LTS Semantics for Use Case Models

Daniel Sinnig Patrice Chalin Ferhat Khendek
Faculty of Engineering and Computer Science
Concordia University
Montreal, Quebec, Canada
{d_sinnig, chalin, khendek}@encs.concordia.ca

ABSTRACT
Formalization is a necessary precondition for the specification of precise and unambiguous use case models, which serve as reference points for the design and implementation of software systems. In this paper, we define a formal semantics for use case models. We build on an abstract syntax definition formalizing the sequencing of use case steps. As a semantic domain we have chosen Labeled Transition Systems (LTSs), which, we believe, intuitively capture the behavioral aspects of the use case model. The mapping into LTSs is defined over the various structural elements of the use case model. The proposed formal semantics allows for various semantic checks such as detection of livelocks and validation of model refinement, an important property in an iterative software development lifecycle. We also introduce our tool "Use Case Model Analyzer".

Categories and Subject Descriptors
D.2.1 [**Software Engineering**]: Requirements/Specifications – *languages, methodologies, tools.*

Keywords
Requirements, Use Cases, Semantics, LTSs

1. INTRODUCTION
Use cases were introduced in the early 90s by Jacobson [1]. 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 [2]). There is accumulating evidence of significant benefits to customers and developers [3].

In current practice use cases are written in prose [4]. While the use of natural language makes use case modeling an attractive tool for facilitating communication among stakeholders, it has been recognized that a certain level of formalization is needed to effectively analyze, process and validate use case models [5]. To date, a formal (and agreed upon) semantics for the use case model does not exist. The implications of this lack of formalization are two-fold: (1) The informal nature of the use case model makes it prone to ambiguities and inconsistencies. (2) The lack of a formal semantics leaves little room for tool support to assist the developer in use case authoring, refinement, verification and transformation.

In this paper we address this shortcoming by proposing a formal semantics for the use case model based on the labeled transition systems (LTSs) formalism [6]. Prerequisite is a formal syntax, which needs to be carefully chosen in order to preserve the intuitive nature of use cases, yet enforce a formal structure needed for the semantic mapping. The formal semantics forms the essential basis for performing important semantic checks such as livelock detection and the verification of refinement. The latter plays a key role in each iterative development lifecycle according to which a base use case model is successively refined by more detailed use case models.

The remainder of this paper is structured as follows: In the next section we introduce the running example, which will be used throughout this paper. Section 3 defines an abstract syntax for use case models. Section 4, the core of this paper, entails the semantic mapping to LTS. In Section 5 we introduce our tool, the *Use Case Model Analyzer* and present a set of semantic checks. Section 6 reviews relevant related work while Section 7 concludes this article and presents an outlook to future work.

Use case: Order Product
Properties
Goal: Customer places an order for a specific product.
Level: User-goal
…
Main Success Scenario
1. Customer specifies desired product category. [spCA]
2. System displays search results. [diSR]
3. Customer selects a product. [slPQ]
4. System validates availability of desired product. [vaPQ]
5. System displays purchase summary. [diPS]
6. Customer submits payment info. [sbPI]
7. System carries out payment. [caPA]
8. System provides a confirmation number. [prCN]
9. *Use case ends successfully*

Extensions
3a. Customer is not satisfied with the search results:
3a1. Customer indicates to repeat product search. [inRS]
3a2. *Use case resumes at step 1.*
4a. The desired product is unavailable:
4a1. System informs Customer that product is unavailable. [inPU]
4a2. *Use case ends unsuccessfully.*
6a. Customer decides to cancel 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 declined. [inPD]
7a2. *Use case resumes at step 7.*

Figure 1. Order Product Use Case

2. RUNNING EXAMPLE
A *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. Every use case starts with a header section containing various properties

SAC'09, March 8-12, 2009, Honolulu, Hawaii, U.S.A.

(e.g. primary actor, goal, goal level, etc). The core part of a use case is its main success scenario. 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.

