Verifying Temporal Properties of Use-Cases in Natural Language

Viliam Simko[1], David Hauzar[1], Tomas Bures[1,2],
Petr Hnetynka[1], and Frantisek Plasil[1,2]

{simko, hauzar, bures, hnetynka, plasil}@d3s.mff.cuni.cz
[1]Department of Distributed and Dependable Systems,
Faculty of Mathematics and Physics,
Charles University, Malostranské náměstí 25,
Prague 1, 118 00, Czech Republic
[2]Institute of Computer Science,
Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, Prague 8, 182 07, Czech Republic

Abstract. This paper presents a semi-automated method that helps iteratively write use-cases in natural language and verify consistency of behavior encoded within them. In particular, this is beneficial when the use-cases are created simultaneously by multiple developers. The proposed method allows verifying the consistency of textual use-case specification by employing annotations in use-case steps that are transformed into temporal logic formulae and verified within a formal behavior model. A supporting tool for plain English use-case analysis is currently being enhanced by integrating the verification algorithm proposed in the paper.

Keywords: Use-Cases, Behavior Modeling, Verification, Natural Language, Label Transition System, Model-Checking, Requirements Engineering, Temporal Logic

1 Introduction

In typical software development practice, majority of the requirement documents created in the early phase of a project, are written in natural language [10]. Such a specification is therefore inherently imprecise, ambiguous, and a potential source of contradictions. An important issue is that in a large software project, the specification phase involves collaboration among a number of team members[1] who express their personal views in natural language. In such an environment, there is a high chance of conflicts among individual parts of the specification.

Use-cases are traditionally used in requirement specification because they can easily capture the behavior of a system under discussion (SuD) from the perspective of different actors. Usually, SuD may be equalled to a component where a use-case describes a part of the interaction between the component and its environment.

[1] For example, the Agile software development methodology proposes teams of 5-9 people.

Since the inclusion of use-cases into the UML standard [14], their use has been greatly extended, making them a mandatory requirement for any object-oriented software development project. As stressed by Cockburn [3] and Larman [9], the main asset of use-cases is that behavior is encoded in natural language and thus accessible to a wide range of stakeholders of a project.

Although an isolated use-case can clearly describe a simple scenario, the overall behavior of combined use-cases may become quite blurry. In particular, the problem can easily appear in specifications where use-cases are composed using *include* and *precede* relationships [3].

The intended behavior expressed by use-cases contains implicit temporal dependencies that are likely violated during the iterative development. Because late detection of such errors leads to significantly higher costs of a project [2], writers of the specification greatly benefit from tools that help them keep the textual specification consistent and that warn them about potential violations immediately during writing.

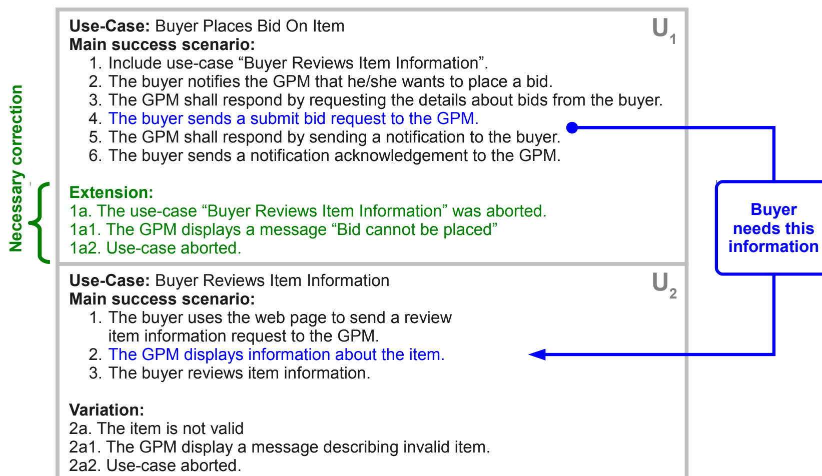

Fig. 1. Example use-cases with aborts (textual form)

Motivation example: Figure 1 shows a pair of the dependent use-cases U_1 and U_2 specified as a sequence of English sentences (U_1 includes U_2). The final text of these use-cases was created in 3 iterations. In the first iteration, an initial version was created with just a simple success scenario. In the second iteration, the use-case U_2 was refined by introducing an optional branch (variation) aborting U_2. However, such specification is not consistent: U_1 does not consider a possible abort in U_2. In more detail, there is a possible trace leading to usage of an unavailable item when U_2 has been aborted.

