.

The remainder of this paper is structured as follows. Section 2 gives an informal introduction to our framework. In Section 3 we review and compare use cases and task models. Section 4 discusses related work with respect to the definition of semantics of scenario-based notations. It is also in this section that we outline core requirements for a common semantic model and present our approach. We conclude and provide an outlook of future work in Section 5.

2 Overall Framework

Our overall research goal has been to define an integrated *methodology* for the development of use cases and task models within an overall software engineering process. **A key** objective of this initiative is the definition of a formal framework for handling use cases and task models at the requirements and design levels. The cornerstone for such a formal framework is a common semantic domain for both notations.

The common semantic domain is the essential basis for the formal definition of a *satisfiability* relation. Such a relation allows us to make (formal) semantic links between successive refinements of use cases and task models. Refinements, and proofs of *satisfiability*, would ideally be aided by tools, supporting the verification. Such tools typically follow two main approaches: Automatic verification (i.e. model-checking, automatic theorem proving) and manual verification (interactive theorem proving). But even without tools, an informal application of the *satisfiability* relation can serve as a rigorous basis for identifying simple traceability links (as is commonly done in software engineering) among the artifacts. Figure 1 visualizes our idea of having a general notion of *satisfiability* that applies equally well between artifacts of a similar nature (e.g. two use cases) as it does between

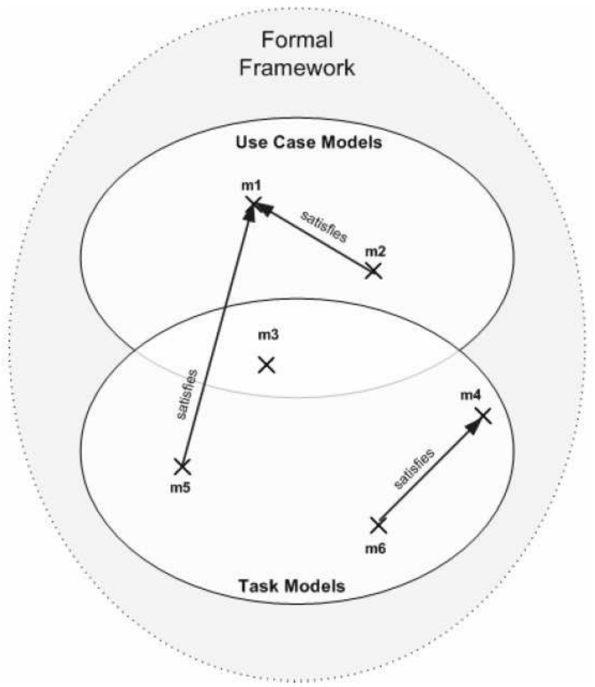


Fig. 1. Relating Use Cases and Task Models within a Formal Framework

use cases and task models.