An example use case is given in Figure 1 . The use case captures the interactions for the "Order Product" functionality of an Invoice Management System (IMS). For the sake of brevity, only the main success scenario and the extensions are shown in full, the header section of the use case is partly omitted. The main success scenario of the use case describes the situation in which the primary actor directly accomplishes his/her goal of ordering a product. The extensions specify alternative scenarios which may (4a, 6a) or may not (3a, 7a) 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.

3. ABSTRACT SYNTAX

Different notations for expressing use cases possessing different degrees of formality have been suggested. The extremes range from purely textual constructs written in prose [4] to entirely formal specifications written in Z [7], as Abstract State Machines (ASM) [8; 9], or as graph structures [10]. We adopt an intermediate solution which enforces a formal structure but also preserves the intuitive nature of use cases, i.e., we provide support for formalizing the sequencing of use case steps and their types, but the respective actions, as well as the associated conditions are specified informally. The property section of the use case, except for the discrete goal-level property, is specified using narrative language.

```
theory uc
imports Main begin
datatype GoalLvlProp = SUMMARY | USERGOAL | SUBFUNCTION
datatype StepType =  SYSTEM | INTERACTION | INTERNAL

record UCProperties = Goal :: GoalProperty
                      PrimaryActor :: ActorProperty
                      GoalLevel :: GoalLvlProp
                      Precondition :: PrecondProperty
datatype Step =
  Atom StepID StepType Label "ExtensionID set"|
  Choice StepID "(Step list) list"  "ExtensionID set" |
  Concurrent StepID "(Step list) list" "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"
```

Figure 2. Abstract Syntax for Use Cases

Figure 2 depicts the abstract syntax definition for use cases. It is represented as an Isabelle/HOL [11] theory which allowed us to use the Isabelle theorem prover to verify basic well-formedness properties such as syntax and type checking.

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. A use case extension is defined as a record consisting of an identifier, a condition, and a list of use case steps.

We distinguish between six different kinds of steps. *Atom* denotes an atomic step and defines an action performed by an actor or the system. *Choice* is a complex use case step, which provides the primary actor with a list of alternative step sequences, among which one alternative must be chosen. Similarly, *Concurrent* is also a complex step, whose sub-steps may be performed in any order. *Goto* denotes a branch to another use case step, whereas *Include* denotes the inclusion of a sub-use case. *Success* and *Failure* denote the successful and unsuccessful termination of a use case scenario, respectively. *Atom* steps can be of three different types: Steps of type *interaction* (e.g. step 1 in Figure 1) are performed by the primary actor, whereas steps of types *application* (e.g. step 2) and *internal* (e.g. step 4) 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.

In order to illustrate the definition of a use case, let us reconsider the previously depicted "Order a Product" use case. Figure 3 portrays parts of its formalization in Isabelle/HOL. For the sake of conciseness, only the main success scenario and extension $e3a$ are shown in full. The properties section and the remaining extensions have been omitted. Furthermore, for each *Atom* step, instead of the full description, the abbreviating label has been used.

```
OrderProduct :: UseCase
 …
 MainSuccessScenario = [
  Atom ''s1'' INTERACTION ''spCA'' {},
  Atom ''s2'' APPLICATION ''diSR'' {},
  Atom ''s3'' INTERACTION ''slPQ'' {''e3a''},
  Atom ''s4'' INTERNAL    ''vaPQ'' {''e4a''},
  Atom ''s5'' APPLICATION ''diPS'' {},
  Atom ''s6'' INTERACTION ''sbPI'' {''e6a''},
  Atom ''s7'' INTERNAL    ''CaPA'' {''e7a''},
  Atom ''s8'' APPLICATION ''prCN'' {},
  Success ],
 Extensions =
  {(*Extension e1*) (|
   ID = ''e3a'',
   Condition = ''Customer is not satisfied with
                 search results'',
   ExtensionScenario = [
    Atom ''s3a1'' INTERACTION  ''inRS'' {},
    Goto ''s1''  ]          |),
   (*Extension e4a, e6a, e7a omitted)
```

Figure 3. "Order Product" UC in Abstract Syntax

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

Definition 1. Use Case Model: A *use case model* D is a tuple $D = (n_0, \mathcal{U})$ where,

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

$\mathcal{U} \in UCName \xrightarrow{\text{fin}} UseCase$ is a finite map of use case names to use case definitions such that $n_0 \in dom(\mathcal{U})$.