Problem statement. Such an inconsistency may be detected only when both use-cases are put into context of one another using the *include* relationship. This makes

such inconsistencies difficult to notice, especially when specification is large with many use-cases and *include* relationships.

So it would be desirable to propose a method that, in an automatized way, detects such an inconsistency and issues a warning. In the example, as a reaction to such a warning, U_1 could be manually extended by adding an abort-handling branch to affect the set of traces that involve branching transitions. Verification would now succeed because the traces involving the abort step in U_2 would be limited to the abort-handling branch.

Goal. Thus, the goal of this paper is to present a method that allows an early detection of violation of temporal dependencies of use-case steps. The proposed method (Use-Case Temporal Verification – UCTV) allows automated derivation of a formal behavior model (LTS) from use-cases in plain English. Moreover, by adding annotations to use-case steps, it is possible to verify temporal properties in an automatized way in order to identify inconsistencies within the original specification. The detected errors are presented to the user as erroneous traces. For automated transformation of the use-cases into the formal model and verification of temporal dependencies, we designed a software tool (REPROTOOL), which stems from the PROCASOR tool [12,4,15] designed earlier in our group.

Other approaches exist that aim at extracting behavior models from text, for example authors of [18] describe how to generate UML Activity Diagrams from use-cases. The method uses restriction rules [19] imposed on the use-case step sentences. In [7], a method for deriving message sequence charts from textual scenarios is described.

Several languages and formalisms for behavior modeling of software systems have been proposed. They range from very generic ones (e.g., process algebras [6,13]), to those specific to components (e.g., Darwin [11], Interface automata [1], or Threaded Behavior Protocols [8]).

To achieve the goal, the paper is structured as follows: In Section 2 we overview the main concepts in UML use-cases as the terminology base used further in the paper. Section 3 describes how users interact with an application that implements our method. In Section 4 we explain the algorithm in detail, while Section 6 concludes the paper.

2 Use-cases in natural language and UML

The prevalent practice of capturing use-cases is to use textual notation and natural language. Futher, UML Use-Case Diagrams provide means for establishing relations among use-cases.

Although there are different styles of writing use-cases, for our purposes we consider the format depicted in Figure 1 and 4. This format is taken from the book [3] as it is widely accepted.

With regard to the structure of a use-case, the *main success scenario* of a use-case consists of several steps that contribute to achieving the use-case goal. Alternative scenarios can be expressed using *variations* and *extensions*. The difference between extensions and variations is that a variation *replaces* the step to which it is attached, while an extension provides *optional branching* from its parent step. For illustration, consider the use-case U_1 in Figure 4. There is a variation $2a$ attached to the step 2 which means

that 2 and $2a$ are mutually exclusive branches. On the other hand, the use-case U_2 contains an extension $1b$ which means that the step 1 is always executed before the optional $1b$ branch.

2.1 Actions in use-case steps

It has been advised by practitioners, e.g. in [3], to use simple sentences when writing use-cases. A sentence should encode a single action, which is either (a) interaction between an actor and SuD, (b) internal action within SuD, or (c) special action (see below). As to the structure of a sentence, in English it should conform to the SVDPI pattern (Subject, Verb, Direct-Object, Preposition, and Indirect-Object); this is very important for an automated processing. The following special actions are introduced in the UCTV method:

Goto action : The trace advances by another step (indicated by this action) within the same use-case. This action is typically used to express looping. Example: *Goto step 1.* (See Figure 4 use-case U_2 step $2a2$).

