Partial order semantics for use case and task models

Daniel Sinnig, Ferhat Khendek, Patrice Chalin

Formal Aspects of Computing

Partial Order Semantics for Use Case and Task Models

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

[1]Institute of Computer Science, University of Rostock, A.-Einstein-Str. 21, D - 18059 Rostock, Germany
E-mail: dasin@informatik.uni-rostock.de

[2]Faculty of Engineering and Computer Science, Concordia University, 1515 St. Catherine W., Montréal, Qc, Canada..
E-mail: khendek@ece.concordia.ca, chalin@encs.concordia.ca

Abstract. Use case models are the specification medium of choice for functional requirements, while task models are employed to capture User Interface (UI) requirements and design information. In current practice, both entities are treated independently and are often developed by different teams, which have their own philosophies and lifecycles. This lack of integration is problematic and often results in inconsistent functional and UI design specifications causing duplication of effort while increasing the maintenance overhead. To address these shortcomings, we propose a formal semantic framework for the integrated development of use case and task models. The semantic mapping is defined in a two step manner from a particular use case or task model notation to the common semantic domain of *sets of partially ordered sets*. This two-step mapping results in a semantic framework that can be more easily reused and extended. The intermediate semantic domains have been carefully chosen by taking into consideration the intrinsic characteristics of use case and task models. As a concrete example, we provide a semantics for our own DSRG use case formalism and an extended version of ConcurTaskTrees, one of the most popular task model notations. Furthermore, we use the common semantic model to formally define a set of refinement relations for use case and task models.

Keywords. Use Case Models; Task Models; Partially Ordered Sets; Semantics; Formal Framework

1. Introduction

Use case models are the artifact of choice for functional requirements specification [Coc01] while User Interface (UI) design typically starts with the creation of a task model [Pre05]. In current practice, UI design and the specification of functional requirements are generally carried out by different teams using different theories, models and lifecycles [SDM05]. As a consequence, interrelated artifacts, such as use cases and task models, are often created independently of each other. The following issues result directly from this lack of integration:

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

A process allowing for UI design to follow as a logical progression from a functional requirements specification does not exist [SCK07]. Our primary research goal is to define an integrated methodology for the development of use cases and task models within an overall software process. Such an integrated development methodology could serve

Correspondence and offprint requests to: D. Sinnig, E-mail: dasin@informatik.uni-rostock.de

to guide practitioners in the definition of iterative and incremental development processes according to which use case and task models are stepwise refined. A prerequisite of such an initiative, is the elaboration of a formal framework for use case models and task models, the definition of which is the main subject of this paper.

To date, neither use case nor task models have a formal and agreed upon semantics and even less so a common semantics. The absence of a formal semantics hinders the effective verification of refinements and leaves little room for tool support. As a consequence, ambiguities and inconsistencies may go undetected, and are likely to propagate to subsequent development stages, resulting in higher costs to repair them. To address these shortcomings, we present a *common formal semantics* for use case and task models.

Our semantic mapping is performed in two steps: First, the source models are mapped to respective intermediate semantic domains, followed by mappings to the common semantic domain of *sets of partially ordered sets* (sets of posets). The notations we have chosen for source models are our own *DSRG-style use case* formalism and E*xtended ConcurTaskTree* (ECTT) specifications (both are defined in the next section). These notations have been defined as improvements to their state-of-the-art counterparts, Cockburn-style use case models [Coc01] and ConcurTaskTrees (CTT) [PaS01], respectively. As intermediate semantic domains, we use *use case labeled transition systems* (UC-LTS) and *generic task expressions* (GTE).

This paper builds upon our earlier work [SCK07] in both the level of detail and coverage. In particular, we define a complete formal semantics for ECTT. Also, we discuss the abstraction and refinement mappings necessary to formally compare use case and task models for refinement. Both, the semantic mappings and the refinement relations, are illustrated by a non-trivial example. The remainder of this paper is organized as follows. In Section 2, we provide some background information on use case and task modeling. Section 3 provides an overview of our formal framework. In Section 4, we formally specify an abstract syntax for DSRG-style use case models and an abstract syntax for ECTT task models. Section 5 defines the intermediate semantic domains and the associated semantic mappings. This is followed (Section 6) by a formalization of the second level mappings of GTEs and UC-LTSs into the common semantic domain of sets of posets. Several refinement relations for use case and task models are formalized in Section 7. Section 8 discusses relevant related work. Finally, in Section 9 we conclude and provide an outlook of future work.

2. Use case and task modeling

In this section we provide the necessary background information on use case and task modeling. For each concept we present the main features, concrete notations (i.e., DSGR-style use case models and ECTT task models), and a comprehensive example. Finally, both concepts are compared and main commonalities and differences are contrasted.

2.1. Use case models

Use cases were introduced in the early 90s by Jacobson [Jac92]. He defined a use case as a "specific way of using the system by using some part of the functionality." Use case modeling is making its way into mainstream practice as a key activity in the software development process (e.g. Rational Unified Process). There is accumulating evidence of significant benefits to customers and developers [MeB05]. The *use case model* captures the complete set of use cases for an application, where each use case specifies possible usage scenarios for a particular functionality offered by the system. As such, the use case model documents the majority of software and system requirements and serves as a contract between stakeholders about the envisioned system behavior [Coc01].

While some of the original concepts in use case modeling have evolved through the work of researchers and practitioners, the fundamental idea remains the same; that is, a use case describes the way a system is employed by its actors to achieve their goals [ArM01]. In other words, a use case captures the interaction between actors and the system under development. Actors represent users or entities (e.g., secondary systems) that interact with the system. By definition actors are outside of the system boundary. It is distinguished between primary and secondary actors. The primary actor, typically a user, initiates the use case in order to accomplish a pre-set goal. Secondary actors play the role of supporting the execution of the use case and may participate in the interaction later [Gom05].

Different notations for expressing use cases possessing different degrees of formality have been suggested. The extremes range from purely textual constructs written in prose [Coc01] to entirely formal specifications written in Z [BGK98], as Abstract State Machines (ASM) [GLS01; BGS03], or as graph structures [Miz07]. While the use of prose makes use case modeling an attractive tool for facilitating communication among stakeholders, its informal

nature makes it prone to ambiguities and thus leaves little room for tool support. In this article, we adopt an intermediate solution, called *DSRG-style use case model*, which enforces a formal structure but also preserves the intuitive nature of use cases. I.e., we provide support for formalizing the sequencing of use case steps and their types, but the respective actions, as well as the associated conditions are specified informally. The property section of the use case, except for the discrete goal-level property is specified using narrative language.

Fig. 1 portrays the structure of the DSRG-style use case notation first introduced in [SiC07]. Similar to Cockburn [Coc01], each use case starts with a header section containing the various properties. The "primary actor" property identifies the actor who initiates the interaction specified by the use case. The "goal" property captures the very intent the primary actor has in mind when executing the use case. "Level" indicates the goal-level of the use case. While different goal-levels exist, -the most important ones are *summary*, *user goal* and *sub-function*. The "precondition" property denotes a condition that must hold, in order to carry out the use case.

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

The main success scenario as well as each extension consists of a sequence of use-case steps, which can be of seven different kinds. *Atomic* steps are performed either by the primary actor or the system and do not contain any sub-steps. *Choice* steps provide the primary actor with the choice between several interactions. Each such interaction is (in turn) defined by a sequence of steps. *Concurrent* steps define a set of steps which may be performed in any order by the primary actor. *Goto* steps denote jumps to steps within the same use case. *Include* steps define the inclusion of a sub-use case. *Success* and *Failure* denote the successful or unsuccessful termination of use case scenario, respectively.

An example use case is given in Fig 2. The use case captures the interactions for the "Order Product"

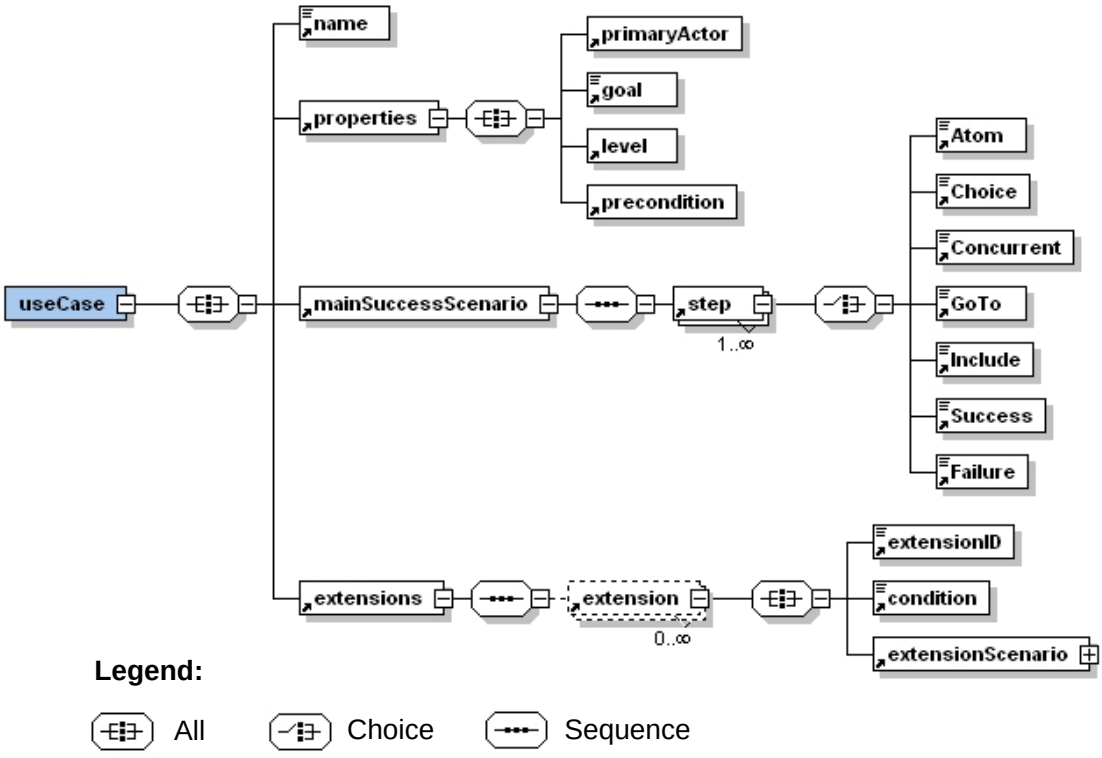

Fig. 1. DSRG-style use case structure

functionality of an Invoice Management System (IMS). The main success scenario of the use case describes the situation in which the primary actor directly accomplishes his/her goal of ordering a product. The extensions specify alternative scenarios which lead to the abandonment of the use case goal. Since this "Order Product" use case is used as a running example for the subsequent syntax and semantics definitions, each use case step is further attributed an abbreviating label, which serves as a short-hand for the narrative action description.

Use case: Order Product

Properties

Goal: Customer places an order for a specific product.
Primary Actor: Customer
Goal-Level: User-goal
Precondition: Customer is logged into the system

Main Success Scenario

1. Customer specifies the desired product category. (spCA)
2. System displays search results that match the Customer's supplied criteria. (diSR)
3. Customer selects a product and identifies the desired quantity. (slPQ)
4. System validates that the product is available in the requested quantity. (vaPQ)
5. System displays the purchase summary. (diPS)
6. Customer *chooses one of the following*

   7A.1. Customer elects to pay by credit card and submits account information. (paCC)

   OR

   7B.1 Customer elects to pay by debit card and submits account information. (paDB)
7. System interacts with the Payment authorization system to carry out the payment. (vaPA)
8. System informs Customer that order is confirmed. (inCO)
9. *Use case ends successfully*

Extensions

3a. Customer is not satisfied with the search results:

3a1. Customer indicates to cancel the use case. (inCA)
3a2. *Use case unsuccessfully.*

4a. The desired product is not available in sufficient quantities:

4a1. System informs Customer that product unavailable in desired quantity. (inIQ)
4a2. *Use case ends unsuccessfully.*

6a. Customer decides to cancel the use case:

6a1. Customer indicates to cancel the use case. (inCA)
6a2. *Use case ends unsuccessfully.*

7a. The payment was not authorized:

7a1. System informs Customer that payment was not authorized. (inPF)
7a2. *Use case ends unsuccessfully.*

Fig 2. "Order Product" use case

2.2. Task models

Task modeling is a well accepted technique supporting user-centered UI design [Pat00]. In most UI development approaches, the task set is the primary input to the UI design stage. *Task models* capture the complete set of tasks that users perform using the application, as well as how the tasks are related to each other. The origin of most task modeling approaches can be traced back to activity theory [Kuu95], where a human operator carries out activities to change part of the environment in order to achieve a certain goal [DiF03]. Like use cases, task models describe the user's interaction with the system. Their primary purpose is to systematically capture the way users achieve a goal when interacting with the system [SLV02]. More precisely, a task model specifies how the user makes use of the system to achieve a goal but also indicates how the system supports the involved (sub)tasks. Various notations for task models exits. Among the most popular ones are ConcurTaskTrees (CTT) [Pat00], GOMS [CMN83], TaO Spec [DFS04], and HTA [AnD67]. Even though all notations differ in terms of presentation, level of formality, and ex-