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. (4) The last element of every use case step sequence be either $Goto$, $Success$, or $Failure$.

4. FORMAL SEMANTICS

The semantic domain for use case models is use case labeled transitions systems (UC-LTSs) which is defined as follows:

Definition 2. 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 $\delta: (Q \times \Sigma) \longrightarrow \mathbb{P}(Q)$ is the transition function.

We believe that UC-LTS has been defined in a manner which easily and intuitively captures the nature of use cases, as we explain next. A use case primarily describes the possible execution order of user and system actions in the form of use case steps: from a given state, the execution of a step leads into another state. Accordingly, in UC-LTSs, the execution of a step is denoted by a transition from a source state to a target state. Each transition is associated with a label, 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.

The mapping from a use case model into a UC-LTS is done in two steps:

1. Generation: For each use case of the use case model, the main success scenario and extensions are mapped into 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 ($\sim$) by combining equivalent states.

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 3. Canonical Form of a Use Case Model: A use case model is in canonical form, if it is well formed and if the following conditions are satisfied:

1. Each use case extension is associated with exactly one step.
2. Each use case (except for the root use case) is invoked by exactly one $Include$ step.

While the "Order Product" use case (Figure 1) 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 (1), instead of the original extension, use case steps are associated with distinct copies of the respective extensions. If steps of the original extension are referenced by means of a $Goto$ step, the respective reference is to be updated accordingly. Similarly, in order to satisfy condition (2) instead of the original sub-use case n, each $Include$ 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 Figure 4, env has three fields, named uc, ext and $step$, where:

- uc is a function that maps use case names to $UCStateInfo$ which, according to Figure 4, 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 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 step and extension ids are unique within any given use case model.

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

Figure 4. Global Environment env and UCStateInfo

$\sim \subseteq STATE \times STATE$ 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: $\sim = \{(q, q) \mid q \in STATE\}$.

4.1 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 (Figure 2), there are 6 kinds of use case steps. Each step kind has its own specific mapping to a UC-LTS.

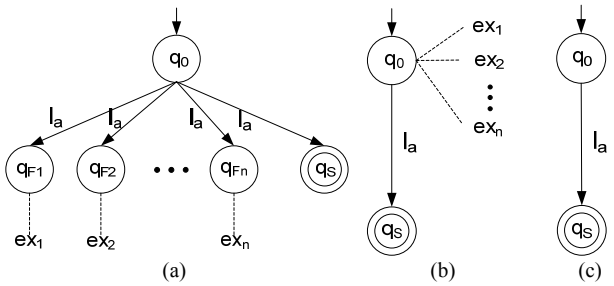

Figure 5. Semantic Mapping of Atomic UC Steps to UC-LTS

As depicted in Figure 5, 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 ($\sim$)[1].

In contrast to internal use case steps, which are performed by the systems and are hidden from the user, steps of type *interaction* are performed by the user. As such, they do not have an alternative outcome per se, but may be associated (by virtue of one or more extensions) with alternative steps which are performed instead of the actual step. As a result, the corresponding UC-LTS consists of only one transition (from q_0 to q_s), representing the use case step (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 [12]. 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.

Having defined the mapping for individual UC steps, we continue with defining the mapping of step sequences to UC-LTS.

Definition 4. 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:

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

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 Figure 6, the sequential composition consists of unifying the final states of the first operand and the initial state of the second operand.