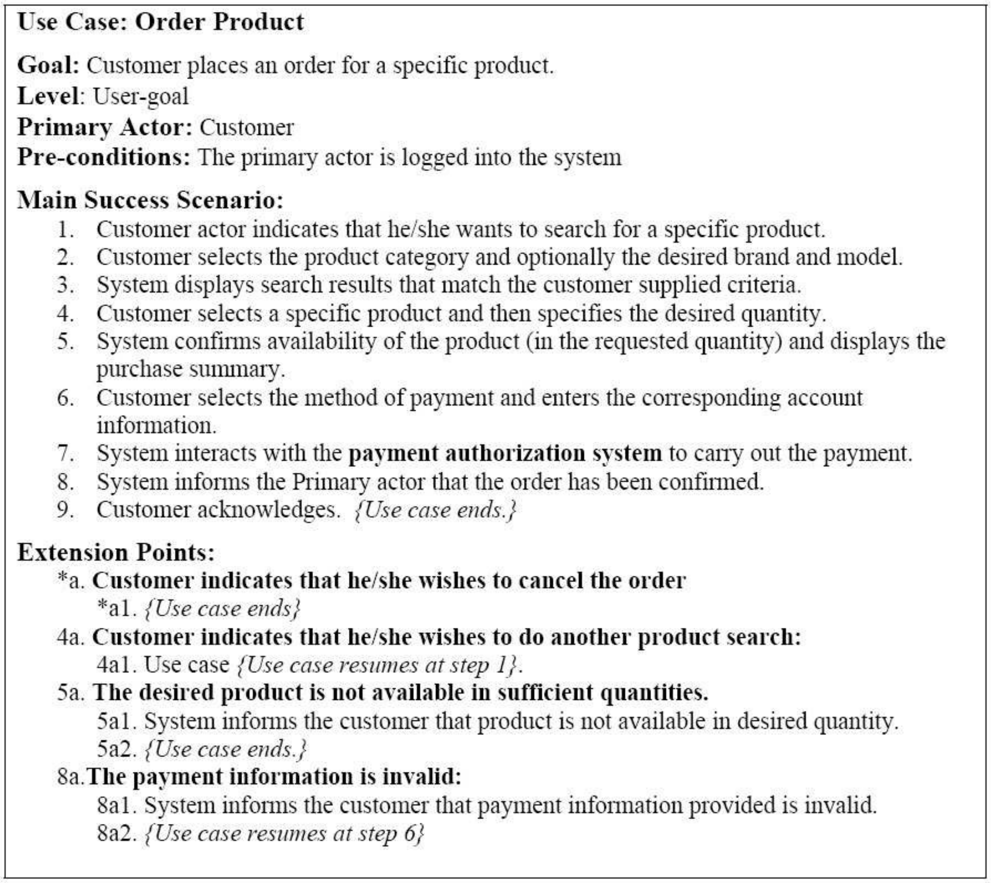
3 Background

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

3.1 Use Case Models

Use cases were introduced roughly 15 years ago by Jacobson. He defined a use case as a *"specific way of using the system by using some part of the functionality"* [8]. More recent popularization of use cases is often attributed by Cockburn [4]. Use case modeling is gradually making its way into mainstream practice which sees it as a key activity in its software development process (e.g. Rational Unified Process) and as a result, there is accumulating evidence of significant benefits to customers and developers [14].

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 [11]. 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



Use Case: Order Product

Goal: Customer places an order for a specific product.
Level: User-goal
Primary Actor: Customer
Pre-conditions: The primary actor is logged into the system

Main Success Scenario:

1. Customer actor indicates that he/she wants to search for a specific product.
2. Customer selects the product category and optionally the desired brand and model.
3. System displays search results that match the customer supplied criteria.
4. Customer selects a specific product and then specifies the desired quantity.
5. System confirms availability of the product (in the requested quantity) and displays the purchase summary.
6. Customer selects the method of payment and enters the corresponding account information.
7. System interacts with the **payment authorization system** to carry out the payment.
8. System informs the Primary actor that the order has been confirmed.
9. Customer acknowledges. *{Use case ends.}*

Extension Points:

*a. **Customer indicates that he/she wishes to cancel the order**
  *a1. *{Use case ends}*
4a. **Customer indicates that he/she wishes to do another product search:**
  4a1. Use case *{Use case resumes at step 1}*.
5a. **The desired product is not available in sufficient quantities.**
  5a1. System informs the customer that product is not available in desired quantity.
  5a2. *{Use case ends.}*
8a.**The payment information is invalid:**
  8a1. System informs the customer that payment information provided is invalid.
  8a2. *{Use case resumes at step 6}*

Fig. 2. Example Use Case for "Order Product"

a contract (of the envisioned system behavior) between stakeholders [4]. In current practice, use cases are promoted as structured textual constructs written in prose language. While the use of narrative languages makes use cases 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.

As a concrete example of a use case, Figure 2 presents a detailed user-goal level use case for "Ordering a Product". A 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. The main success scenario consists of a sequence of interaction steps between the user and the system. The interaction steps indicate the most common way in which the primary actor can reach his/her goal by using the system.

The 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 alternative (and sometimes exceptional) 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

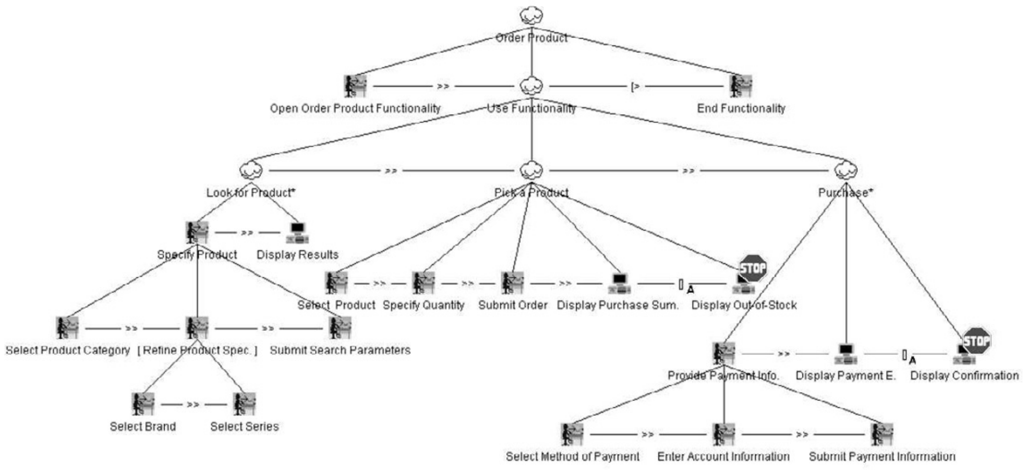


Fig. 3. "Order Product" Task Model

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 requirement elicitation device.