Table 1. Temporal operators of ECTT

Operator	Syntax	Interpretation
Enabling	$t_1 >> t_2$	Upon successful termination of t_1, t_2 becomes enabled.
Choice	$t_1 [] t_2$	Either t_1 or t_2 is executed. The execution of one task disables the other one.
Order Independence	$t_1 \boxplus t_2$	Execution of t_1 and t_2 in any order.
Concurrency	$t_1 \| t_2$	Interleaved execution of t_1 and t_2 and their subtasks.
Disabling	$t_1 [> t_2$	t_1 becomes deactivated as soon as the first task of t_2 is performed.
Suspend-resume	$t_1 \|> t_2$	At any time the execution of t_1 may be interrupted by t_2. After t_2 has finished its execution t_1 resumes.
Iteration	t^*	t may be executed repetitively (0 or many times).
Optional Tasks	$[t]$	t may be executed or not.
Stop	stop(t)	t cannot enable any tasks.
Resume	resume(t)	Counteracts the effect of *stop*.

pressiveness they share the following common tenet: Tasks are hierarchically decomposed into sub-tasks until an atomic level has been reached. In what follows we describe in detail the task-modeling notation *Extended CTT* (ECTT). ECTT was first introduced in [SWF07] and extends ConcurTaskTrees (CTT) in two dimensions:

1. **ECTT defines two novel temporal operators *Stop* and *Resume*** which allow modeling error and failure cases, and provide a mechanism to "catch" errors and prevent their propagation. Intuitively, *Stop* and *Resume* denote the deactivation and reactivation of the respective operand task. As such their interplay is similar to the *throw* and corresponding *catch* of an exception of programming languages like Java. A task which "throws" a *Stop* exception cannot enable any tasks. *Stop* denotes an exceptional case, which, "untreated", leaves the super-ordinate task incomplete and thus inevitably leads to the premature termination of a scenario. *Resume* is used to "catch" a *Stop* exception and as such counteracts and limits the effects of *Stop*. After *Resume*, the execution of the affected task returns back to "normal"; i.e. its execution will enable respective subsequent tasks.
2. **ECTT is defined in a *modular fashion*** allowing task model to be developed in a true top-down manner while taking advantage of encapsulation. Each ECTT task model consists of a set of atomic tasks and task definitions, where each task definition denotes a high-level task. High-level tasks are further decomposed into so-called task expressions, which are compositions of lower-level task definitions or task references. Task references denote the inclusion of already existing task definitions. In contrast to CTT (which only allows the inclusion of tasks within the same task-tree hierarchy), an ECTT task definition allows the inclusion of any task definition which belongs to the ECTT task model, regardless of whether it is part of the same task hierarchy or not.

Similar to CTT, tasks are arranged hierarchically, with more complex tasks decomposed into simpler sub-tasks. ECTT includes a set of binary and unary temporal operators. The former are used to temporally link sibling tasks, at the same level of decomposition, whereas the latter are used to identify optional and iterative tasks. A summary of ECTT operators together with their intuitive interpretation is given in Table 1. We note that most binary operators (except for suspend/resume) have similar (yet not semantically identical) counterparts in LOTOS [Int97].

An example of an ECTT task model is given in Fig. 3. It corresponds to the "Order Product" use case defined in

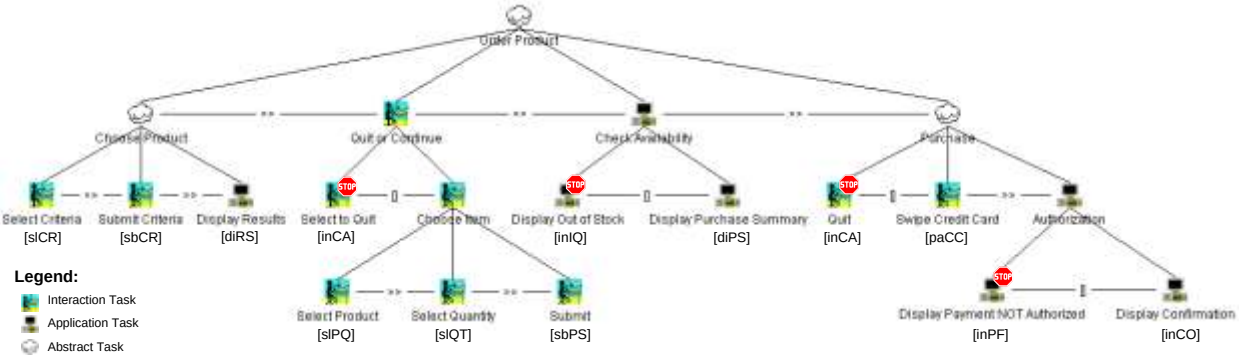

Fig. 3. "Order Product" task model in ECTT

Fig 2. The task model is visualized as a task tree, which clearly portrays the hierarchical breakdown of high-level tasks into lower-level tasks. The execution order of tasks is determined by temporal operators that are defined between peer tasks. An indication of task types is given by the used symbol to represent tasks. ECTT distinguishes between three different task types: *interaction tasks*, *application tasks*, and *abstract tasks*. While interaction tasks are performed by the user (through the UI), application tasks are performed by the system and have an externally visible outcome to the user. Abstract tasks denote high-level tasks which can involve both interaction and application tasks.

2.3. Use case vs. task models

In the previous sections, the main characteristics of use case and task models were presented. In what follows, we will analyze and compare both kinds of artifacts and outline noteworthy differences and commonalities. Use case and task models are both scenario-based and as such capture sets of usage scenarios of the system. On one hand, a use case describes system functionality by means of a main success scenario and extensions. On the other hand, a task specification captures user-system interactions within a hierarchical task tree. At a certain level of abstraction, both models can be used to capture the same information. In current practice, however, use case models are employed to document functional requirements whereas task models are used to describe UI requirements and/or designs. We identify two main differences that are pertinent to their purpose of application:

- In use case models, requirements are captured at a higher level of abstraction whereas task models are more detailed. Hence, the atomic actions of a task specification are often lower-level UI details that are irrelevant (actually contraindicated [Coc01]) in the context of a use case.
- Task models concentrate on aspects that are relevant for UI design and as such, usage scenarios are strictly depicted as input-output relations between the user and the system. System interactions that are hidden from the end user (e.g. involvement of secondary actors or internal computations), as specified in use case models, are *not* captured.

Ideally, the functional requirements captured in use cases are independent of a particular user interface [Coc01]. On the contrary, the requirements and design information captured in task models take into account the specifics of a particular type of user interface. In other words, the use case model captures the bare functional requirements of the system, which are then "instantiated" to a particular type of user interface by means of a task model specification. If the application supports multiple UIs (e.g. Web UI, GUI, Mobile, etc.) then one use case is refined by several task models; one for each "type" of user interface. If given the choice, a task model may only implement a subset of the scenarios specified in the use case model. 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. Generally, refinement between the two models can take two different forms: (1) Structural (event) refinement, which consists of breaking previously atomic use case steps or tasks into sub-steps and sub-tasks respectively; and/or (2) Behavioral refinement, which restricts the set of possible scenarios.

If we compare the "Order Product" use case (Fig 2) with the "Order Product" ECTT task model (Fig. 3) we note that the task model has more UI details. For example use case step 1 ("Specification of Product Category") has been refined by two sequential tasks ("Selection Criteria" and "Submit Criteria"). Moreover, the task model only implements a subset of the functionality of the use case. From a pure functionality point of view (use case) the system supports both payment by credit card and payment by debit card. The capabilities of the UI (task model), however only allows the user to pay by credit card. Finally it is noticeable that the task model does not specify corresponding tasks for use case steps 4 and 7. These steps denote internal system interactions which are irrelevant for UI design.

3. Formal framework

In this section we provide a general overview of our framework for formalizing use case and task models. Fig. 4 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 partially ordered sets* (sets of posets). The semantic mapping is performed in two steps: First, the source models are mapped to respective intermediate semantic domains, followed by mappings to the common semantic domain. The main reason behind a two-step mapping, rather than a direct mapping, is to provide a semantic framework that can be more easily 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

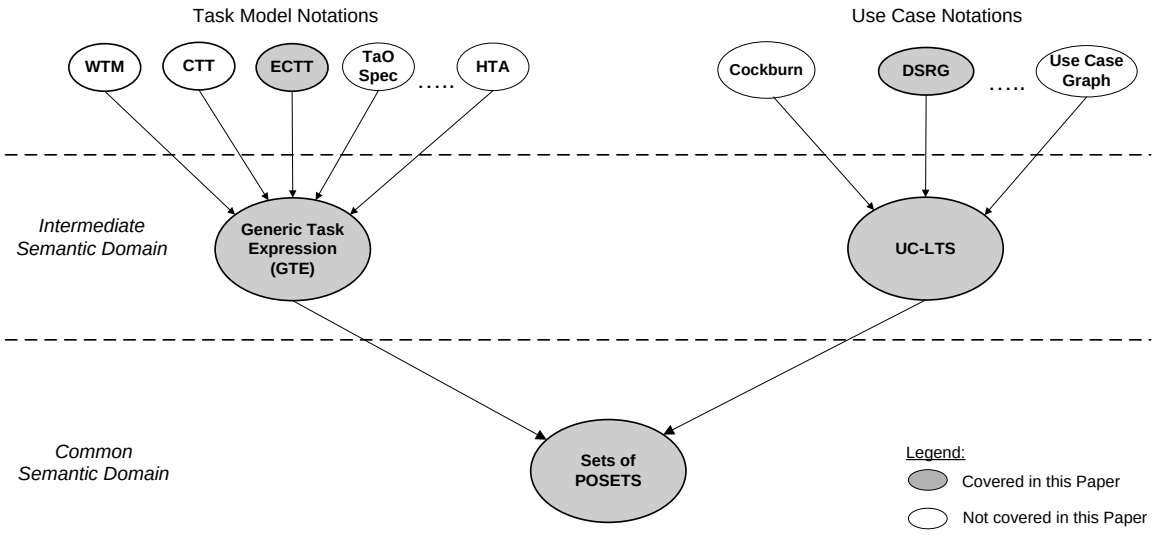

Fig. 4. Two-step semantic mapping

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 the next sections we define the various components of the framework. We start by providing an abstract syntax for a particular use case and task model notation, namely the before-mentioned DSRG-style use case models and Extended Concurrent Task Trees (ECTT). Then we introduce UC-LTS and GTE as intermediate semantic domains and define the involved mappings. Finally we provide formalizations of the semantic mapping to sets posets.

4. Abstract syntax

4.1. Abstract syntax for DSRG-style use case models

We define a DSRG-style use case model as a collection of use cases with one use case designated as the root use case.

Definition 1 (DSRG-style use case model). A *DSRG-style use case model* D is a pair $D = (n_0, \mathcal{U})$ where, $n_0 \in UCNAME$ is the name of the root use case and $\mathcal{U} \in UCNAME \rightarrow USECASE$ is a map of use case definitions (with a finite domain) such that $n_0 \in dom(\mathcal{U})$. If $(n, uc) \in \mathcal{U}$ then we shall write $[n \coloneqq uc]_{\mathcal{U}}$, sometimes omitting the subscript, when it is clear from the context.

The abstract syntax for an individual use case is given in Fig. 5 as an Isabelle/HOL theory[1] [NPW08]. Analogously to the informal definition discussed in Section 2.1, each use case is defined as a record consisting of a use case name, a set of properties, a main success scenario, and a set of extensions. The main success scenario consists of a list of use case steps among which we distinguish between seven different step kinds (datatype $Step$). A use case extension is defined as a record consisting of an identifier, a condition, and a list of use case steps. The latter denote an alternative flow, relative to the main success scenario (or a super-ordinated extension). In case of *atomic* use case steps we further distinguish between three different step types: steps of type *interaction* are performed by the primary actor, whereas steps of types *application* and *internal* are carried out by the system, with the difference that the former have an externally visible effect (to the primary actor) while the effects of the latter are invisible.

[1]Expressing parts of our formal system in Isabelle allowed us to use the Isabelle theorem prover to verify basic well-formedness properties such as syntax and type checking

```
theory uc
imports Main begin

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

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

types PrimStep = Label

datatype Step = Atom StepID StepType Label "ExtensionID set"|
                Choice StepID "(Step list) list²"  "ExtensionID set" |
                Concurrent StepID "PrimStep set" "ExtensionID set" |
                Goto StepID StepID |
                Include StepID UCName |
                Success StepID |
                Failure StepID

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

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

Fig. 5. DSRG-style use case syntax formalized in Isabelle.

As well-formedness conditions, we required that (1) all use case steps and extension IDs be unique, (2) for every step or extension reference, there exist a corresponding use case step or use case extension within the same use case, respectively, (3) for every $Include\ (id, n)$ we require that $n \in dom(U)$ and that there be no circular inclusions, and (4) the last element of every use case step sequence be either $Goto$, $Success$, or $Failure$. In order to illustrate the syntactic definition of a use case, let us reconsider the previously depicted "Order a Product" use case. Fig. 6 portrays parts of its formalization in Isabelle/HOL. For the sake of conciseness, for each *atomic* step, instead of the full description, the abbreviating label has been used.