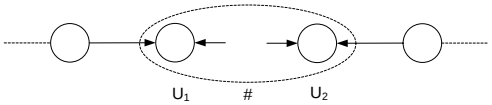

Figure 6. Sequential Composition of UC-LTSs

[1] $\sim := \left(\sim \cup \left\{\left(q_{F_1}, q_{0_{ex_1}}\right), \left(q_{F_2}, q_{0_{ex_2}}\right), \dots, \left(q_{F_n}, q_{0_{ex_n}}\right)\right\}\right)^{*1}$ with $q_{0_{ex_i}} = env.ext(id_{ex_i}, id)$

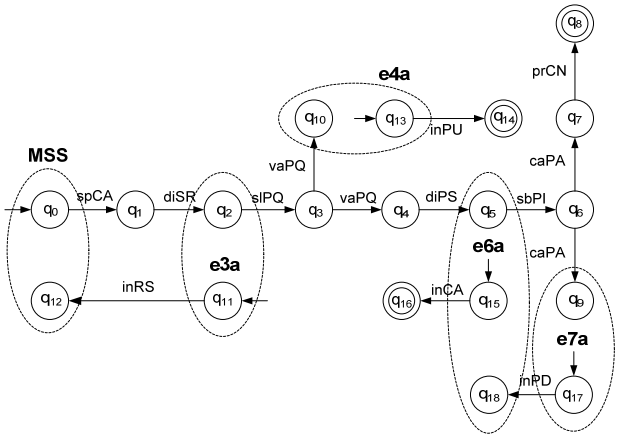

Figure 7. UC-LTSs of the "Order Product" Use Case

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 5. 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 $ex_i = (id_i, condition_i, s_i)$. We then obtain the corresponding set of UC-LTSs as follows:

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

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 6. 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, Figure 7 portrays the set of UC-LTSs of the IMS use case model. (For the sake of simplicity, we assume that the use case model consists of only one use case; the aforementioned "Order Product" use case.) 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.

4.2 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 ($\sim$).

Definition 7. 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, and $\{U_1, U_2, \dots, U_n\}$ be the result of $\mathcal{M}_{Ucs}[\![\{uc_1, uc_2, \dots, uc_m\}]\!]$ with $U_i = (\Sigma_i, Q_i, q_{0_i}, F_i, \delta_i, \tau_i)$ and $n \geq m$. The mapping to UC-LTS is then defined as follows:

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

$$\Sigma = \Sigma_1 \cup \Sigma_2 \cup \dots \cup \Sigma_n$$
$$Q = (Q_1 \cup Q_2 \cup \dots \cup Q_n)/\sim$$
$$q_0 = [env.uc(n_0).q_0]$$

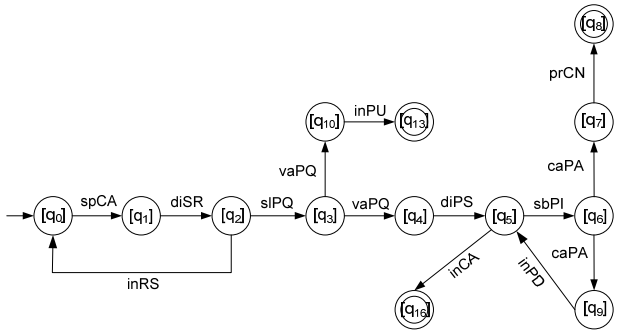

Figure 8. UC-LTS Representing the "Order Product" UC

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

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

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

Figure 8 portrays the UC-LTS obtained by merging the various UC-LTSs given of Figure 7. The resulting UC-LTS has three final states: $[q_8]$ denoting the successful outcome of the use case (i.e., the customer succeeded to order the product), $[q_{16}]$ denoting the case where the user cancels the use case, and $[q_{13}]$ denoting the case where the product is not available. Notice that the resulting UC-LTS has fewer states than the accumulated number of states of the involved UC-LTSs. This is because, during merging, two or more states are combined into a single state, representing the respective equivalence class.

5. SEMANTIC CHECKS

Based on the semantics above, we have developed the "Use Case Analyzer" tool. It supports the developer in formally validating and comparing use case models. In this section we discuss two such semantic checks supported by our tool, namely *livelock detection* and verification of *refinement* between use case models.