3.2 *Task Models*

User task modeling is by now a well understood technique supporting user-centered UI design [18]. In most UI development approaches, the task set is the primary input to the UI design stage. User 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 [5].

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 [24]. Different presentations of task models exist, ranging from narrative task descriptions, work flow diagrams to formal hierarchical task descriptions.

Figure 3 shows an adapted ConcurTaskTreesEnvironment (CTTE) [16] visualization of the user task model. CTTE is a tool for graphical modeling and analyzing ConcurTaskTrees (CTT) models [17]. The figure illustrates the hierarchical break down and the temporal relationships between tasks involved in the "Order Product" functionality (depicted in the use case of Section 3.1). More precisely the depicted 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. An indication of task types is given by the used symbol to represent tasks. The task model is organized as a directed graph. Tasks are hierarchically decomposed into sub-tasks and atomic actions. Leaf tasks are also called actions, since they are the task that actually carried out by the user or the system. The execution order of tasks is determined by temporal operators that are defined between peer tasks. Various temporal operators exist; the most popular are: *enabling* ($>>$), *choice* ($[]$), *concurrency* ($||$), and *disabling* ($[>$). A complete list of the CTT operators together with definition of their interpretation can be found in [17].

We note that the binary temporal operator used between the tasks "Display Purchase Summary" and "Display out of Stock" and between the tasks "Display Confirmation" and "Display Payment Error" is not part of CTT. We have introduced the operator as an extension to CTT. It is called the *Abort Choice* operator and is represented by the symbol ($[]_A$) and the *STOP* sign hovering over its right operand. The interpretation of the *Abort Choice* operator is similar to the build-in *Choice* operator, in the sense that either the task specified by the first operand or the task specified by the second operand is executed. However, after the execution of the second operator all tasks of the model become disabled. Hence, no more tasks can be executed and the scenario ends.

Main motivation for the introduction of this temporal operator was the fact that without it we were not able to conveniently implement the flow specified in the "Order Product" use case as a CTT task model. Particularly problematic are the use case steps which prematurely lead to termination. The only way to simulate such an effect in a CTT task model is to create several main alternative branches in the task tree. One branch represents the case when the "Order Product" is completely performed, whereas the other branches represent cases when the task terminates prematurely. Such a modeling however, creates a significant amount of duplication (since identical starting tasks would be repeated in each brand) and unnecessarily increases the complexity of the visualization of the task tree.

3.3 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 approaches 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. Taking this perspective, 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 [4]) in the context of a use case.

The above mentioned difference is manifest in the use case and task model provided as examples. Compared to the "Order Product" use case the corresponding "Order Product" task model has more UI details as it contains steps that are pertinent to a graphical UI. For example the task model contains additional tasks which deal with the submission of selected or entered values (e.g. "Submit Search Parameters" or "Submit Payment Information"). These steps are not specified in the corresponding use case, as they are geared to a UI which requires an extra submission step as a confirmation for a data input. In addition, some of the use case steps (which are the smallest possible units of a use case) have been split into even smaller action tasks in the task model. For example, use case step 2 corresponds to one connected user activity, which however needs to be supported by three UI elements capturing the input of the Product Category, Series and Brand.

In many cases however, a use case will contain (behavioral) information that is not present in the task model. 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. Interactions with secondary actors (which are specified in the use case model) are omitted since they are irrelevant for UI design. An example of this is use case step 7 ("System interacts with the payment authorization system to carry out the payment") of the "Order Product" use case of Figure 2.

4 Semantic Domains for Use Cases and Task Models

In this section we begin with a review of formalism used for scenario-based notations, and thus, those most likely to serve as a common foundation for use cases and task models. This is followed by a discussion of the *requirements* that would need to be addressed by a common semantic framework for use cases and task models. Finally, we present our proposed semantic domain, which is based on partial order sets.