```
constdefs
OrderProductUC :: UseCase
 "OrderProductUC == (|
   Name = ''Order Product'',
   Properties = (|
    Goal = ''Primary actor places an order for a specific product'',
    PrimaryActor = ''Customer'',
    GoalLevel = USERGOAL,
    Precondition = ''Primary actor is logged into the system.''
   |),
   MainSuccessScenario = [
      Atom ''s1'' INTERACTION ''spCA'' {},
      Atom ''s2'' APPLICATION ''diSR'' {},
      Atom ''s3'' INTERACTION ''slPQ'' {''e1''},
      Atom ''s4'' INTERNAL    ''vaPQ'' {''e2''},
      Atom ''s5'' APPLICATION ''diPS'' {},
      Choice ''s6'' [
       [Atom ''s6A1'' INTERACTION ''paCC'' {} ],
       [Atom ''s6B1'' INTERACTION ''paDB'' {} ]
       ] {''e3''},
      Atom ''s7'' INTERNAL    ''vaPA'' {''e4''},
      Atom ''s8'' APPLICATION ''inCO'' {},
      Success ''s9''
   ],
```

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

```
    Extensions = {

(|(*Extension 3a*)
     ID = ''e1'',
     Condition = ''Primary Actor is not satisfied with search results'',
     ExtensionScenario =
      [ Atom ''s3a1'' INTERACTION ''inCA'' {},
        Failure ''s3a2'']
    |),
    (|(*Extension 4a*)
      ID = ''e2'',
      Condition = ''The product is unavailable in sufficient quantities'',
      ExtensionScenario =
       [ Atom ''s4a1'' APPLICATION ''inIQ'' {},
         Failure ''s4a2'' ]
    |),
    (|(*Extension 6a*)
      ID = ''e3'',
      Condition = ''Primary Actor decides to cancel the use case'',
      ExtensionScenario =
       [ Atom ''s6a1'' INTERACTION ''inCA'' {},
         Failure ''s6a2'' ]
     |),
    (|(*Extension 7a*)
      ID = ''e4'',
      Condition = ''The payment was not authorized'',
      ExtensionScenario =
       [ Atom ''s7a1'' APPLICATION ''inPF'' {},
         Failure ''s7a2'' ]
    |) } |)"
```

Fig. 6. Formalized syntax of "Order Product" use case

4.2. Abstract syntax for ECTT task models

In this section, we define an abstract syntax for ECTT task models.

Definition 2 (ECTT task model). An *ECTT task model* C is a triple $C = (n_0, \mathcal{D}, \tau)$ where, $n_0 \in TASKNAME$ is the name of the main task definition of the ECTT task model, $\mathcal{D} \in TASKNAME \rightharpoonup TASKEXPR$ is a partial map of task definitions. (The set of task expressions ($TASKEXPR$), is the smallest set closed under the following two rules: (1) $n \in TASKNAME$ is a task expression. (2) Let υ, φ be ECTT task expressions then $\upsilon \gg \varphi$, $\upsilon [\,] \varphi$, $\upsilon \parallel \varphi$, $\upsilon \boxplus \varphi$, $\upsilon[> \varphi$, $\upsilon \mid > \varphi$, $[\upsilon]$, υ^*, υ^+, $stop(\upsilon)$, $resume(\upsilon)$ are also ECTT task expressions. Note that $n_0 \in dom(\mathcal{D})$. If $(n, o) \in \mathcal{D}$ then we shall write $[n := o]_{\mathcal{D}}$, sometimes omitting the subscript, when it is clear from the context. We say that a task name n denotes an **atomic task** if $n \notin dom(\mathcal{D})$ or a **task reference** if $n \in dom(\mathcal{D})$). $\tau \in TASKNAME \rightharpoonup \{abstract, interaction, application\}$ is a typing function that associates a task type with each task name in C.

C = (Order Product, $\mathcal{D}$, τ), with
$\mathcal{D}$ = {
Order Product := Choose Product ≫ Quit or Continue ≫ Check Availability ≫ Purchase
Choose Product := slCR ≫ sbCR ≫ diRS
Quit or Continue := $stop$(inCA)[ ] Choose Item
Choose Item := slPQ ≫ slQT ≫ sbPS
Check Availability := diPS[ ] $stop$(inIQ)
Purchase := ($stop$(inCA) [ ] paCC) ≫ Authorization
Authorization := $stop$(inPF)[ ] inCO

$$\tau(t) = \begin{cases} abstract, & if\ t \in \{\text{Order Product, Choose Product, Purchase}\} \\ interaction, & if\ t \in \{\text{slCR, sbCR, Quit or Continue, inCA, Choose Item, slPQ, slQT, sbPS, paCC}\} \\ application, & if\ t \in \{\text{diRS, Check Availability, diPS, inIQ, Authorization, inCO, inPF}\} \end{cases}$$

Fig. 7. Partial ECTT formalization of the IMS task model

In contrast to CTT, the various task definitions ($\mathcal{D}$) do *not* need to be *connected* by some task-subtask hierarchy. This allows for a more modular setup, enabling the UI designer to work on multiple task hierarchies concurrently and eventually connect them using references. The creation of a single monolithic task tree (as required by CTT) is not necessary. For an ECTT task model to be well-formed, we require that C contain only non-recursive task definitions, that the task type of atomic tasks be either $interaction$ or $application$ and that direct and indirect operands of ||, |>, [> be of type interaction. In order to illustrate the before-mentioned definitions let us reconsider the "Order Product" task model visualized by Fig. 3. The corresponding formalization as an ECTT task model is depicted in Fig. 7. Leaf task names are abbreviated by the label displayed underneath the respective task symbols.

5. Intermediate semantic domains

In this section we introduce use case labeled transition systems (UC-LTSs) and generic task expressions (GTEs) as intermediate semantics domains for use case and task models, respectively. We also formally define the mappings from DSRG-style use case models and ECTT task models to their respective intermediate semantic domains.

5.1. UC-LTS

The intermediate semantic domain for use case models is UC-LTSs. Its definition is similar to the definition of an ordinary LTS [BaW90] with the exception that transitions are associated with sets of labels rather than single labels.

Definition 3 (Use case labeled transition system). A *Use case labeled transition system* (UC-LTS) is a tuple $U = (\Sigma, Q, q_0, F, \delta)$, where Σ is the set of labels representing atomic use case steps, Q is a set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\delta: (Q \times \mathbb{P}_1(\Sigma)) \longrightarrow \mathbb{P}(Q)$ is the (total) transition function.

We believe that UC-LTSs are defined in a manner which easily and intuitively captures the nature of use cases, as we explain next. A use case primarily describes the possible execution order of user and system actions in the form of use case steps: from a given state, the execution of a step leads into another state. Accordingly, in UC-LTSs, the execution of a step (or set of steps, as we shall explain shortly) is denoted by a transition from a source state to a target state. Each transition is associated with a non-empty set of labels, where each label represents an atomic use case step. The execution order of use case steps is modeled using transition sequences, where the target state of a transition serves as the source state of the following transition. For a given transition, if the associated label set contains more than one label, then no specific execution order exists between the corresponding use case steps. I.e., a transition is triggered when all associated primitive steps are executed; the execution order, however, is arbitrary. The mapping from a DRSG-style use case model to a UC-LTS is defined in two steps:

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

Definitions of the mappings require: (1) An input use case model in canonical form, (2) proper initialization of a global environment (env) and (3) the global equivalence relation ($\sim$). In the following, details of each requirement will be given.

Definition 4 (Canonical form of a use case model). A use case model is in a *canonical form*, if and only if:

i. it is well formed,
ii. each use case extension is associated with exactly one step, and
iii. each use case (except for the root use case) is invoked by exactly one $Include$ step.

While the "Order Product" use case (Fig. 6) is already in canonical form, an arbitrary well-formed use case model can be transformed into canonical form in a straight-forward manner. In order to satisfy condition (ii), instead of the original extension, use case steps are associated with distinct copies of the respective extensions. If steps of the original extension are referenced by means of a $Goto$ step, the respective reference is to be updated accordingly. Similarly, in order to satisfy condition (iii) instead of the original sub-use case n, each $Include$ step is associated with a distinct copy (n') of n. For example, if, throughout the use case model, use case n is included three times by

steps $Include(id_1, n)$, $Include(id_2, n)$ and $Include(id_3, n)$, then we create three copies of n (n', n'', n''') and modify the inclusion steps as follows: $Include(id_1, n')$, $Include(id_2, n'')$ and $Include(id_3, n''')$.

We also require the proper initialization of a global environment, env. As defined by Fig. 8 (left hand side), env has three fields, named uc, ext and $step$, where: uc is a function that maps use case names to $UCStateInfo$ which, according to Fig. 8 (right hand side), defines for each use case the initial state (q_0) of the UC-LTS representing the main success scenario and the set of states representing the successful (F_S) and unsuccessful (F_F) termination of the use case. $step$ and ext are bijective functions that map a given step id or extension id to the initial state of the UC-LTS representing the use case step and use case extension, respectively. Recall that according to the well-formedness conditions of a DSRG-style use case model, step and extension ids are unique within any given use case model.

$\sim \subseteq STATE \times STATE$ is an equivalence relation (reflexive, symmetric, and transitive) defined over $STATE$, the set of all states. During merging, all equivalent states will be merged to a single state denoting its respective equivalence class. In order to satisfy the reflexivity requirement, $\sim$ is initialized as follows: $\sim = \{(q, q) \mid q \in STATE\}$.