Include action : Similar to calling a procedure, the trace advances in the included use-case, when it is finished, the include action is concluded. Example: *Include use-case "Generate city"* (e.g. use-case U_2, step 1 in Figure 4).

Abort action : The use-case execution is aborted. However, if the aborted use-case U_3 was included into another use-case U_2, the trace immediately advances in U_2. Example: *Use-case aborts* (e.g. use-case U_3, step $2a1$ in Figure 4).

2.2 Relations in the UML Use-Case Diagrams

UML provides means for expressing dependencies among use-cases using stereotyped relations in the UML Use-Case Diagrams. The UCTV method takes into account the **«includes»** (via the include special action) and **«precedes»** UML relationships:

U_1 **«includes»** U_2 : The *include* relationship allows inserting the behavior from one use-case into another. It minimizes duplication and improves comprehension of the whole specification when used carefully. The use of *include* means that at the given point in use-case A, the trace advances over the steps in B and when B is finished, it returns back to A [9].

U_1 **«precedes»** U_2 : Rosenberg and Stephens in the book [16] define the *precedence* relationship as: The use-case U_1 must take place in its entirety before another use-case U_2 even begins, i.e. there is temporal precedence in which U_1 must occur before U_2. For example a $Login$ use-case must be completed before $Checkout$ is begun.

We use the $Prec$ precedence relation formed by the pairs of use-cases, in which first use-case precedes the second one.

3 User's perspective

Before we present UCTV in a formal way, let us describe use-case design from the user's perspective. Figure 2 contains a screenshot from our application REPROTOOL that the user employs when writing a use-case specification[2].

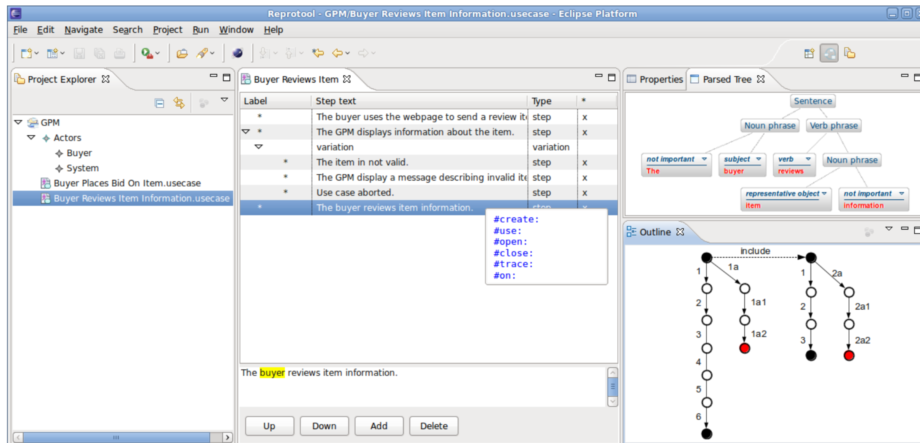

Fig. 2. Screenshot from the REPROTOOL application.

In the first phase, the user creates several use-cases with steps written as sentences in plain English. Each sentence (step) is automatically parsed and transformed into a linguistic parse tree. Depending on the sentence structure, the type of the action is derived automatically or set manually by the user. This way, REPROTOOL derives LTS from the use-cases and renders a graphical representation of the LTS as depicted in Figure 2.

In the next phase, the user can assign annotations to individual steps to define precedence relations determining temporal dependencies among use-cases and their steps. These will be verified in the next phase.

When looking at the motivation example, the temporal dependency between U_1 and U_2 can be captured using a pair of annotations – *use:item* and *create:item* (for illustration see Figure 3 providing annotated use-cases and capturing their creation in iterations). The semantics of them is that in each trace containing a step with the *use:item* annotation, any other step with a *create:item* annotation has to precede the former (pairwise).

At some point, during the iterative process of writing use-cases, the user initiates the verification procedure performed within the REPROTOOL application.