4.1 Related work

Within the domain of scenario-based notations the behavioral aspects of a system (capturing the ordering and the progression of events) play a pivotal role. While several different formalisms have been proposed for scenario-based notations, in what follows we briefly discuss three prominent approaches, namely: process algebras, partial order sets and graph structures.

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 (which defines terms of algebra) with associated "deduction/transformation" rules for reasoning about algebraic specifications. 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) [2][6]. This work is a continuation of preliminary research first established by Mauw et. Reniers [13]. In more recent work, Rui and Butler also suggest a process algebraic semantics for use case models, with the overall goal of formalizing use case refactoring [22][25]. In their approach, scenarios are represented as basic MSCs–as suggested by [21]. 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.

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 [7] based on timed labeled partial order sets (lposets) [26]. Partial order semantics for (regular, un-timed) MSCs have been proposed by Alur [1] and Katoen and Lambert [9]. Alur et. al. propose a semantics for a subset of MSCs which only allow message events as possible MSC events types. In contrast, the semantics of Katoen and Lambert is more complete. They map MSCs to a set of partial order multi-sets (pomsets). A pomset is a so-called isomorphic class of a corresponding lposet. A pomset contains all objects that can be derived by a bijective projection from a base lposet. Approaches based on pomsets are very similar to approaches based on lposets.

Mizouni et. al. propose use case graphs as an intermediate notation for use cases [15]. Use case graphs are directed, potentially cyclic graphs whose edges represent use case actions and nodes represent system states. This allows for a natural representation of the order in which actions are to be performed. In order to integrate several use cases into a single specification, Mizouni et. al. describe an algorithm for transforming a set of (related) use case graphs (each representing one use case) into an extended finite state machine (EFSM). The merging of the graphs is done on the basis of common states within the use case specifications.

The semantic definition proposed in this paper was originally inspired by the lposet approach proposed by Zheng et. al. In addition, similar to the work of Mizouni et. al., we employ use case graphs as an intermediate notation for use cases. Before we present our approach, we discuss some of the core requirements that need to be addressed by any formalism that is to be used to model both use cases and task models.

4.2 Requirements for a Semantic Framework

In Section 3 we reviewed key characteristics of use cases and task models and discussed their current (and specialized) areas of application. In this section, we will re-consider this information in order to compile a set of requirements that would be particular to any common semantic framework for use cases and task models. Both notations are used to specify scenarios that indicate how the system is used. Technically a scenario consists of a, possibly infinite, sequence of events. Therefore, we require that a semantic model for use cases and task models formally **captures sets of usage scenarios**. It should be possible to mechanically extract valid usage scenarios from formal specifications. Also, given a specification and a scenario, it should be possible to unambiguously decide whether the scenario is valid or not, relative to the given specification.

In task modeling (e.g. CTT), one often distinguishes between different task types. Examples are: "data input", "data output", "editing", "modification", or "submit". In the corresponding semantic model events should be **distinguishable by their types** as well. Based on the typing, the sequencing of events may be further constrained. An example of such a constraint is that an event representing the entry of information ("data input") must precede an event of submitting the very same data ("submit"). Of the formalisms we surveyed in Section 4.1, the approach based on labeled partial order sets formalism also distinguishes between different types of events that can occur during a run of a MSC. (In particular, it is the purpose of the labeling function to assign a type to each MSC event.)

In use case modeling, state conditions often constrain the execution of use case steps. For example the pre-condition attribute 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.

So far we have bound the requirements for the semantic model to the intrinsic characteristics of use cases and task models. The next requirement, however, is more tightly related to the software development process within which use cases and task models are to be crafted. One view of a software development process is as a series of "disciplines" during which models are iteratively transformed/refined until an implementation level has been reached. Use case and task modeling are part of such a lifecycle. Therefore, a common semantic model should easily **support refinement**.

This last requirement is directly related to one's choice of concurrency models. In interleaving models, the concept of true concurrency is omitted and concurrent system behavior is said to be equivalent to the non-deterministic choice of all possible (interleaved) sequential executions. As it turns out, interleaving approaches typically do not support arbitrary refinement of events (or actions) into sub-events (or sub-actions). The main reason is that, in an interleaving model, *"exactly what is interleaved depends on which events of a process one takes to be atomic"* [19]. Therefore, if a formerly atomic action is further refined, new interleavings among the sub-actions are introduced, which were not taken into account prior the refinement. Hence most of the equivalence relations (e.g. trace equivalence and bissimulation equivalence) are not preserved under arbitrary refinement [3]. This problem does not occur in non-interleaving concurrency models (also referred to as partial order semantics or true concurrency semantics) because the concept of concurrency is fundamental. System behavior is represented in terms of causally inter-related events based on a partial order relation. Events, that are not causally related, are interpreted as concurrent.