```
record Environment =                          record UCStateInfo =
   uc   :: "UCName => UCStateInfo"               q0 :: "STATE"
   ext  :: "ExtensionID => STATE"                Fs :: "STATE set"
   step :: "StepID => STATE"                     Ff :: "STATE set"
```

Fig. 8. Definition of global environment and UCStateInfo

5.1.1. Generation

Given a use case model in canonical form, the generation of a set of UC-LTSs is performed in a bottom-up manner. We start with the mapping of an individual use case step. As defined in the abstract use case syntax (Fig. 5), there are 7 kinds of use case steps. Each step kind has its own specific mapping to a UC-LTS.

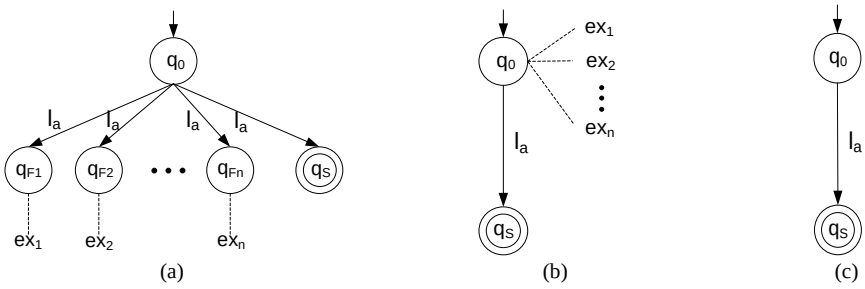

Fig. 9. UC-LTSs representing atomic use case steps

As depicted in Fig. 9, depending on the step type (denoted by t) atomic steps are mapped into different UC-LTSs. The rationale behind each case is as follows: Each *internal* (a) use case step has $n + 1$ different outcomes, among which one is captured in the main success scenario and the remaining $n \geq 0$ outcomes are captured by the corresponding extensions. Hence, the resulting UC-LTS consists of $n + 1$ transitions; one transition for the main success scenario and n transitions for each defined extension. The former results in a final state, which will be used for the sequential composition of use case steps. The latter result in a set of non-final states. During merging, these states will be joined with the initial states of the UC-LTSs representing the various extensions. This is defined by adding the respective state pairs to the global equivalence relation ($\sim$).

In contrast to internal use case steps, which are performed by the system and are hidden from the user, steps of type *interaction* are performed by the user. As such, they do not have an alternative outcome per se, but may be associated (by virtue of one or more extensions) with alternative steps which are performed instead of the actual step. As a result, the corresponding UC-LTS consists of only one transition (from q_0 to q_s), representing the use case step (b). Alternative steps are captured in the UC-LTSs representing the corresponding extensions. During "Merging" the initial states of each UC-LTS representing an extension are identified with q_0. This is defined by updating $\sim$ accordingly. Steps of type *application* are performed by the system and have an externally visible effect to the user. They are performed in response to an *internal* or *interaction* step. As a consequence, they are not associated with any extension, and the corresponding UC-LTS consists of only one transition (c).

The mapping of the remaining step kinds is briefly outlined next. The full details can be found in [Sin08]. A *Choice* step is mapped to a UC-LTS which results from merging the initial states of the UC-LTSs representing the

involved step sequences. The mapping of a *Concurrent* step corresponds to the construction of the product machine of the involved UC-LTSs. *Goto* steps denote a branching to a use case step. The corresponding UC-LTS consists of a single state, which is defined equivalent (by means of $\sim$) with the initial state of the UC-LTS representing the target use case step. *Include* denotes the invocation of a sub-use case. The corresponding UC-LTS consists of two states (q_0 and q_s) which are not (yet) connected by any transition. During "Merging" the initial state of the UC-LTS representing the main success scenario and all final states of the UC-LTSs representing the sub-use case will be merged with q_0 and q_s , respectively. *Success* and *Failure* steps denote the successful or unsuccessful termination of a use case scenario. In both cases the corresponding UC-LTS consists of only a single (final) state.

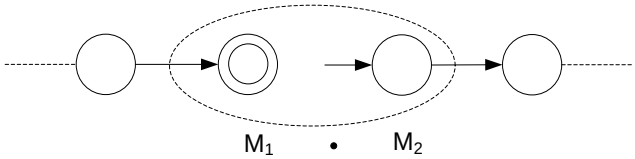

Fig. 10. Sequential composition of nFSMs

Having defined the mapping for individual UC steps, we continue with defining the mapping of step sequences to UC-LTS. The mapping of a list of use case steps corresponds to the binary sequential composition ($\cdot$) of the UC-LTSs of the individual steps. As schematically depicted in Fig. 10, the sequential composition consists of unifying the final states of the first operand and the initial state of the second operand.

Definition 5 (Mapping a step sequence to a UC-LTS). Given $\langle s_1, s_2, \dots s_k \rangle$, a non-empty step sequence of $k \geq 1$ steps, we define the mapping of step sequences to a UC-LTS as follows:

$$\mathcal{M}_{Seq}[\![\langle s_1, \dots s_k \rangle]\!] = \mathcal{M}_{Step}[\![s_1]\!] \ \cdots \cdot \mathcal{M}_{Step}[\![s_k]\!].$$

An entire use case is mapped into a set of UC-LTSs. The resulting set contains one UC-LTS for the main success scenario and one UC-LTS for each defined extension.

Definition 6 (Mapping a use case to a set of UC-LTSs). Let $uc = (n, Prop, Mss, \{ex_1, ex_2, \dots, ex_n\})$ be a use case with e$x_i = (id_i, condition_i, s_i)$. We then obtain the corresponding set of UC-LTSs as follows:

$$\mathcal{M}_{Uc}[\![uc]\!] = \{\mathcal{M}_{SeqUclts}[\![Mss]\!], \mathcal{M}_{SeqUclts}[\![s_1]\!], \dots, \mathcal{M}_{SeqUclts}[\![s_n]\!]\}.$$

Finally, we define the mapping of a set of use cases to a set of UC-LTSs as the union of the sets of UC-LTSs representing the various use cases.

Definition 7 (Mapping a set of use cases to a set of UC-LTSs). Let $\{uc_1, uc_2, \dots, uc_m\}$ be a set of use cases. We then obtain the corresponding set of UC-LTSs as follows: $\mathcal{M}_{Ucs}[\![\{uc_1, uc_2, \dots, uc_m\}]\!] = \bigcup_{i=1}^{m} \mathcal{M}_{Uc}[\![uc_i]\!]$.

For illustrative purposes, Fig. 11 portrays the set of UC-LTSs of the "Order Product" use case model. As depicted, the set consists of five UC-LTSs; one for the main success scenario of "Order Product" and one for each of the four extensions. States that belong to the same equivalence class (by means of $\sim$) are circled by a dashed line. During "Merging", these states will be combined to a single state to obtain a single consolidated UC-LTS.

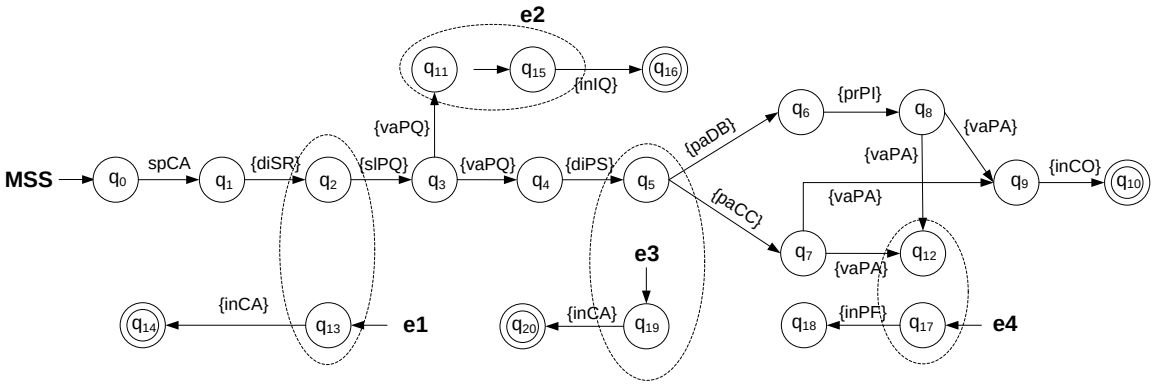

Fig. 11. UC-LTSs of the "Order Product" use case

5.1.2. Merging

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

Definition 8 (Mapping a use case model to UC-LTS). Let $D = (n_0, UC)$ be a well-formed use case model in canonical form, $\{uc_1, uc_2, \dots, uc_m\}$ be the range of UC, and $\{U_1, U_2, \dots, U_n\}$ be the result of $\mathcal{M}_{Ucs}[\![\{uc_1, uc_2, \dots, uc_m\}]\!]$ with $U_i = (\Sigma_i, Q_i, q_{0_i}, F_i, \delta_i)$ and $n \geq m$. The mapping to UC-LTS is then defined as follows: $\mathcal{M}_{Ucm}[\![n_0, UC]\!] = (\Sigma, Q, q_0, F, \delta)$ with $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \dots \cup \Sigma_n$, $Q = (Q_1 \cup Q_2 \cup \dots \cup Q_n)/\sim$, $q_0 = [env.uc(n_0).q_0]$, $F = \Pi(env.uc(n_0).F_S \cup env.uc(n_0).F_F)$, where Π is the generalized canonical projection map defined as $\Pi(Q, \sim) = \{\pi(q, \sim) | \ q \in Q\}$, and $\delta([q]_\sim, w) = \bigcup_{\hat{q} \in [q]} (\bigcup_{i=1}^{n} \Pi(\delta_i(\hat{q}, w), \sim))$.

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

Fig. 12 portrays the UC-LTS obtained by merging the various UC-LTSs given in Fig. 11. The resulting UC-LTS has five final states: $[q_{10}]$ denoting the successful outcome of the use case (i.e., the customer succeeded to order the product), $[q_{14}]$ and $[q_{20}]$ denoting the case where the user cancels the use case, $[q_{16}]$ denoting the case where the product is not available, and $[q_{18}]$ denoting case where the payment was not authorized. Notice that the resulting UC-LTS has fewer states than the accumulated number of states of the involved UC-LTSs. This is because, during merging, two or more states are combined into a single state, representing the respective equivalence class.

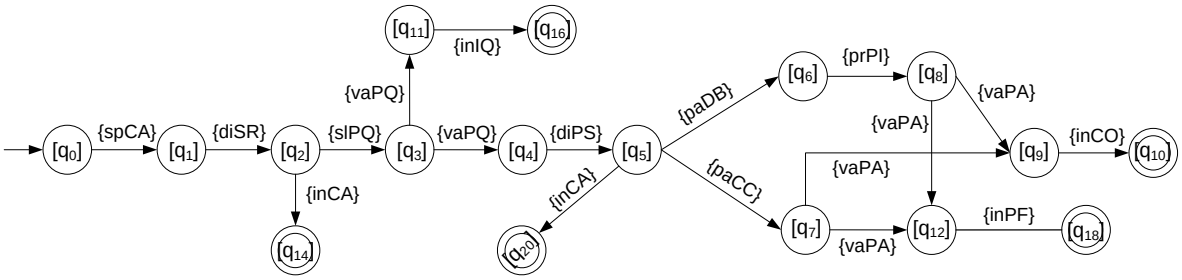

Fig. 12. UC-LTS representing "Order Product" use case

5.2. Generic task expressions

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

Definition 9 (Generic task expression). Let ψ and ρ be generic task expressions and $\alpha \in T$ be an atomic task, then the set of *generic task expression GTE* is the smallest set closed under following rules: (1) α is a generic task expression and (2) $\psi \gg \rho$, $\psi \, [\,] \, \rho$, $\psi \parallel \rho$, $[\psi]$, (ψ^*), $stop(\psi)$, $resume(\psi)$ are also generic task expressions.

In contrast to an ECTT task model (Definition 2), a generic task expression abstracts away from high-level task names (i.e. task definitions). Instead of using task definitions, the behavior of the task model is captured in a single generic task expression. While high-level task names are important at the modeling stage to foster the comprehension of the task model, they are irrelevant for capturing behavioral information. Compared to an ECTT task expression, a generic task expression may not contain high-level operators (e.g., *order independency*, *disabling*, or *suspend / resume*). These operators are important, as syntactic sugar, at the modeling stage to obtain a concise and comprehensible task model. However, they do not enrich the expressiveness of an ECTT task expression and can be rewriting using low-level operators. In what follows we present a mapping which transforms each ECTT task expression into a corresponding generic task expression.

Definition 10 (Mapping an ECTT task expression to a generic task expression). Let υ, φ be ECTT task expressions, n be a task name and $\mathcal{D}$ be a finite map of ECTT task definitions. We then define the mapping $\mathcal{M}_{EcttGte}$ to generic task expressions, relative to a given task definition map $\mathcal{D}$, as follows:

$$\mathcal{M}_{EcttGte}[\![n]\!]_{\mathcal{D}} = \begin{cases} \mathcal{M}_{EcttGte}[\![\mathcal{D}(n)]\!]_{\mathcal{D}}, & if\ n \in dom(\mathcal{D}) \\ n, & otherwise \end{cases}$$
$$\mathcal{M}_{EcttGte}[\![\upsilon \gg \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}} \gg \mathcal{M}_{EcttGte}[\![\varphi]\!]_{\mathcal{D}}$$
$$\mathcal{M}_{EcttGte}[\![\upsilon\ [\ ]\ \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}}\ [\ ]\ \mathcal{M}_{EcttGte}[\![\varphi]\!]_{\mathcal{D}}$$
$$\mathcal{M}_{EcttGte}[\![\upsilon \parallel \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}} \parallel \mathcal{M}_{EcttGte}[\![\varphi]\!]_{\mathcal{D}}$$
$$\mathcal{M}_{EcttGte}[\![\upsilon \boxplus \varphi]\!]_{\mathcal{D}} = (\mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}} \gg \mathcal{M}_{EcttGte}[\![\varphi]\!]_{\mathcal{D}})\ [\ ]\ (\mathcal{M}_{EcttGte}[\![\varphi]\!]_{\mathcal{D}} \gg \mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}})$$
$$\mathcal{M}_{EcttGte}[\![\upsilon\ [>\ \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{EcttGte}[\![\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}} \gg \varphi\ ]\!]$$
$$\mathcal{M}_{EcttGte}[\![\upsilon\ |>\ \varphi]\!]_{\mathcal{D}} = \mathcal{M}_{EcttGte}[\![\mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ \varphi^*]\!]$$
$$\mathcal{M}_{EcttGte}[\![[\upsilon]]\!]_{\mathcal{D}} = [\mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}}]$$
$$\mathcal{M}_{EcttGte}[\![\upsilon^*]\!]_{\mathcal{D}} = (\mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}})^*$$
$$\mathcal{M}_{EcttGte}[\![\upsilon^+]\!]_{\mathcal{D}} = \mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}} \gg (\mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}})^*$$
$$\mathcal{M}_{EcttGte}[\![stop(\upsilon)]\!]_{\mathcal{D}} = stop(\mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}})$$
$$\mathcal{M}_{EcttGte}[\![resume(\upsilon)]\!]_{\mathcal{D}} = resume(\mathcal{M}_{EcttGte}[\![\upsilon]\!]_{\mathcal{D}})$$

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

$$\text{slCR} \gg \text{sbCR} \gg \text{diRS} \gg \big(stop(\text{inCA})\ [\ ]\ (\text{slPQ} \gg \text{slQT} \gg \text{sbPS})\big) \gg (stop(\text{inIQ})\ [\ ]\ \text{diPS}\ ) \gg\ (stop(\text{inCA})[\ ]\ \text{paCC}) \gg (stop(\text{inPF})[\ ]\ \text{inCO})$$

6. Common formal semantics

In this section we define the second-level mappings (Fig. 4) to the semantic domain of *Sets of Partial Order Sets* (set of posets). We start by providing necessary definitions. Then, we present a procedure that, given a UC-LTS, generates the corresponding set of posets. We also define a semantic function that maps a generic task expression (GTE) to a corresponding set of posets.

6.1. Definitions

Fundamental to our approach is a differentiation between *events* and *event names* as well as a formalization of the set operation *disjoint union* (+).

6.1.1. Mathematical Definitions and Preliminaries

Definition 11 (Events). Let $EVENTNAME$ represent the set of all possible event names. We then define an *event* as a pair consisting of an event name n and an index i. Correspondingly, the set of all events is defined as: $EVENT = EVENTNAME \times \mathbb{N}$. For all $(n,\ i) \in EVENT$ we define the obvious projection function $name: EVENTNAME \times \mathbb{N} \longrightarrow EVENTNAME$, such that $name(n, i) \mapsto n$ and its generalization, applied to sets of events being applied to all elements of the set. We reserve the name $STOP \in EVENTNAME$; its semantics will be given later.

In what follows, event names may be used to represent events with index 0; i.e., as needed, we assume the implicit conversion from $EVENTNAME \longrightarrow EVENT$, defined by $n \longmapsto (n, 0)$. We will use the symbol E, possibly decorated with primes ($E', E'', \ldots$) and/or subscripts ($E_1, E_2, \ldots$), to represent a subset of $EVENT$. Next we define the set operation *disjoint union*. It is used as an auxiliary function for the definition of composition operators for partially ordered sets, which are needed for the semantic mapping.