[2] REPROTOOL is based on Eclipse and uses Eclipse Modeling Framework (EMF) as a tool for data representation. The application is still under development and not yet completely finished, see `http://code.google.com/a/eclipselabs.org/p/reprotool/`.

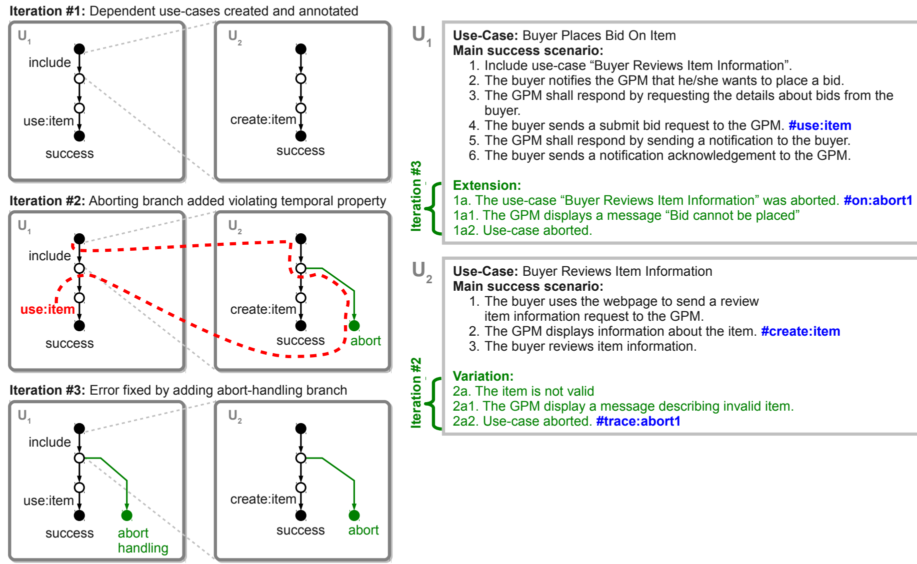

Fig. 3. Verification of dependent use-cases with aborts using temporal annotations

If a verification error is detected, the model-checker shows a trace that violates the temporal properties determined by annotations. After the verification is finished, the user can adjust the textual specification to fix the reported error by:

- Adding precedence relationships among use-cases, which fix the missing *create* annotation that the preceded use-case might have provided.
- Adding an abort-handling branch as seen in the motivation example (Fig. 3).
- Adjusting annotations of steps or rewriting/reorganizing them.

In the motivation example, after introducing the abort branch (variation $2a$ of U_2), the model-checker detected an error trace (Figure 3, *Iteration #2*). The user fixed the error by adding an extension $1a$ into U_1 (Figure 3, *Iteration #3* and Figure 1, "Necessary correction").

In addition to the *create-use* annotation pair, we have also described the *open-close* annotation pair in the text below. These annotations cover the majority of dependencies among use-cases that we encountered in our survey [17]. However, since the UCTV method internally uses LTL formulae to capture desired temporal properties, our approach is also applicable for other annotations, the semantics of which can be described by LTL or other temporal logic formlae.

4 Verification of use-cases

In this section, we describe all the annotations and the REPROTOOL verification algorithm in detail.

4.1 Annotations in use-case steps

There are two types of annotations: (a) annotations expressing temporal dependencies (technically translated to LTL), and (b) annotations constraining the set of traces to be inspected by the model-checker.

(a) Annotations expressing temporal dependencies ("temporal annotations") :
The create-use annotation pair: In all traces it must hold that for any step annotated by *use:x* there must previously appear a step annotated by *create:x* (as above, x is a user-chosen identifier). That is, if x is used, it must be created before. Next, it must hold that for each step annotated by *create:x*, there must be a trace reaching this step and then eventually reaching another step annotated by *use:x*. In other words, if x is created then it must be somewhere used in the future [3]. An example is shown in Figure 4.
The open-close annotation pair: For any trace that with a step annotated by *open:x*, there must eventually appear a step annotated by *close:x*. Obviously, another *open:x* step is not allowed in between. In a similar vein, *close:x* cannot appear without a preceding *open:x*.