4.3 Semantic Domain Based on Sets of Posets

In this section, we illustrate an approach to semantics in which we demonstrate how CTT task models and use cases can be mapped to sets of partially ordered sets. We start by reiterating the definition of a partially ordered set (poset) and then define some operators over posets. Finally, we will describe semantics functions that will define a mapping from use cases and task models into sets of posets.

4.3.1 Mathematical Preliminaries (and Notation)

Definition 4.1 For our purposes, 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 casual order of events.

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

Definition 4.2 The binary operations: sequential composition (.) and parallel composition (||) of two posets p and q are defined as next. Note that R^* denotes the reflexive, transitive closure of R.

Let $p = (E_p, \leq_p)$ and $q = (E_q, \leq_q)$ with $E_p \cap E_q = \emptyset$ then:

$p.q = (E_p \cup E_q, (\leq_p \cup \leq_q \cup \{(e_p, e_q) |\ e_p \in E_p \textit{ and } e_q \in E_q\})^*)$
$p||q = (E_p \cup E_q, \leq_p \cup \leq_q)$

In our approach we define semantics for use cases and task models using the following operations over sets of posets.

Definition 4.3 For two sets of posets P and Q, sequential composition (.), parallel composition (||), and alternative composition (#) are defined as follows:

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

Also fundamental to our model is the notion of a trace. Next we define the set of traces for a given poset, and for a given set of posets.

Definition 4.4 A *trace* t of a poset $p = (E, \leq)$ is defined as a (possibly infinite) sequence of events from E such that

$\forall (i, j \text{ in the index set of } t).\ i < j \implies \neg(t(j) \leq t(i))$ and
$\bigcup t(i) = E$
where $t(i)$ denotes the i^{th} event of the trace.

Definition 4.5 The set of all traces of a poset p is defined as $tr(p) = \{t \mid t \text{ is a trace of } p\}$.

Definition 4.6 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 4.7 A set of posets Q is a refinement of a set of posets P if, and only if:
$Tr(Q) \subseteq Tr(P)$

4.3.2 Mapping CTT Task Models to Sets of Posets

We now briefly outline how CTT task models can be mapped to sets of posets. The mapping process consists of two steps: (1) conversion of a CTT task tree into a task expression; (2) application of a mapping function that relates the task expression to a corresponding set of posets.

In order to derive a task expression from the task model we first create a corresponding expression tree. In general, an expression tree is a tree whose leaves are operands and whose inner nodes are operators. In this case, the operands of the expression tree are actions (tasks at the leaf-level) and the operators are the temporal relations defined in CTT. In CTT, all temporal relations are defined as either binary operators (e.g. *enabling*, *disabling*) or unary operators (i.e. *iteration*, *option*). Hence in the expression tree all inner nodes have between one and two children. Since the conversion of trees to expressions is fairly conventional, it will not be described any further.

The next step consists of mapping that task expression into a corresponding set of posets. Action tasks correspond to the elements of the poset. Composite tasks are represented by sets of posets, which have been composed using the composition operators, defined in Section 4.3.1. Our (compositional) semantic function is defined in the common denotational style.

Definition 4.8 The semantic function $\mathcal{M}$ is inductively defined over the possible terms within CTT task expressions, with the following interpretations:
$\mathcal{M}[\![t]\!] = \{(\{t\}, \{(t,t)\})\}$ *//atomic action*
$\mathcal{M}[\![t_1 >> t_2]\!] = \mathcal{M}[\![t_1]\!] \;.\; \mathcal{M}[\![t_2]\!]$ *//enabling*
$\mathcal{M}[\![t_1 \mid\mid t_2]\!] = \mathcal{M}[\![t_1]\!] \mid\mid \mathcal{M}[\![t_2]\!]$ *//concurrent execution*
$\mathcal{M}[\![t_1 \;[]\; t_2]\!] = \mathcal{M}[\![t_1]\!] \;\#\; \mathcal{M}[\![t_2]\!]$ *//choice*
$\mathcal{M}[\![t_1| + |t_2]\!] = (\mathcal{M}[\![t_1]\!] \;.\; \mathcal{M}[\![t_2]\!]) \;\#\; (\mathcal{M}[\![t_2]\!] \;.\; \mathcal{M}[\![t_1]\!])$ *//order independency*
$\mathcal{M}[\![t^{opt}]\!] = \mathcal{M}[\![t]\!] \;\#\; (\emptyset, \emptyset)$ *//optional execution*
$\mathcal{M}[\![t^*]\!] = \{(\emptyset,\emptyset), \mathcal{M}[\![t]\!], (\mathcal{M}[\![t]\!].\mathcal{M}[\![t]\!]), (\mathcal{M}[\![t]\!].\mathcal{M}[\![t]\!].\mathcal{M}[\![t]\!]), ...\}$ *//iteration*