Definition 12 (Disjoint union). Using standard notation, we represent the *disjoint union* $(+)$ of two event sets as: $E_p + E_q = E_p^{*0} \cup E_q^{*1}$, where $E^{*b} = \{e * b \mid e \in E\}$ for $b \in \{0,1\}$ with $(n, i) * b = (n, i \times 2 + b)$

Our definition of the *disjoint union* is similar to what has been proposed by Blyth [Bly75]. In both cases an index set is used to distinguish between events that have the same name. In contrast to Blyth, however, we use a natural number instead of an *n*-ary tuple over $\{0,1\}$. We generalize $+$ and $\_^{*b}$ to binary relations over $EVENT$; i.e., given $R: \mathbb{P}(EVENT \times EVENT)$ we define $R^{*b} = \{(e * b, e' * b) | (e, e') \in R\}$ and $R_p + R_q = R_p^{*0} \cup R_q^{*1}$.

6.1.2. Semantic Domain: Sets of Posets

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

Definition 13 (Poset over events). A partially ordered set (poset) over events is a tuple $(\boldsymbol{E}, \leq)$, where $E \subseteq EVENT$ is a set of events and $\leq \subseteq E \times E$ is a partial order relation (reflexive, anti-symmetric, transitive) defined over E. This relation specifies the causal order of events.

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

Definition 14 (Sequential and parallel composition of posets). Let $p = (E_p, \leq_p)$ and $q = (E_q, \leq_q)$ be posets. We define the *sequential composition* $(\cdot)$ and *parallel composition* $(\|)$ as follows.

$$p \cdot q = \begin{cases} p, & STOP \in name(E_p) \\ (E_r, \leq_r), & otherwise \end{cases} \text{ where } \begin{array}{l} E_r = E_p + E_q \\ \leq_r = (\leq_p + \leq_q) \cup \{(e_p * 0, e_q * 1) \mid e_p \in E_p, e_q \in E_q\} \end{array}$$

$$p \parallel q = (E_p + E_q, \leq_p + \leq_q)$$

Intuitively, if the event set of p does *not* contain $STOP$, then the *sequential composition* $p \cdot q$ places all events of q strictly after all the events of p. Otherwise, $p \cdot q$ simplifies to p regardless of q. In contrast to the *sequential composition*, the *parallel composition* does not make a case distinction between posets that contain or do not contain an event named $STOP$. The insertion and the removal of *STOP* to/from a poset are the so-called *closing* and *opening* operations and are defined as follows:

Definition 15 (Closing and opening of a poset). Let $p = (E_p, \leq_p)$ be a poset. We define the *closing* operation $close(p) = (E_p \cup \{STOP\}, \leq_p \cup \{(STOP, STOP)\})$ and the *opening* operation $open(p) = p \setminus \{STOP\}$.

In [Sin08], we have formally proven that posets are closed under the operations *sequential composition*, *parallel composition*, *opening* and *closing*. Also fundamental to our model is the notion of a *trace*. In general, a trace of a partial order set corresponds to a totally ordered event-name sequence such that the corresponding sequence of events is a linear extension of the partial order. It is important to note that events with event name $STOP$ are *not* part of a trace. Note that $e_j \not\leq e_i$ holds true, if either $e_i \leq e_j$ or e_i and e_j are unrelated (by means of $\leq_p$).

Definition 16 (Set of all traces). Let $p = (E_p, \leq_p)$ be a poset. We define the function Tr which yields the set of all traces of p as follows:

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

The common semantic domain for use case and task models is *sets of partially order sets* and is defined as follows:

Definition 17 (Set of partial order sets). A *set of partial order sets P* is a possibly infinite collection of posets

$$P = \{p_1, p_2, \ldots\}.$$

Sets of posets can be composed by the operators defined in Definition 18. In contrast to the operators defined on posets we additionally introduce *alternative composition* and *closure*. Both are needed for the semantic mappings defined in the next sections. Note that similar to the Kleene star operation for regular expressions [GMU07], *closure* returns the set of posets that are formed by computing the union of all possible (repeated) sequential compositions of a given set of posets.

Definition 18 (Operators for sets of posets). Let P and R be sets of posets. We define the *sequential composition* ($\cdot$), *parallel composition* ($\|$), *alternative composition* (#), *closing*, *opening*, and *closure* ($*$) as follows:

$$P \cdot R = \{p_i \cdot r_j \mid p_i \in P, r_j \in R\}$$
$$P \parallel R = \{p_i \parallel r_j \mid p_i \in P, q_j \in R\}$$
$$P \# R = P \cup R$$
$$close(P) = \{close(p) \mid p \in P\}$$
$$open(P) = \{open(p) \mid p \in P\}$$
$$P^* = \bigcup_{k=0}^{\infty} P^k \text{ where } P^k \text{ is defined as } P^k = \begin{cases} \{\emptyset_{poset}\}, & if\ k = 0 \\ P \cdot P^{k-1}, & k > 0 \end{cases}$$

Finally, we define the *set of all traces* for a set of posets. It forms the essential basis for establishing refinement relations (Section 7) between use case and task models.

Definition 19 (Set of all traces of a set of posets). The *set of all traces* of a set of posets P is defined as:

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

Based on the definition of the *set of all traces*, we can derive certain *trace properties* for each defined set of poset operation. These are given in Appendix B.

6.2. Semantic rules

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

6.2.1. Mapping UC-LTSs to sets of posets

Definition 20 (Mapping UC-LTS to a set of posets). Let $U = (\Sigma, Q, q_0, F, \delta)$ be a UC-LTS. We then define the mapping to a set of posets as follows:

$\mathcal{M}_{UcltsSposet}[\![U]\!] = LTS\_to\_SPO(U)$. For this purpose we have devised the algorithm LTS_to_SPO. Fig. 13 gives the corresponding pseudo code.

Without loss of generality, the algorithm assumes that there are no outgoing transitions from any of the final states in the input UC-LTS. This is a valid assumption since, according to the well-formedness rules for DSRG-style use cases (Section 4.1), *Failure* and *Success* steps are always at the end of any step sequence and cannot have any extensions. We also note that the main idea for the algorithm stems from the well-known algorithm that transforms a deterministic finite automaton into an equivalent regular expression [GMU07]. Instead of stepwise composing regular expressions, we compose sets of posets.

The procedure starts (1) with the creation of an initial *generalized UC-LTS* internally represented by a two-dimensional array ('SPO'). The array is populated with all transitions of the given UC-LTS specification. Indexed by a source and a target state, an array cell contains a set of posets constructed from the label(s) associated to the representative transition. If the label is a singleton set, then the corresponding set of posets contains a single poset containing the respective event. If the label consists of multiple events, indicating the concurrent or unordered execution of use case steps, the set of posets will contain a poset which consists of several elements. Those elements, however, are not causally related. We note that the idea of a generalized UC-LTS is similar to the concept of a *generalized finite state machine* [GMU07]. Instead of labeling the transitions with regular expressions, transitions are labeled with sets of posets.

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

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

Fig. 13. LTS_to_SPO algorithm transforming a UC-LTS to a set of posets

The core part of the algorithm consists of two nested loops. The outer loop (2) iterates through all states of the generalized UC-LTS (except for the initial and the final states) whereas the inner loop (3) iterates through all pairs of incoming and outgoing transitions for a given state. For each found pair (q_k, p_m), we perform the following (4): Compute the *alternative composition* of:

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

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

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

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

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

Next (5) we add a new transition from the source state of the incoming transition to the target state of the outgoing transition. Note that the corresponding cell in SPO has already been populated with the result of (4). Back in the outer loop, we eliminate (6) the currently visited state from the generalized UC-LTS and proceed with the next state. Once the generalized UC-LTS consists of only the initial state and the final states we exit the outer loop and perform the following two computations, in order to obtain the final result. First (7) we perform an *alternative composition* of the sets of posets of all the transitions from the initial state to a final state. Second, if the initial state additionally contains a self loop (8) then we *sequentially compose* the result of the *closure composition* of the set of posets denoted by that self loop and the result of the before-mentioned *alternative composition*.

If we apply the LTS_to_SPO algorithm to the "Order Product" UC-LTS we obtain the set of posets depicted in Fig. 14. For the sake of conciseness events are represented only by their name, while the index has been omitted. This simplification was possible because none of the entailed posets contains two or more events sharing the same name. The various parts of the resulting set of posets are interpreted as follows: Having indicated the desire to order a product, the primary actor searches for a product and as a result (1, 3) elects to quit the system, (2) the selected product is not available in the desired quantity or he/she decides to checkout and pay. In the latter case, the debit or credit card payment is either authorized (4b, 5b) or rejected (4a, 5a).

(1)	$\{(\{\text{spCA}, \text{diRS}, \text{inCA}\}, \{(\text{spCA}, \text{diRS}), (\text{diRS}, \text{inCA})\}^*)\} \cup$
(2)	$\{(\{\text{spCA}, \text{diRS}, \text{slPQ}, \text{vaPQ}, \text{inIQ}\}, \{(\text{spCA}, \text{diRS}), (\text{diRS}, \text{slPD}), (\text{slPD}, \text{vaPQ}), (\text{vaPQ}, \text{inIQ})\}^*)\} \cup$
(3)	$\{(\{\text{spCA}, \text{diRS}, \text{slPQ}, \text{vaPQ}, \text{diPS}, \text{inCA}\}, \{(\text{spCA}, \text{diRS}), (\text{diRS}, \text{slPQ}), (\text{slPQ}, \text{vaPQ}), (\text{vaPQ}, \text{diPS}), (\text{diPS}, \text{inCA})\}^*)\} \cup$
(4a)	$\left\{\left(\{\text{spCA}, \text{diRS}, \text{slPQ}, \text{vaPQ}, \text{diPS}, \text{paCC}, \text{vaPA}, \text{inPF}\}, \left\{\begin{matrix}(\text{spCA}, \text{diRS}), (\text{diRS}, \text{slPQ}), (\text{slPQ}, \text{vaPQ}), (\text{vaPQ}, \text{diPS}), (\text{diPS}, \text{paCC}), \\ (\text{paCC}, \text{vaPA}), (\text{vaPA}, \text{inPF})\end{matrix}\right\}^*\right)\right\} \cup$
(4b)	$\left\{\left(\{\text{spCA}, \text{diRS}, \text{slPQ}, \text{vaPQ}, \text{diPS}, \text{paCC}, \text{vaPA}, \text{inCO}\}, \left\{\begin{matrix}(\text{spCA}, \text{diRS}), (\text{diRS}, \text{slPQ}), (\text{slPQ}, \text{vaPQ}), (\text{vaPQ}, \text{diPS}), (\text{diPS}, \text{paCC}), \\ (\text{paCC}, \text{vaPA}), (\text{vaPA}, \text{inCO})\end{matrix}\right\}^*\right)\right\} \cup$
(5a)	$\left\{\left(\{\text{spCA}, \text{diRS}, \text{slPQ}, \text{vaPQ}, \text{diPS}, \text{paDB}, \text{vaPA}, \text{inPF}\}, \left\{\begin{matrix}(\text{spCA}, \text{diRS}), (\text{diRS}, \text{slPQ}), (\text{slPQ}, \text{vaPQ}), (\text{vaPQ}, \text{diPS}), (\text{diPS}, \text{paDB}), \\ (\text{paDB}, \text{vaPA}), (\text{vaPA}, \text{inPF})\end{matrix}\right\}^*\right)\right\} \cup$
(5b)	$\left\{\left(\{\text{spCA}, \text{diRS}, \text{slPQ}, \text{vaPQ}, \text{diPS}, \text{paDB}, \text{vaPA}, \text{inCO}\}, \left\{\begin{matrix}(\text{spCA}, \text{diRS}), (\text{diRS}, \text{slPQ}), (\text{slPQ}, \text{vaPQ}), (\text{vaPQ}, \text{diPS}), (\text{diPS}, \text{paDB}), \\ (\text{paDB}, \text{vaPA}), (\text{vaPA}, \text{inCO})\end{matrix}\right\}^*\right)\right\} \cup$

Fig. 14. Set of posets representation of "Order Product" use case

6.2.2. Mapping GTM to sets of posets

This section specifies how a generic task expression is mapped into a corresponding set of posets. As given in Definition 21, $\mathcal{M}_{GteSposet}$ is defined in the common denotational style. An atomic generic task expression (denoted by α) is mapped to a set containing the corresponding singleton poset. Composite task expressions are represented by sets of posets, which are composed using the operators, defined in the Section 6.1.2.

Definition 21 (Set of posets semantics of generic task expressions). Let ψ, ρ be generic task expressions and α be an atomic task. We then define the mapping $\mathcal{M}_{GteSposet}$ to sets of posets as follows:

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

In what follows we illustrate the semantic rules by mapping the "Order Product" generic task expression to a set of posets.

$$\mathcal{M}_{GteSposet}[\![\text{slCR} \gg \text{sbCR} \gg \text{diRS} \gg \big(stop(\text{inCA})\,[\,]\,(\text{slPD} \gg \text{slQT} \gg \text{sbPS})\big) \gg (stop(\text{inIQ})\,[\,]\,\text{diPS}\,) \gg (stop(\text{inCA})[\,]\,\text{paCC}) \gg (stop(\text{inPF})[\,]\,\text{inCO})]\!]$$

According to Definition 21 the application of $\mathcal{M}_{GteSposet}$ to the entire generic task expression is successively broken down into the application of $\mathcal{M}_{GteSposet}$ to sub-expressions and the corresponding set of posets operations. Fig. 15 depicts the resulting set of posets expression, using the same shorthand notation as introduced in the previous section. It allows a subset of the traces allowed by the set of posets of the corresponding use case. Upon initiation, the user searches for a product until he/she either (1, 3) elects to quit, (2) the selected item is out of stock or the product is available and the attempt to pay by credit card is either rejected (4a) or authorized (4b). The option to pay by debit card (as specified in the use case) is not available. This may be due to restrictions of the supported user interface, which may not be equipped with a debit card reader.