(b) Annotations constraining traces :
The trace-on annotation pair: These annotations serve to control application of use-case variations.

Technically, the *trace:x* annotation marks with a flag x all the traces going through the step where this annotation appears. This flag may be later tested and used as a guard in branching via the annotation *on:x*. That is, a trace that goes through a step marked with *trace:x* annotation and reaches this branching state must continue using the step marked with an *on:x* annotation and a trace that does not go through a step marked with a *trace:x* annotation and reaches this branching state must continue using any other step going from this state.

Typically, this annotation pair is used when detecting unhandled aborts in use-cases. Figures 3 and 4 show examples.

4.2 The verification strategy

Verification of textual use-cases is done in two phases. First the precedences and includes are statically checked for presence of cyclic dependencies. Second, a dedicated type of LTS (use-case automaton) is built from the textual use-cases and further model-checking is employed to verify temporal dependencies expressed by annotations.

Before the actual verification starts, the textual use-cases are parsed using the method described in [4] into an internal form where the sequence of steps, variations and extensions of steps, actions in use-case steps, and annotations of use-case steps are specifically represented. After the internal form has been created, the verification proceeds as described below.

[3] Strictly speaking, creating something without its usage is not an error. Nevertheless, since it is not a good practice, we consider such a trace to be erroneous.

Static check of precedences and includes In this phase, precedence and include use-case relationships are checked statically. The cyclic dependencies among use-cases represented in the internal form are detected by creating an oriented precedence and include graphs.

Model-checking of temporal dependencies In this phase, the internal form is used to build LTS-like structure (based on use-case automata). The annotations expressing temporal dependencies are converted into an LTL formula. Finally, a modified LTL model-checking algorithm (Use Case Model Checking – UCMC)[4] is applied to verify the LTL properties. This phase comprises three steps:

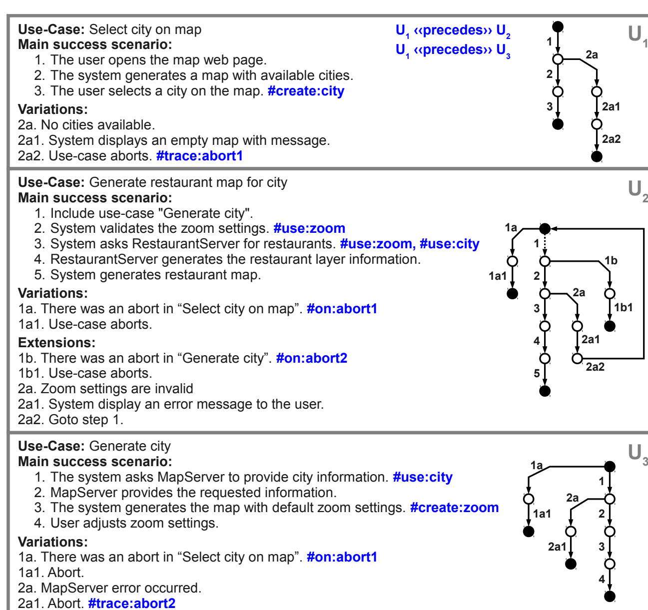

Fig. 4. Fragment of a use-case specification of a web portal providing information about restaurants. For clarity reasons, the annotations (prefixed by the "#" character) are visible only in the textual specification and hidden in the corresponding label transition system (use-case automaton with includes).

- **A use-case automaton with includes (UCAI)** is built for each use-case. Basically, UCAI is an LTS with transitions corresponding to steps of a use-case. Specifically, it contains *include transitions*, which correspond to include steps in the use-case. Figure 4 shows an example of three textual use-cases and the corresponding UCAIs.