Note that if the CTT task expression contains the temporal operators *Disabling* ([>), *Suspend/Resume* (|>) or the newly introduced operator *Abort Choice* ($[]_A$) it needs to be transformed into an equivalent task expression which does not involve

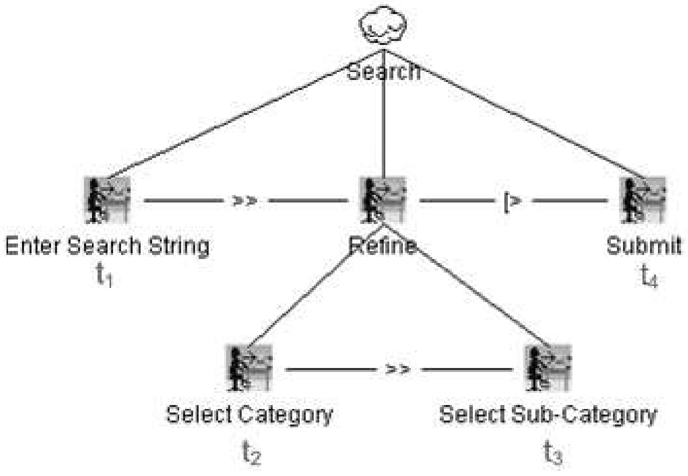


Fig. 4. “Search” Task

these operators, prior to the application of the mapping function.

A simple example illustrates how a task model of a “Search” task (illustrated in Figure 4) is mapped into a corresponding set of posets. In order to perform a search, the user first enters the search string. Next the user either directly submits the search parameter or further refines the search criteria. The “Refine” task consists of the sequential execution of the “Select Category” task and the “Select Sub-Category” task, and may be interrupted (and disabled) at any time by executing the “Submit” task. We employed the binary disabling ([>) operator to specify the desired behavior. The meaning of the 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 (in this case, the “Submit” task) is executed, the task specified by the first operand (in this case the “Refine” task) becomes disabled.

From the task model we can derive the following task expression:

$$t_1 >> ((t_2 >> t_3)[> t_4)$$

Note that the various tasks are represented by using the identifiers (t_1, t_2, t_3, t_4). Next we have to transform the task expression into an equivalent task expression, which does not make use of the *disabling* operator. This can be done by examining the set of possible scenarios that can be extracted from the specification. In our example we have the choice between the following three scenarios:

(i) The user enters and submits the search string $(t_1 >> t_4)$.

(ii) The user enters the search string, selects a category and then submits the search parameter $(t_1 >> t_2 >> t_4)$.

(iii) The user enters the search string and selects a category as well as a sub-category before submitting $(t_1 >> t_2 >> t_3 >> t_4)$.

Consequently the task expression can be rewritten as follows:

$$(t_1 >> t_4)[](t_1 >> t_2 >> t_4)[](t_1 >> t_2 >> t_3 >> t_4)$$

The task expression is now in elementary form and hence we can apply our semantic function $\mathcal{M}$. According to its recursive definition, the application can be broken down into the following steps:

$$\begin{aligned}
&\mathcal{M}[\![(t_1 >> t_4)[\,](t_1 >> t_2 >> t_4)[\,](t_1 >> t_2 >> t_3 >> t_4)]\!]\\
&= \mathcal{M}[\![t_1 >> t_4]\!] \; \# \; \mathcal{M}[\![t_1 >> t_2 >> t_4]\!] \; \# \; \mathcal{M}[\![t_1 >> t_2 >> t_3 >> t_4]\!]\\
&= \mathcal{M}[\![t_1]\!].\mathcal{M}[\![t_4]\!] \; \# \; \mathcal{M}[\![t_1]\!].\mathcal{M}[\![t_2]\!].\mathcal{M}[\![t_4]\!] \; \# \; \mathcal{M}[\![t_1]\!].\mathcal{M}[\![t_2]\!].\mathcal{M}[\![t_3]\!].\mathcal{M}[\![t_4]\!]\\
&= \{(\{t_1, t_4\}, \{(t_1, t_4)\}^*)\} \cup\\
&\quad \{(\{t_1, t_2, t_4\}, \{(t_1, t_2), (t_2, t_4)\}^*)\} \cup\\
&\quad \{(\{t_1, t_2, t_3, t_4\}, \{(t_1, t_2), (t_2, t_3), (t_3, t_4)\}^*)\}
\end{aligned}$$

As a result we obtain a set of three posets, where each poset represents one of the scenarios discussed before.