(1)	$\{(\{\text{slCR}, \text{sbCR}, \text{diRS}, \text{inCA}, \text{STOP}\}, \{(\text{slCR}, \text{sbCR}), (\text{sbCR}, \text{diRS}), (\text{diRS}, \text{inCA})\}^*)\} \cup$
(2)	$\left\{\left(\{\text{slCR}, \text{sbCR}, \text{diRS}, \text{slPD}, \text{slQT}, \text{sbPS}, \text{inIQ}, \text{STOP}\}, \left\{\begin{matrix}(\text{slCR}, \text{sbCR}), (\text{sbCR}, \text{diRS}), (\text{diRS}, \text{slPD}),\\ (\text{slPD}, \text{slQT}), (\text{slQT}, \text{sbPS}), (\text{sbPS}, \text{inIQ})\end{matrix}\right\}^*\right)\right\} \cup$
(3)	$\left\{\left(\{\text{slCR}, \text{sbCR}, \text{diRS}, \text{slPD}, \text{slQT}, \text{sbPS}, \text{diPS}, \text{STOP}\}, \left\{\begin{matrix}(\text{slCR}, \text{sbCR}), (\text{sbCR}, \text{diRS}), (\text{diRS}, \text{slPD}),\\ (\text{slPD}, \text{slQT}), (\text{slQT}, \text{sbPS}), (\text{sbPS}, \text{diPS})\end{matrix}\right\}^*\right)\right\} \cup$
(4a)	$\left\{\left(\{\text{slCR}, \text{sbCR}, \text{diRS}, \text{slPD}, \text{slQT}, \text{sbPS}, \text{diPS}, \text{paCC}, \text{inPF}, \text{STOP}\}, \left\{\begin{matrix}(\text{slCR}, \text{sbCR}), (\text{sbCR}, \text{diRS}), (\text{diRS}, \text{slPD}), (\text{slPD}, \text{slQT}),\\ (\text{slQT}, \text{sbPS}), (\text{sbPS}, \text{diPS}), (\text{diPS}, \text{paCC}), (\text{paCC}, \text{inPF})\end{matrix}\right\}^*\right)\right\} \cup$
(4b)	$\left\{\left(\{\text{slCR}, \text{sbCR}, \text{diRS}, \text{slPD}, \text{slQT}, \text{sbPS}, \text{diPS}, \text{paCC}, \text{inCO}\}, \left\{\begin{matrix}(\text{slCR}, \text{sbCR}), (\text{sbCR}, \text{diRS}), (\text{diRS}, \text{slPD}), (\text{slPD}, \text{slQT}),\\ (\text{slQT}, \text{sbPS}), (\text{sbPS}, \text{diPS}), (\text{diPS}, \text{paCC}), (\text{paCC}, \text{inCO})\end{matrix}\right\}^*\right)\right\}$

Fig. 15. Set of posets representation of "Order Product" task model

7. Refinement between use case and task models

In the spirit of modern software development, use case and task models are best developed iteratively through a series of refinement steps. For each refinement step it is important to verify that the resulting model is a valid refinement of its source specification. Having defined a common semantics for use case and task models we are now able to formalize refinement between these two kinds of artifact.

In Section 2.3, we noted that use cases are used to capture functional requirements, whereas task models are used to capture UI interaction requirements and design details. While use case models are exclusively used at the requirements stage, task models may be used at the requirements and at the design stage. When the task model is used as a requirements artifact, this detailed specification of the UI is considered part of the contract between stakeholders about the envisioned interactive application, whereas when exclusively used as a design document it is not part of the requirements contract. Based on this observations, it becomes evident that two notions of refinement are necessary depending on the purpose of the refining model; i.e., whether the refining model is a requirements artifact or a design artifact. In the following, both cases are discussed in detail.

At the requirements stage, a use case or task model may be refined by a more detailed artifact. In such a case, the refinement is deemed valid if the refining model does not allow more scenarios than its base model. The refining model, however, may further restrict the set of allowed scenarios. In practice, such a restriction may be the result of filtering the requirements in order to establish a base line. As a consequence, requirements with a low priority or that are considered too risky may be dropped in the refining model. In our semantics, the restriction of scenarios can be expressed in terms of *trace inclusion*.

Definition 22 (Refinement at requirements stage). For $i \in \{1,2\}$, let UCM_i be (well formed) DSRG-style use case models, TM_i be (well formed) *requirements* ECTT task models and P_{UCM_i} and P_{TM_i} be the respective sets of posets representations. We then define refinement between use case and task models as follows:

$$UCM_1 \sqsubseteq UCM_2 \Leftrightarrow Tr(P_{UCM_2}) \subseteq Tr(P_{UCM_1})$$

$$UCM_1 \sqsubseteq TM_2 \Leftrightarrow Tr(P_{TM_2}) \subseteq Tr(P_{UCM_1})$$

$$TM_1 \sqsubseteq TM_2 \Leftrightarrow Tr(P_{TM_2}) \subseteq Tr(P_{TM_1})$$

The artifacts gathered during requirements specification are part of the contract between stakeholders about the envisioned application. When moving from a requirements model to a design model, it is important to ensure that the refining model truly implements the requirements. As a consequence, the refining model may only add information in terms of structural refinement (refinement of previously atomic use case steps or tasks), but must not restrict or extend the number of possible scenarios. For example, when moving from a use case model to a design-level task model we have to ensure that the task model adopts the entirety of the functional requirements specified in the use case model and integrates them into the UI design. Similarly, if a requirements task model is refined by a design-level task model we require that each task of the requirements model be present in the design-level task model and that the execution orders of all "implemented" requirements-level tasks be preserved. In our semantics, scenario equivalence is expressed in terms of *trace equivalence*:

Definition 23 (Refinement at design stage). Let UCM_1 be a (well formed) DSRG-style use case model, TM_{Req_1} be a (well formed) requirements ECTT task model, TM_{Des_1}, TM_{Des_2} be (well formed) *design* ECTT task models and

P_{UCM_1}, $P_{TM_{Req_1}}$, $P_{TM_{Des_1}}$ and $P_{TM_{Des_2}}$ be their respective sets of posets representations. We then define refinement between use case and task model as follows:

$$UCM_1 \sqsubseteq TM_{Des_2} \Leftrightarrow Tr(P_{TM_{Des_2}}) = Tr(P_{UCM_1})$$

$$TM_{Req_1} \sqsubseteq TM_{Des_2} \Leftrightarrow Tr(P_{TM_{Des_2}}) = Tr(P_{TM_{Req_1}})$$

$$TM_{Des_1} \sqsubseteq TM_{Des_2} \Leftrightarrow Tr(P_{TM_{Des_2}}) = Tr(P_{TM_{Des_1}})$$

A precondition for the application of the definition is that the involved sets of posets are defined over the same event-name alphabet. In what follows, we discuss two techniques to resolve alphabet conflicts, called *refinement mapping* and *event hiding*.

1. A mapping from events of the base specification to events of the refining specification is referred to as *refinement mapping*. We distinguish between two main types:
   - **Choice Mapping:** An atomic element of the base specification may be refined by a *set* of atomic elements which are alternatively composed by either the ECTT choice ([ ]) operator or a *Choice* use case step.
   - **Many-To-One mapping:** An atomic element of the base specification may be refined into a set of sub-elements. In contrast to choice refinement, the execution of the entirety of sub-elements resembles the execution of the base element.
2. The second technique that can be used to unify the event-name alphabet of two specifications is *event hiding*. As already mentioned, use case models are used to document functional requirements while task models specify UI requirements and design details. As such, they abstract from internal system actions, which are irrelevant for UI design. Hence, in order to compare use case and task models for refinement, we have to abstract from all internal system steps in the use case model. This is achieved by removing all events that represent internal events in the set of posets representing the use case model.

In order to illustrate the application of the introduced refinement definitions, let us recall the "Order Product" use case and the "Order Product" task model. In order to formally verify that the task model is a valid *requirements*-level refinement of the use case, we need to prove that the set of all traces of the set of posets representing the task model is a subset of the set of all traces of the set of posets representing the use case. After hiding, all events representing internal system steps (displayed in gray in Fig. 14) in the set of posets representing the use case, we obtain the following trace set.

$$Tr(P_{UCM}) = \left\{ \begin{array}{l} \langle \text{spCA, diRS, inCA} \rangle, \langle \text{spCA, diRS, slPQ, inIQ} \rangle, \langle \text{spCA, diRS, slPQ, diPS, inCA} \rangle, \\ \langle \text{spCA, diRS, slPQ, diPS, paCC, inPF} \rangle, \langle \text{spCA, diRS, slPQ, diPS, paCC, inCO} \rangle, \\ \langle \text{spCA, diRS, slPQ, diPS, paDB, inPF} \rangle, \langle \text{spCA, diRS, slPQ, diPS, paDB, inCO} \rangle \end{array} \right\}$$

The set of posets representing the task model, after the refinement mapping, is given in Fig. 16. In particular we applied the many-to-one mapping from tasks "Select Criteria" (slCR) and "Submit Criteria" (sbCR) to use case step "Specify Product Category" (spCA) and the many-to-one mapping from tasks "Select Product" (slPD), "Select Quantity" (slQT) and "Submit" (sbPS) to use case step "Specify Product and Quantity" (slPQ). We then obtain the following set of traces:

$$Tr(P_{TM}) = \left\{ \begin{array}{l} \langle \text{spCA, diRS, inCA} \rangle, \langle \text{spCA, diRS, slPQ, inIQ} \rangle, \langle \text{spCA, diRS, slPQ, diPS, inCA} \rangle, \\ \langle \text{spCA, diRS, slPQ, diPS, paCC, inPF} \rangle, \langle \text{spCA, diRS, slPQ, diPS, paCC, inCO} \rangle \end{array} \right\}$$

Clearly $Tr(P_{TM}) \subseteq Tr(P_{UCM})$, and hence we conclude that at the requirements stage the "Order Product" task model is a valid refinement of the "Order Product" use case. Recall that each task model is geared to a particular user interface and as such is confined by its limitations. In this case, the user interface may not be equipped with a debit card reader and hence does not offer the user this payment option (even though from a pure functionality point of view a debit card payment could have been processed by the system (as specified by the use case). We conclude this section by noting that at the *design stage*, the "Order Product" task model is *not* a valid refinement of the use case. With $Tr(P_{TM}) \neq Tr(P_{UCM})$ the requirements contract (as specified by the use case) is not fulfilled by the design which supports only a subset of the offered functionality.

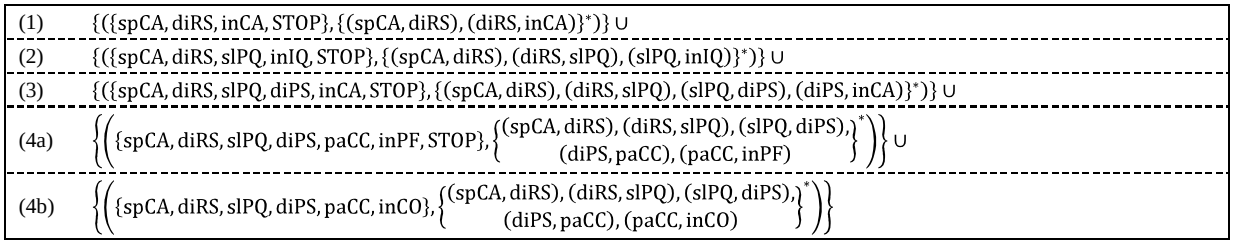

Fig. 16. Set of posets representation of "Order Product" task model *after* refinement mapping

8. Related work

In this paper, we have defined a common formal semantics for use cases and task models. Both are used to model behavioral aspects of the system. The formalization of behavioral specifications has been attempted by various researchers. Börger et al. [BCR00a; BCR00b] propose a formal framework for UML statecharts based on an multi-agent ASM formalism. The behavior of the statechart is controlled by a set of ASM agents, which execute actions depending on the currently active state(s). The actions are formalized by a set of ASM rules. Reggio et al. [RAC00] define an operational semantics for UML statecharts based on algebraic specifications. Pursuing the goal statechart verification Kwon [Kwo00] proposes a formalizations in PROMELA/SPIN and SMV, respectively. Activity diagrams are used to describe the flow of behavior within a system. Similarly to use cases, activity diagrams are equipped with constructs to express *sequences*, *choices* and *parallelism*. Several attempts have been made to define formal semantics for activity diagrams. E.g., the research by Börger et al. [BCR00c] defines a semantics by translating activity diagrams to abstract state machines. Bolton and Davies [BoD00] provide a formalization of activity diagrams using CSP.

UML interaction diagrams (e.g., collaboration/communication and sequence diagrams) are used to model system functionality and the control flow within a system. Engels et al. [EHS99] define a formal semantics for UML collaboration diagrams based on graph transformation rules. Storrle [Sto03] and Haugen et al. [HHR05] define trace-based semantics for sequence diagrams. The semantics proposed by Grosu and Smolka [GrS05] employs safety and liveness properties to formally distinguish between valid and invalid behaviors. For the closely related message sequence charts (MSCs), Zheng [Zhe04], proposes a non-interleaving semantics based on timed labeled partial order sets (lposets). Partial order semantics for (regular, un-timed) MSCs have been proposed by Alur et al. [AHP96] and Katoen and Lambert [KaL98]. 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).

The definition of formal semantics for use case models has been attempted by various researchers. Fröhlich and Link [FrL00] present a transformation algorithm that derives a UML state chart model from a given set of textual use cases. Similar to our approach, a distinction is made between use case steps that are performed by the system and steps performed by the primary actor. The former are represented by actions, whereas the latter are modeled as events, causing state transitions. Övergaard and Palmkvist [ÖvP98] propose an ODAL [MPW92] formalization of use cases and their relationships (*uses* and *extends*). It is assumed that use cases are pre-formalized in a proprietary methods / operations notation. The formalizations of *uses* roughly corresponds to our *include* step, the formalizations of *extends* corresponds to our notion of a use case extension. Stevens [Ste01] discusses how use cases and their relationships may be formalized using labeled transition systems (LTS). Use cases are interpreted as processes, which are internally represented by LTSs. Relationships between use cases are modeled by relating the corresponding LTSs.