[4] The model-checking algorithm cannot be used in its standard form since we consider also finite traces and LTS with guards – discussed in detail in Section 5.5

– By creating **resolved use-case automata (RUCAs)**, includes in UCAIs are inlined. RUCA is obtained from UCAI by replacing each of the *include* transitions by inlining the reference automaton. See Figure 5a for an example of resolution of the automaton U_2 from Figure 4.
  Moreover the annotations constraining the traces are converted into guards (controlled by dedicated variables) on the automata transitions (Figure 5c shows an automaton with guards).
– The annotations expressing **temporal dependencies are converted to LTL formulae**[5]. Figure 5b shows the automaton with annotations on transitions.
– A **set of automata which captures overall behavior of the system (OB)** is created. Because the precedes relations define only a partial ordering of use-case applications, the overall behavior *OB* is determined by a set of all possible sequences of the use-case applications.
  Technically, each such a sequence is represented by a RUCA created by a concatenations of the RUCAs representing the individual members of the sequence. Figure 5d shows an example of concatenation of the use-case automata U_1 and U_2 from Figure 4.
– In the final step, the UCMC algorithm is used to verify each RUCA in OB against the extracted LTL formulae.

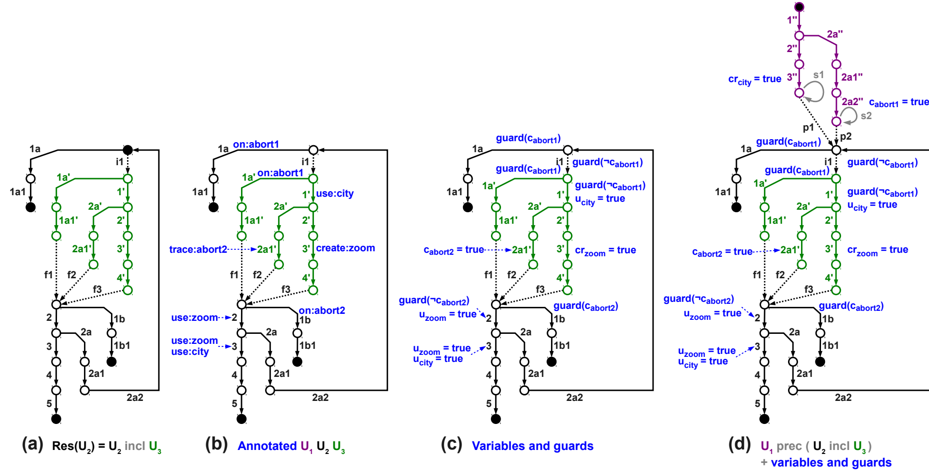

Fig. 5. Visual representation of the construction of the verifying LTS: (a) Included LTS is inlined to the base LTS, (b)+(c) annotations are initialized either as LTL variables or control variables with guards, (d) all final states of the preceded LTS are connected to the initial state of the base LTS. Note: These LTS automata correspond to use-cases from the Figure 4.

[5] In this paper, we only consider LTL formulae corresponding to the *create-use* and *open-close* annotation pairs.

5 Theoretical background

In this section, we provide a formal definition of the key abstraction used by the UCTV method, specifically this includes UCAI, RUCA and a proof of the correctness of the UCMC algorithm.

5.1 Use-case automaton with includes

We define *use-case automaton with includes (UCAI)* and the way it corresponds to a textual use-case. The correspondence is straightforward – steps of a textual use-case correspond to transitions of a *use-case automaton with includes*.