4.3.3 Transforming Use Cases to Sets of Posets

In this section we discuss how use cases can be transformed into sets of posets. The transformation consists of two parts. First the textual use case is transformed into an intermediate graph form, which we will refer to as the use case graph. Next, based on the use case graph a corresponding set of posets is iteratively constructed.

Definition 4.9 A use case graph is a labeled transition system

$U = (Q, q_0, q_f, T)$ *where,*

Q is a finite set of states
$q_0 \in Q$ is the initial state
$q_f \in Q$ is the final state
$T \subseteq Q \times Q$ is the transition relation.

Similar to the work of Mizouni et. al [15] (discussed in Section 4.1), the transitions of the labeled transition system represent use case steps, whereas the nodes represent states. The composition of the use case graph from the actual use case depends on the flow constructs, which are implicitly or explicitly entailed in the use case. Examples of such flow constructs are: jumps (e.g. *use case resumes at step X*), sequencing information (e.g. the numbering of use case steps), or branches to use case extensions. It is to be noted that if the use case is captured in purely narrative form the derivation of the use case graph will be a manual activity.

Based on the use case graph a set of posets is constructed. The construction can be performed mechanically using the following two steps: **First**, we assign a set of posets to each transition in the use case graph. Typically the set of posets consists of a single poset, which in turn defines a single event representing the execution of the corresponding use case step. **Second**, the use case graph is iteratively transformed into a labeled transition system that only consists of an initial state and a final state. With each iteration one node of the use case graph is eliminated and a new transition is defined between its incoming node(s) and its outgoing node(s). Similar to the first step, a set of posets is assigned to the newly inserted transition.

This set of posets is the result of the composition of the sets of posets attached to the incoming transition and the outgoing transition. At this point it no longer represents a single use case step, but a composition of use case steps. Special care must be taken if the eliminated node contains a self loop or if there already exists a transition from the incoming node to the outgoing node.

Once the graph consists of only the initial and the final state, the set of posets associated to the transition between the two states denotes the set of posets representing the original use case graph. We note that the main idea of the presented algorithm stems from the well-known algorithm that transforms a deterministic finite automaton (DFA) into an equivalent regular expression. However, instead of step-wise composition of regular expressions, we compose sets of posets. We refer the reader to [12] for more details.

In the next and final Section we conclude by summarizing the main ideas of this paper. The proposed semantic domain is related back to our enumerated requirements and an outlook to future avenues is given.

5 Conclusion and Future Work

In this paper we highlighted the need for an integrated methodology for developing use cases and task models. This methodology would rest upon a common semantic framework. In theory, both notations can be used to describe the same information. However, in practice, use cases are mainly employed to document functional requirements whereas task models are used to describe UI requirements and design decisions.

With respect to the definition of the semantic framework we reviewed related work and formalisms. Based on the intrinsic characteristics of use cases and task models, we compiled a list of four core requirements that should be met by any formal framework: (1) capture of sets of usage scenarios, (2) offer a distinction between different event types, (3) capture the notion of the state, and (4) support for event refinement. We then presented our initial approach which maps use cases and task models to partially ordered sets. Thus far, the poset formalism, as presented in this paper, only meets the first and the last requirement. Valid event sequences are specified by relating events using a partial order relation. A scenario is said to be valid if a trace can be extracted from the corresponding set of posets that resembles the event sequence of the scenario. Regarding the requirement of supporting event refinement, we used the poset formalism to specify non-interleaving semantics, which naturally support refinement [3].

As future work, we will be tackling the remaining two requirements. For example, the requirement of supporting different event types can be addressed by defining a labeling function, which maps an event type to each element of the poset. Additionally, rules to further restrict the definition of a valid trace need to be introduced. An example of such rule may be the condition that an event of type *data input* must always be followed by a corresponding event of type *submit*. In the same manner as the labeling function assigns types to events, a similar function can be defined to associate state conditions to occurrence of events. For example: an event execution may be conditional to the satisfaction of a pre-condition; furthermore, the event execution may result in a state satisfying a certain post condition.