Rui *et al.* suggest a process algebraic semantics for use case models, with the overall goal of formalizing use case refactorings [Rui07]. In their approach, scenarios are represented as basic MSCs by partially adapting the ITU MSC semantics [Itu99]. Fernandes *et al.* [FTJ07] present an approach to translate use cases into Colored Petri net models. It is assumed that each use case is represented by a UML sequence diagram. The translation is performed in a top down manner: First, the use case model is mapped into a global Petri net which contains placeholders for each individual use case. Then, each placeholder is replaced by a sub-Petri net capturing the various scenarios of the use case. Probably the most comprehensive approach has been defined by Somé [Som07]. He proposes execution semantics for use cases by defining a set of mapping rules from well-formed use cases to basic Petri nets. A use case is deemed well-formed if it syntactically corresponds to a predefined meta-model and satisfies a set of consistency

and well-formedness rules. The mapping to Petri nets is defined over the various components of the use case (e.g. use case step, extension, control flow construct, etc.).

Paternò and Santoro define formal semantics for a *subset* of the task modeling notation CTT (ConcurTaskTrees) based on LOTOS [PaS03]. Tasks and subtasks from the CTT task model are mapped in a one-to-one fashion to LOTOS process specifications. Temporal relations between tasks are mapped to LOTOS process composition operators. Ait-Ameur *et al.* [ABK05] specify a mapping from CTT to Event-B [Abr96]. Main motivation behind their work is the formal validation of whether a concrete implementation of a UI is consistent with its design specification. Klug & Kangasharju [KlK05] propose a formalization for task models where a task is not regarded as an atomic entity (like in CTT) but has a complex lifecycle, modeled by a so-called task-state machine. In the approach by van den Bergh and Coninx [BeC07] entire task expressions are translated into state charts, including high-level tasks. Bomsdorf [Bom07] defines an elaborated life cycle for tasks. The work is focused on the development of web applications and considers external events related to web technology (e.g. session timeouts and user aborts).

The approach presented in this paper is inspired by the approach by Zheng [Zhe04], who proposed a non-interleaving semantics for timed MSC 2000 [Itu99] based on timed labeled partial order sets (lposets). Compared to the poset definition introduced in this paper, a *timed lposet* additionally contains labeling and timing functions. The labels serve as an indicator for the corresponding event type. Possible event types are: *message input*, *message output*, *internal action*, *start timer*, *stop timer* and *timeout*. Furthermore Zheng defines two functions which attach timing constraints to events in order to specify the time range within which an event could occur and to define delays between two events. The semantic mapping is performed by associating an MSC with a set of timed lposets which capture the possible execution scenarios of the MSC.

According to our integrated development methodology, use case and task models are successively refined into more detailed specifications. Refinement relations for event-based specifications have been investigated for decades and definitions have been proposed for various models [Den87; IYK90; But92; BuB06; SiC07; BSB02]. Khendek et. al [KBV01] propose a refinement relation for basic MSCs. It ensures that a scenario, described in the source MSC specification, is also available in the refined specification. In much the same vein, a scenario that is forbidden in the source specification must never occur (or be derivable) in the refined specification [Li00]. In other words, the behavior of the source MSC must be preserved in the target MSC. Events defined in the source MSC should also occur in the target MSC, and the relative order of these events needs to be preserved. The order of newly introduced events is not restricted. In our work, we used a similar approach by defining refinement through *trace inclusion* and *trace equivalence*. While the former is applied at the requirements level, the latter is used at the design-level to express the condition that the design shall faithfully fulfill the contract statement as laid out by the requirements.

9. Conclusion

The lack of a common formal semantics for use case and task models hinders the effective verification of well-formedness properties, leaves little room for tool support, and hampers the definition of an integrated development methodology. As a consequence, ambiguities and inconsistencies may go undetected, and are likely to propagate in subsequent development stages, resulting in higher costs to repair them. To address these shortcomings, we have defined a common semantics for use case and task models. The formal framework defines a two-step mapping from use case or task model notations to the common semantic domain of sets of posets. Our two-step mapping results in a semantic framework that can be more easily reused and extended. The intermediate semantic domains have been carefully chosen by taking into consideration the intrinsic characteristics of task models and use cases. In particular, we defined a Use Case Labeled Transition System (UC-LTS) as an intermediate semantic domain for use cases. It was demonstrated that UC-LTSs 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 task models, 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.

As a concrete example, we demonstrated how our framework can be used to define a common semantics for DSRG-style use case models and ECTT task models. Both have been defined as improvements to their respective state-of-the-art counterparts, Cockburn-style use case models and CTT. Each improvement has been carefully selected to ensure that the intent and nature of each model is preserved. In the case of DSRG-style use case models, we introduced *step kinds* and *step types* as distinguishing factors for use case steps. In case of ECTT, as our main contribution, we defined two novel temporal operators: *Stop* and *Resume*, that allow the developer to model error and failure cases, and provide a mechanism to *catch* errors and prevent their propagation. Also, in order to overcome the

predominant, yet obsolete, monolithic task-tree structure, we defined ECTT in a modular fashion allowing task models to be developed in a true top-down manner while taking advantage of encapsulation.

The common semantics presented here formally relates use cases and task models and allows for cross-artifact refinement checks. We have defined two refinement relations based on trace inclusion and trace equivalence. The former is used at the requirements level, whereas the latter is used when moving from the requirements to the design level. The presented refinement definitions are one possible utilization of the common formal semantics for use cases and task models. Depending on the usage context, more elaborate notions of refinement (other than trace inclusion or equivalence) can be defined as well. A set of prototypical tools were developed as proofs of concept for the syntactic and semantic definitions. We developed an Isabelle/HOL theory which allows for validating syntactic and well-formedness properties of DSRG-style use cases. We also developed the tool Use Case – Task Model Verifier, which partly implements the semantic mappings to the intermediate semantic domains.

Future avenues deal with the extension of the proposed semantics to capture state information. State information is often employed in use case or task models to express and evaluate conditions. For example, the precondition of a use case denotes the set of states in which the use case is to be executed. In addition, every use case extension is triggered by a condition that must hold before the steps defined in the extension are executed. To be able to evaluate conditions, the semantic model must provide a means to capture the system state and should be able to map state conditions to the occurrence of events.

Acknowledgments

This work has been partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[ABK05] Ait-Ameur, Y., M. Baron and N. Kamel, Encoding a process algebra using the Event B Method. Application to the validation of user interfaces, in Proceedings of *2nd IEEE International Symposium on Leveraging Applications of Formal Methods (ISOLA) 2005*, Columbia, USA, 2005.

[Abr96] Abrial, J. R., Extending B without changing it (for developing distributed systems). in Proceedings of *Putting Into Pratice Methods and Tools for Information System Design*, 1996.

[AHP96] Alur, R., G. J. Holzmann and D. Peled, An Analyzer for Message Sequence Charts., in Software - Concepts and Tools, **17** (**2**), pp. 70-77, 1996.

[AnD67] Annett, J. and K. D. Duncan, Task Analysis and Training Design, in Occupational Psychology, **41**, pp. 211-221, 1967.

[ArM01] Armour, F. and G. Miller, Advanced Use Case Modeling, Addison-Wesley, 2001.

[BaW90] Baeten, J. C. M. and W. P. Weijland, Process algebra, Cambridge University Press, New York, NY, USA, 1990.

[BCR00a] Börger, E., A. Cavarra and E. Riccobene, Modeling the Dynamics of UML, in Proceedings of *ASM'2000*, Switzerland, pp. 223-241, 2000.

[BCR00b] Börger, E., A. Cavarra and E. Riccobene, On formalizing UML state machines using ASMs, in Information and Software Technology, **46** (**5**), pp. 287-292, 2004.

[BCR00c] Börger, E., A. Cavarra and E. Riccobene, An ASM Semantics for UML Activity Diagrams, in Proceedings of *8th International Conference on Algebraic Methodology and Software Technology*, Iowa City, Iowa, USA, pp. 293 - 308, 2000.

[BeC07] van den Bergh, J., Coninx, K. , From Task to Dialog Model in the UML, in Proceedings of *TaMoDia 2007*, Toulouse, France, pp. 98-111, 2007.

[BGK98] Butler, G., P. Grogono and F. Khendek, A Z Specification of Use Cases, in Proceedings of *APSEC 1998*, pp. 94-101, 1998.

[BGS03] Barnett, M., W. Grieskamp, W. Schulte, N. Tillmann and M. Veanes, Validating use-cases with the AsmL test tool, in Proceedings of *Quality Software 2003*, pp. 238-246, 2003.

[Bly75] Blyth, T. S., Set theory and abstract algebra, Longman, London, 1975.

[BoD00] Bolton, C. and J. Davies, Activity graphs and processes, in Proceedings of *Integrated Formal Methods*, Berlin, Germany, pp. 77–96, 2000.

[Bom07] Bomsdorf, B., The WebTaskModel Approach to Web Process Modelling, in Proceedings of *Task Models and Diagrams for User Interface Design* Toulouse, France, pp. 240-253, 2007.

[BSB02] Bowman, H., M. W. A. Steen, E. A. Boiten and J. Derrick, A Formal Framework for Viewpoint Consistency, in Proceedings of *Formal Methods in System Design*, pp. 111-166, September 2002.

[BuB06] Burns, A. and G. Baxter, Time Bands for Systems Structure, chapter in *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, Springer, 2006.

[But92] Butler, M., A CSP Approach to Action Systems, PhD Thesis in *Computing Laboratory*, Oxford University, Oxford, 1992.

[CMN83] Card, S., T. P. Moran and A. Newell, The Psychology of Human Computer Interaction, 1983.

[Coc01] Cockburn, A., Writing Effective Use Cases, Addison-Wesley, Boston, 2001.

[Den87] De Nicola, R., Extensional Equivalences for Transition Systems, in Acta Informatica, **24**, pp. 211-237, 1987.

[DFS04] Dittmar, A., F. Forbrig, S. Stoiber and C. Stary, Tool Support for Task Modelling - A Constructive Exploration, in Proceedings of *Design, Specification and Verification of Interactive Systems 2004*, July 2004.

[DiF03] Dittmar, A. and P. Forbrig, Higher-Order Task Models, in Proceedings of *Design, Specification and Verification of Interactive Systems 2003*, pp. 187-202, 2003.

[EHS99] Engels, G., R. Hücking, S. Sauer and A. Wagner, UML Collaboration Diagrams and Their Transformation to Java in Proceedings of *UIML'99*, Fort Collins, CO, USA, 1999.

[FrL00] Fröhlich, P. and J. Link, Automated Test Case Generation from Dynamic Models, in Proceedings of *ECOOP'00*, Sophia Antipolis and Cannes, France pp. 472-492, 2000.

[FTJ07] Fernandes, J., S. Tjell, J. B. Jorgensen and O. Ribeiro, Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Coloured Petri Net, in Proceedings of *Sixth International Workshop on Scenarios and State Machines (SCESM'07)*, Minneapolis, MN, IEEE Computer Society, 2007.

[GLS01] Grieskamp, W., M. Lepper, W. Schulte and N. Tillmann, Testable Use Cases in the Abstract State Machine Language, in Proceedings of *Second Asia-Pacific Conference on Quality Software*, IEEE Computer Society, 2001.

[GMU07] Hopcroft, J. E., R. Motwani and J. D. Ullman, Introduction to automata theory, languages, and computation, 3rd edition, Pearson/Addison Wesley, 2007.

[Gom05] Gomaa, H., Designing Software Product Lines with UML, Addison-Wesley, 2005.

[GrS05] Grosu, R. and S. A. Smolka, Safety-liveness semantics for UML 2.0 sequence diagrams, in Proceedings of *Fifth International Conference on Application of Concurrency to System Design*, Los Alamitos, CA, USA, pp. 6–14, 2005.