A *livelock* is a phenomenon, where an application performs an infinite sequence of internal actions. According to [13], a live-locked system is highly undesirable from the user's point of view since the user may be able to observe the presence of internal activity and hence hope "eternally" that some output will emerge eventually. We believe that some livelocks are (partly) induced due to erroneous use case specifications, which contain loops of internal *system* steps. Clearly, as a requirements specification, the use case model serves as direct input to the subsequent development phases, and it is likely that the livelock will propagate to the implementation stage.

Within the domain of LTSs, livelock is formally known as divergence [13]. Several algorithms for divergence detection have been proposed [14; 15; 16]. In our tool "Use Case Analyzer" we implemented our own version of livelock detection, which is similar to the algorithm used in the CSP model checker FDR [16], but has been customized to fit better our application domain. The main idea is as follows: We successively traverse all states of the UC-LTS and check if they are divergent. A state q is divergent if and only if the directed graph formed by considering only transitions triggered by events representing internal use case steps has a cycle reachable from q.

Another important semantic check is the verification of *refinement* between two use case models. In modern software engineering, the development lifecycle is divided into a series of iterations. Within each iteration, a set of disciplines and associated activities are performed while the resulting artifacts are incrementally perfected and refined. The development of the use case model is no exception to this rule. Ongoing prioritization and filtering activities during the early stages of development will gradually refine the requirements captured in the use case model. It is important to ensure that the refining use case model is a proper refinement of its relative base model (and all its predecessor models).

For this purpose, we have implemented two different refinement checks in our tool: *Equivalence* and *Deterministic Reduction*. The former verifies whether a refining use case model has the same traces *and* exposes the same non-determinism. The latter allows the refining model to have a subset of the traces of the base model but requires that existing nondeterministic transitions are preserved. If we translate this condition back to the domain of use cases, we allow the refining use case model to offer the user fewer options to choose from (e.g. as a consequence of requirements filtering in order to establish a base line), while the system output in response to a given user interaction (modeled through nondeterministic transitions) needs to be preserved.

In order to verify the aforementioned refinement criteria we implemented a model-checking algorithm. Similar to FDR [13] the respective LTSs are first normalized by transforming them into acceptance graphs. The refinement is proven by pairwise comparing the acceptance sets of "trace equivalent" states of the base and refining model. In case of *equivalence*, we required that the acceptance sets be identical, while in case of *deterministic reduction* we require that the respective acceptance sets have the same cardinality. A formalization of both criteria is given in [12].

6. RELATED WORK

The work reported in this paper was conducted in the context of a larger project involving the formalization of use case models and task models [17; 18]. One of the outcomes of our earlier work which supported the research done in the context of this paper was the creation of a meta-model for use case documentation similar to the abstract syntax definition presented in Section 2. Preliminary results towards the definition of a formal semantics for use cases were reported in [17], where we proposed an nondeterministic finite state machine (nFSM) formalization for the use case and task models for the purpose of formally verifying that a given task model is consistent with a respective use case specification.

In the domain of use case authoring, a number of guidelines have been proposed to provide some degree of formalization to textual use cases. Most notable are the CREWS Guidelines [19] and the CS Rules [20]. The emphasis, however, is on the use of natural language and the structure within use cases rather than on the definition of execution semantics. This shortcoming was partly addressed by the work of Somé [5]. In pursuit of the overall goal of automatically generating test cases, he defines a mapping from use cases to basic Petri nets. Fröhlich and Link [21] 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. In both cases, the focus is on the derivation of test cases from the use case model, while in our work the focus is the semantic analysis of well-formedness properties and verification of refinement.

Li [22] describes a translation method that details how a "normalized" use case is translated into UML Sequence Diagrams. Each use case step is represented by a corresponding message in the sequence diagram. Sinha et al. [23] present an approach which generates EFSM models from textual use case specifications. Use cases are assumed to be UML 2.0 compliant and to possess a formal structure. Each use case step belongs to a predefined category (e.g., output statement, query statement, update statement, etc.). In the approach by Mizouni et al. [10], use case automata (UCA) are used to represent the behavioral aspects of individual use cases. Since each use case is only a partial specification of the system, the authors define a set of composition operators for UCAs.