Another future avenue deals with the definition of a *satisfiability* relation for use case and task model specifications. A natural definition of *satisfiability* with respect to the used formalism can be formulated through refinement. In this paper we formally defined refinement between two sets of posets based trace inclusion. In this vein, a specification *satisfies* another specification, if the corresponding set of posets of the former specification is a refinement of the set of posets representing the latter specification. Our ongoing efforts aim at further investigating the definition of a suitable *satisfiability* relation but also focus on tool support for the actual verification process.

References

[1] Alur, R., G. Holzmann, and D. Peled, An Analyzer for Message Sequence Charts, *in Proc. of TACAS'96*, (1996) pp. 35-48.

[2] Baeten, J., and W. Weijland, Process Algebra, *Cambridge Tracts in Theoretical Computer Science* 18. Cambridge University Press., Cambridge (1990).

[3] Castellano, L. and G. de Michelis, Concurrency vs. interleaving: introductive example. *Bulletin of the EATCS*, **31**, February, (1987) pp 12-25.

[4] Cockburn, A., Writing Effective Use Cases, Addison-Wesley, (2001).

[5] Dittmar, A. and P. Forbrig, Higher-Order Task Models, *in Proceedings of Design, Specification and Verification of Interactive Systems 2003.*

[6] ITU-T, Recommendation Z.120- Message Sequence Charts, Geneva, 1996.

[7] ITU-T, Recommendation Z.120- Message Sequence Charts, Geneva, 1999.

[8] Jacobson, I., P. Jonsson, M. Christerson, and G. Overgaard, Object-Oriented Software Engineering - A Use Case Driven Approach, Addison Wesley Longman, Upper Saddle River, N.J., 1992.

[9] Katoen, J. and L. Lambert, Pomsets for Message Sequence Charts, *in Proc. of SAM'98*, 1998.

[10] Kuutti, K., Activity Theory as a Potential Framework for Human-Computer Interaction Research. *In Context and Consciousness: Activity Theory and Human Computer Interaction*, MIT Press, 1996.

[11] Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, second edition, Prentice-Hall, 2002.

[12] Linz, P., An Introduction to Formal Languages and Automata, Jones and Bartlett Publishers, Second edition, pp. 83-86, 1997.

[13] Mauw, S., and M.A. Reniers, An Algebraic Semantic of Basic Message Sequence Charts, *In the Computer Journal*, Vol. 37, No. 4, 1994.

[14] Merrick P., and P. Barrow, The Rationale for OO Associations in Use Case Modelling, *In Journal of Object Technology*, Vol. 4, No. 9, 2005.

[15] Mizouni, R., A. Salah, R. Dssouli, and B. Parreaux, Integrating Scenarios with Explicit Loops, *in Proceedings of NOTERE 2004*, Essaidia Morocco, June 2004.

[16] Mori, G., F. Paterno and C. Santoro, CTTE: Support for Developing and Analyzing Task Models for Interactive System Design, *in IEEE Transactions on Software Engineering*, August 2002, pp. 797-813, 2002.

[17] Paterno, F., Model-Based Design and Evaluation of Interactive Applications, Springer, 2000.

[18] Paterno, F., Towards a UML for Interactive Systems, *in Proceedings of EHCI 2001*, Toronto, Canada, pp. 7-18, 2001.

[19] Pratt, V.P., Modeling Concurrency with Partial Orders, *International Journal of Parallel Programming*, 15(1), pp. 33-71, February 1986.

[20] Pressman, R., Software Engineering - A Practitioner's Approach, 6th Edition, Mc Graw Hill, 2005.

[21] Regnell, B., M. Andersson, and J. Bergstrand, A Hierarchical Use Case Model with Graphical Representation, *Proceedings of ECBS'96*, IEEE International Symposium and Workshop on Engineering of Computer-Based Systems, March 1996.

[22] Rui, K., A Process Algebraic Semantics for Refactoring Use Case Models, Doctoral Proposal, Concordia University, 2004.

[23] Seffah, A., M. Metzger, and D. Engelberg, Software and Usability Engineering: Prevalent Myths, Obstacles and Integration Avenues. *In Human-Centered Software Engineering -Integrating Usability in the Software Development Lifecycle*, Springer, 2005.

[24] 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.

[25] Xu, J., W. Yu, K. Rui, and G. Butler, Use Case Refactoring: A Tool and a Case Study, *in Proceedings of APSEC 2004*, Busan, Korea, pp. 484-491.

[26] Zheng, T., F. Khendek and B. Parreaux, Refining Timed MSCs, *in SDL 2003: System Design*, Lecture Notes in Computer Science, Volume 2708/2003, pp. 234-250.