[HHR05] Haugen, Ø., K. E. Husa, R. K. Runde and K. Stølen, STAIRS towards formal design with sequence diagrams., in Software and System Modeling, **4** (**4**), pp. 355-357, 2005.

[Int97] Interactions, I.-I. P. S.-O. S. (1987). ISO 8807: LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.

[Itu99] ITU-T (1999). Recommendation Z.120- Message Sequence Charts. Geneva.

[IYK90] Ichikawa, H., K. Yamanaka and J. Kato, Incremental specification in LOTOS, in Proceedings of *Protocol Specification, Testing and Verification X*, Ottawa, Canada, pp. 183-196, 1990.

[Jac92] Jacobson, I., Object-Oriented Software Engineering: A Use Case Driven Approach, ACM Press (Addison-Wesley Pub), New York, 1992.

[KaL98] Katoen, J. P. and L. Lambert, Pomsets for Message Sequence Charts, in Proceedings of *Formale Beschreibungstechniken für verteilte Systeme*, Cottbus, Germany, Shaker Verlag, pp. 197-207, 1998.

[KBV01] Khendek, F., S. Bourduas and D. Vincent, Stepwise Design with Message Sequence Charts, in Proceedings of *Formal Techniques for Networked and Distributed Systems (FORTE)*, Cheju Island, Korea, pp. 19-34, 2001.

[KlK05] Klug, T. and J. Kangasharju, Executable task models, in Proceedings of *4th international workshop on Task models and diagrams*, Gdansk, Poland, pp. 119-122, 2005.

[Kuu95] Kuutti, K., Activity theory as a potential framework for human-computer interaction research, chapter in *Context and consciousness: activity theory and human-computer interaction*, Massachusetts Institute of Technology, pp**.** 17-44, 1995.

[Kwo00] Kwon, G., Rewrite Rules and Operational Semantics for Model Checking UML Statecharts, in Proceedings of *UML'2000*, York, UK, pp. 528-540, 2000.

[Li00] Li, L., Translating Use Cases to Sequence Diagrams, in Proceedings of *IEEE ASE 2000* Grenoble, France, pp. 293-296, 2000.

[MeB05] Merrick, P. and P. Barrow, The Rationale for OO Associations in Use Case Modelling, in Journal of Object Technology, **4** (**9**), pp. 123-142, 2005.

[Miz07] Mizouni, R., Formal Composition of Partial System Behaviors, PhD Thesis in *Department of Electrical and Computer Engineering*, Concordia University, Montreal, 2007.

[MPW92] Milner, R., J. Parrow and W. D., A Calculus of Mobile Processes, in Information and Computation **100**, pp. 1-40, 1992.

[NPW08] Nipkow, T., L. Paulson and M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Springer, 2008.

[ÖvP98] Övergaard, G. and K. Palmkvist, A Formal Approach to Use Cases and Their Relationships in Proceedings of *UML'98*, Mulhouse, France, 1998.

[PaS01] Paternò, F. and C. Santoro, The ConcurTaskTrees Notation for Task Modelling, Technical Report in CNUCE-C.N.R, 2001.

[PaS03] Paternò, F. and C. Santoro, Support for Reasoning about Interactive Systems through Human–Computer Interaction Designers' Representations, in The Computer Journal, **48** (**4**), pp. 340-357, 2003.

[Pat00] Paternò, F., Model-Based Design and Evaluation of Interactive Applications, Springer, 2000.

[Pre05] Pressman, R. S., Software engineering: A practitioner's approach, McGraw-Hill, Boston, Mass., 2005.

[RAC00] Reggio, G., E. Astesiano, C. Choppy and H. Hußmann, Analysing UML Active Classes and Associated State Machines - A Lightweight Formal Approach, in Proceedings of *Third International Conference on Fundamental Approaches to Software Engineering*, Berlin, Germany, pp. 127-146, 2000.

[Ros05] Roscoe, A. W., The Theory and Practice of Concurrency, Prentice-Hall (Pearson), 2005.

[Rui07] Rui, K., Refactoring Use Case Models, PhD Thesis in *Department of Computer Science and Software Engineering*, Concordia University, Montreal, 2007.

[SCK07] Sinnig, D., P. Chalin and F. Khendek, Common Semantics for Use Cases and Task Models, in Proceedings of *Integrated Formal Methods*, Oxford, England, pp. 579-598, 2007.

[SDM05] Seffah, A., M. C. Desmarais and M. Metzger, Software and Usability Engineering: Prevalent Myths, Obstacles and Integration Avenues, chapter in *Human-Centered Software Engineering -Integrating Usability in the Software Development Lifecycle*, Springer, 2005.

[SiC07] Sinnig, D., P. Chalin and F. Khendek, Consistency between Task Models and Use Cases, in Proceedings of *DSV-IS 2007*, Salamanca, Spain, 2007.

[Sin08] Sinnig, D., Use Case and Task Models: Formal Unification and Integrated Development Methodology, PhD Thesis in *Department of Computer Science and Software Engineering*, Concordia University, Montreal, 2008 (available at http://users.encs.concordia.ca/~d_sinnig/phd/Sinnig_PhDThesis2009.pdf).

[SLV02] 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.

[Som07] Somé, S., Petri Nets Based Formalization of Textual Use Cases, Technical Report in SITE, TR-2007-11, University of Ottawa, 2007.

[Ste01] Stevens, P., On Use Cases and Their Relationships in the Unified Modelling Language, in Proceedings of *4th International Conference on Fundamental Approaches to Software Engineering*, pp. 140-155, 2001.

[Sto03] Storrle, H., Semantics of interactions in UML 2.0, in Proceedings of *Symposium on Human Centric Computing Languages and Environments*, Los Alamitos, CA, USA, pp. 129-136, 2003.

[SWF07] Sinnig, D., M. Wurdel, P. Forbrig, P. Chalin and F. Khendek, Practical Extensions for Task Models in Proceedings of *TaMoDia '07*, Toulouse, France Springer, 2007.

[Zhe04] Zheng, T., Validation and Refinement of Timed MSC Specifications, PhD Thesis in *Department of Electrical and Computer Engineering*, Concordia University, Montreal, 2004.

A. Rewriting of *disabling* and *suspend / resume*

In this section we give formal definitions of the auxiliary operators *deep optionalization* ($\mathcal{O}$) and *interleaved insertion* ($\mathcal{I}$). Both are needed in Definition 10 for the rewriting of the ECTT operators *disabling* und *suspend/resume*, respectively. Intuitively the meaning of the *disabling* operator is defined as follows: Both tasks specified by its operands are enabled concurrently. As soon as the first (sub-) task specified by the second operand is executed, the task specified by the first operand becomes disabled. If the execution of the task(s) specified by the first operand is completed (without interruption) the task(s) specified by the second operand are subsequently executed. In other words, none of the (sub-) tasks of the first operand must necessarily be executed, whereas the execution of the tasks of the second operand is mandatory. Hence, an ECTT task expression including the *disabling* operator can be rewritten as the optional execution of the *deep optionalization* ($\mathcal{O}$) of all tasks involved in the first operand, followed by the execution of the second operand ($\upsilon \; [> \; f = \mathcal{O}[\![\upsilon]\!]_{\mathcal{D}} \gg f$). We note that the definition of the CTT *disabling* operator has been inspired by the disabling operator of the LOTOS process algebra [Int97]. Yet, the interpretations of both operators are *not* identical. In particular, in LOTOS the subsequent execution of the second operand, after completion of the first one is not allowed.

The interpretation of the *suspend/resume* operator is similar to the one of the *disabling* operator. Both tasks specified by its operands are enabled concurrently. At any time the execution of the first operand can be interrupted by the execution of the first (sub-) task of the second operand. An exception to this rule are tasks within the scope of the concurrency operator ($\|$). Such tasks, although interrupted, may (concurrently) continue their execution. Contrary to the *disabling* operator, the execution of the task specified by the first operand is only suspended and will (once the execution of the second operand is complete) be reactivated from the state reached before the interruption [Pat00]. 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. In order to model this behavior, we have defined the auxiliary binary operator *interleaved insertion* ($\mathcal{I}$). It "injects" the task specified by its second operand at any possible position in-between the (sub-) tasks of the first operand. Using the auxiliary operator it is now possible to rewrite a term containing the *suspend/resume* operator as follows: $\upsilon \; | > \; \varphi = \mathcal{I}[\![\upsilon]\!]_{\mathcal{D}} \; \varphi^*$.

Definition 24 (Deep optionalization and interleaved insertion). Let υ, φ, u be ECTT task expressions, n be a task identifier and $\mathcal{D}$ be a finite map of ECTT task definitions. We then define the operators *deep optionalization* ($\mathcal{O}$) and *interleaved insertion* ($\mathcal{I}$) inductively as follows:

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

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

$$\mathcal{O}[\![\upsilon \gg \varphi]\!]_{\mathcal{D}} = \left[\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}} \gg [\mathcal{O}[\![\varphi]\!]_{\mathcal{D}}]\right] \qquad \mathcal{I}[\![\upsilon \gg \varphi]\!]_{\mathcal{D}}\ u = \mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u \gg \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ u$$

$$\mathcal{O}[\![\upsilon\ [\ ]\ \varphi]\!]_{\mathcal{D}} = [\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}\ [\ ]\ \mathcal{O}[\![\varphi]\!]_{\mathcal{D}}] \qquad \mathcal{I}[\![\upsilon\ [\ ]\ \varphi]\!]_{\mathcal{D}}\ u = \mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u\ [\ ]\ \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ u$$

$$\mathcal{O}[\![\upsilon\ \|\ \varphi]\!]_{\mathcal{D}} = [\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}]\ \|\ [\mathcal{O}[\![\varphi]\!]_{\mathcal{D}}] \qquad \mathcal{I}[\![\upsilon\ \|\ \varphi]\!]_{\mathcal{D}}\ u = \mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u\ \|\ \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ u$$

$$\mathcal{O}[\![\upsilon \boxplus \varphi]\!]_{\mathcal{D}} = \mathcal{O}[\![\upsilon \gg \varphi]\!]_{\mathcal{D}}\ [\ ]\ \mathcal{O}[\![\varphi \gg \upsilon]\!]_{\mathcal{D}} \qquad \mathcal{I}[\![\upsilon \boxplus \varphi]\!]_{\mathcal{D}}\ u = \mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u \gg \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ u\ [\ ]\ \mathcal{I}[\![\varphi]\!]_{\mathcal{D}}\ u \gg \mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u$$

$$\mathcal{O}[\![\upsilon[> \varphi]\!]_{\mathcal{D}} = \mathcal{O}[\![[\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}] \gg \varphi]\!]_{\mathcal{D}} \qquad \mathcal{I}[\![\upsilon >\ \varphi]\!]_{\mathcal{D}}\ u = \mathcal{I}[\![[\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}] \gg \varphi]\!]_{\mathcal{D}}\ u$$

$$\mathcal{O}[\![\upsilon \mid > \varphi]\!]_{\mathcal{D}} = \mathcal{O}[\![\mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ \varphi^*]\!]_{\mathcal{D}} \qquad \mathcal{I}[\![\upsilon \mid >\ \varphi]\!]_{\mathcal{D}}\ u = \mathcal{I}[\![\mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ \varphi^*]\!]_{\mathcal{D}}\ u$$

$$\mathcal{O}[\![[\upsilon]]\!]_{\mathcal{D}} = [\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}] \qquad \mathcal{I}[\![[\upsilon]]\!]_{\mathcal{D}}\ u = [\mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u]$$

$$\mathcal{O}[\![\upsilon^*]\!]_{\mathcal{D}} = \upsilon^* \gg [\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}] \qquad \mathcal{I}[\![\upsilon^*]\!]_{\mathcal{D}}\ u = (\mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u)^*$$

$$\mathcal{O}[\![\upsilon^+]\!]_{\mathcal{D}} = \upsilon^* \gg [\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}] \qquad \mathcal{I}[\![\upsilon^+]\!]_{\mathcal{D}}\ u = (\mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u)^+$$

$$\mathcal{O}[\![stop(\upsilon)]\!]_{\mathcal{D}} = stop(\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}) \qquad \mathcal{I}[\![stop(\upsilon)]\!]_{\mathcal{D}}\ u = stop(\mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u)$$

$$\mathcal{O}[\![resume(\upsilon)]\!]_{\mathcal{D}} = resume(\mathcal{O}[\![\upsilon]\!]_{\mathcal{D}}) \qquad \mathcal{I}([\![resume(\upsilon)]\!]_{\mathcal{D}}\ u) = resume(\mathcal{I}[\![\upsilon]\!]_{\mathcal{D}}\ u)$$

B. Trace properties

Based on the definitions of Section 6.1.2, we derive the following trace properties for sets of posets operations.

Proposition (Trace properties of sets of posets operations). The sets of posets operations: *sequential composition* ($\cdot$), *parallel composition* ($\|$), *alternative composition*(#), *closure* (*), *close* and *open* have the following trace properties (Table 2):

Table 2. Trace properties for sets of posets operations

Operation	Trace Property
$P \cdot R$	$Tr(P \cdot R) = \{x^{\wedge}y \mid x \in Tr(p) \ \wedge y \in Tr(R) \wedge p = (E_p, \leq_p) \in P \wedge STOP \notin E_p\} \cup \{x \mid x \in Tr(p) \wedge p = (E_p, \leq_p) \in P \wedge STOP \in E_p\}$, where ^ denotes the sequential composition of two event name sequences.
$P \parallel R$	$Tr(P \parallel R) = \bigcup\{x \mid\mid\mid y \mid x \in Tr(P) \ \wedge y \in Tr(R)\}$, where $\mid\mid\mid$ is defined as [Ros05]: $\langle\ \rangle \mid\mid\mid s = \{s\}$ $s \mid\mid\mid \langle\ \rangle = \{s\}$ $\langle a\rangle^{\wedge}s \mid\mid\mid \langle b\rangle^{\wedge}t = \{\langle a\rangle^{\wedge}u \mid u \in (s \mid\mid\mid \langle b\rangle^{\wedge}t)\} \cup \{\langle b\rangle^{\wedge}u \mid u \in (\langle a\rangle^{\wedge}s \mid\mid\mid t)\}$
$P \# R$	$Tr(P \# R) = Tr(P) \cup Tr(Q)$
P^*	$Tr(P^*) = \bigcup_{k=0}^{\infty} Tr(P^k)$, where $Tr(P^k)$ is defined as: $Tr(P^k) = \begin{cases} \{\langle\ \rangle\}, & k = 0 \\ Tr(P), & k = 1 \\ \{x^{\wedge}y \mid x \in Tr(p) \wedge p = (E_p, \leq_p) \in P \wedge STOP \notin E_p \wedge y \in Tr(P^{k-1})\}, & k > 1 \end{cases}$
$close(P)$	$Tr(close(P)) = Tr(P)$
$open(P)$	$Tr(open(P)) = Tr(P)$

The corresponding proof for the proposition is given in [Sin08].