7. CONCLUSION

In this paper we proposed a formalization of the use case model. The main motivation behind our research was the fact that in current practice use cases are used informally. The absence of a formal semantics hinders the effective verification of well-formedness properties and leaves little room for tool support. As a consequence, ambiguities and inconsistencies in the use case model may go undetected and are likely to propagate in subsequent development stages, resulting in higher costs for repair.

To address this shortcoming, we defined a formal semantics for the use case model. In order to preserve the intuitive nature of use cases, we have carefully chosen a syntax definition that formalizes the sequencing of use case steps and their types while the respective actions and associated conditions remain in prose format. By taking into account the intrinsic characteristics of use cases, we have selected UC-LTSs as semantic domain, which allow for a natural representation of the order in which actions are to be performed. Moreover, UC-LTSs allow us to make a distinction between user and system choices by using deterministic and non-deterministic transitions, respectively. Based on the formal semantics we outlined how important semantic checks, such as livelock detection or refinement verification, are formalized. We briefly introduced our "Use Case Model Analyzer" tool, which assists the developer in the automatic verification of the aforementioned properties.

The work reported in this paper was conducted in the context of a larger project involving the integrated formalization of functional and non-functional requirements. Future avenues deal with the extension of the "Use Case Model Analyzer" to offer additional functionality such as deadlock detection and use case refactoring. We plan to extend the proposed semantics to capture state information using Kripke structures. State information is often employed in a use case to express and evaluate conditions. For example, every use case extension is triggered by a condition that must hold before the steps defined in the extension are executed. The ultimate goal of our research is to effectively use the proposed formalization within an MDD process, according to which the use case model and other requirements models are automatically transformed into design-level models.

8. REFERENCES

[1] Jacobson, I., Object-Oriented Software Engineering : A Use Case Driven Approach, ACM Press, New York, 1992.

[2] Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), Prentice Hall PTR, 2004.

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

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

[5] Somé, S., Petri Nets Based Formalization of Textual Use Cases, Tech. Report in SITE, TR2007-11, Uni. of Ottawa, 2007.

[6] Keller, R., Formal Verification of Parallel Programs, Communications of the ACM, 19, pp. 561-572, 1976.

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

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

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

[10] Mizouni, R., Salah, A., Kolahi, S. and Dssouli, R., Merging partial system behaviours: composition of use-case automata, in Software, IET, 1(4), pp. 143-160, 2007.

[11] Nipkow, T., Paulson, L. and Wenzel, M., Isabelle/HOL: A Proof Assistant for Higher Order Logic, Springer, 1998.

[12] Sinnig, D., Use Case and Task Models: Formal Framework and Integrated Development Methodology, PhD Thesis in *Department of Computer Science and Software Engineering*, Concordia University, Montreal, 2008.

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

[14] CADP, Construction and Analysis of Distributed Processes - Software Tools for Designing Reliable Protocols and Systems [Internet], Available from http://www.inrialpes.fr/vasy/cadp/, Accessed: May 2007, Last Update: 2007.

[15] Magee, J. and Kramer, J., Concurrency: State Models & Java Programs, John Wiley & Sons, 1999.

[16] Roscoe, A. W., Model-checking CSP, chapter in *A Classical Mind: essays in Honour of C.A.R. Hoare*, Prentice-Hall.

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

[18] Sinnig, D., Chalin, P. and Khendek, F., Towards a Common Semantic Foundation for Use Cases and Task Models, in ENTCS, 183C, 2006.

[19] Salinesi, C., Authoring Use Cases, chapter in *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*. I. Alexander and N. Maiden, John Wiley.

[20] Phalp, K., Vincent, J. and Cox, K., Improving the Quality of Use Case Descriptions: Empirical Assessment of Writing Guidelines, in Software Quality, 2007.

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

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

[23] Sinha, A., Paradkar, A. and Williams, C., On Generating EFSM Models from Use Cases, in Proceedings of *SCESM 2007*, Minneapolis, MN, 2007.