Definition 1 (Use-case automaton with includes–UCAI).
A use-case automaton with includes (*UCAI*) $P = \langle V_P, V_P^{init}, V_P^{abort}, V_P^{succ}, A_P, \tau_P \rangle$ *consists of the following elements:*

- V_P *is a set of states.*
- $V_P^{init} \subseteq V_P$ *is a set of initial states. We require that* V_P^{init} *contains at most one state. If* $V_P^{init} = \emptyset$*, then* P *is called empty.*
- $V_P^{abort} \subseteq V_P$ *is a set of abort states.*
- $V_P^{succ} \subseteq V_P$ *is a set of succeeded states. We require that* V_P^{succ} *contains at most one state.*
- $A_P = A_P^I \cup A_P^{include} \cup \{\epsilon\}$ *is a set of all actions.* $A_P^I, A_P^{include}$ *are mutually disjoint sets of internal and include actions,* ϵ *is the empty action.*
- $\tau_P \subseteq V_P \times A_P \times V_P$ *is a set of transitions.*

Definition 2 (Annotation function). *Let* A_P *be a set of actions of UCAI* P *and* N *a set of annotations.* Annotation function $\mathrm{Af} : \tau_P \mapsto 2^N$ *maps a transition of* P *to a set of annotations.*

Note that that two instances of an annotation with an identical name – i.e. two $on : x$ annotations annotating different steps of use-case – are not considered as equal. Hence, there is no annotation that annotates two different steps.

Definition 3 (Correspondence of a use-case to UCAI). *Let* U *be a use-case, let* $P = \langle V_P, V_P^{init}, V_P^{abort}, V_P^{succ}, A_P, \tau_P \rangle$ *be UCAI. We say that* P corresponds to U *if for each step* st_i *of* U *there is the corresponding transition* $t_i = (s_i, a_i, s_i') \in \tau_P, s_i, s_i' \in V_P, a_i \in A_P$ *of* P *such that:*
If st_i *is:*

- *an include step then* $a_i \in A_P^{include}$*, if* st_i *is an abort or a goto step* $a_i = \epsilon$*, otherwise* $a_i \in A_P^I$.

  V_P, V_P^{init}, V_P^{abort}, V_P^{succ}, *and* τ_P *are defined as:*

- *is a transition such that there exists another transition* $st_j \neq st_i$ *with the same target state, then either* st_j *or* st_i *is a goto step, the last step of a variation, or the last step of an extension,*

- *not the first step of the main success scenario, the first step of a variation, or the first step of an extension, let* $st_{i-1} \in U$ *be a step preceding the step* st_i *and* $(s_{i-1}, a_{i-1}, s'_{i-1}) \in \tau_P$ *a corresponding transition of* P*. It holds that* $s'_{i-1} = s_i$*,*
- *the first step of the main success scenario of* U *then* $s_i \in V_P^{init}$*,*
- *the last step of the main success scenario of* U *then* $s'_i \in V_P^{succ}$*,*
- *the first step of a variation of the step* $st_j \in U$ *and let* $(s_j, a_j, s'_j) \in \tau_P$ *be the transition of* P *corresponding to the step* st_j*, then it holds that* $s_i = s_j$*,*
- *the first step of an extension of the step* $st_j \in U$ *and let* $(s_j, a_j, s'_j) \in \tau_P$ *be the transition of* P *corresponding to the step* st_j *and* $st_{j+1} \in U$ *be the step following the step* st_j *and* $(s_{j+1}, a_{j+1}, s'_{j+1}) \in \tau_P$ *corresponding transition, it holds that* $s_i = s'_j$*,*
- *the last step of a variation or an extension and it is an abort step, then* $a_i = \epsilon$ *and* $s'_i \in V^{abort}$*,*
- *the last step of a variation or an extension and it is not an abort or goto step, then let* st_j *be the step that* st_i *extends or variates and* $(s_j, a_j, s'_j) \in \tau_P$ *be the corresponding transition, it holds that* $s'_i = s'_j$*,*
- *a goto step and* $st_j \in U$ *is the target step and let* (s_j, a_j, s'_j) *be the transition of* P *corresponding to the step* st_j*, it holds that* $s'_i = s_j$*.*

The annotation function Af *is defined as: if* st_i *is annotated by a set of annotations* N*, then* $\mathrm{Af}(t_i) = N$*.*

Example 1. Figure 4 shows three textual use-cases U_1, U_2, and U_3 and the corresponding UCAIs.

5.2 Resolution of the include relationship

We define the operation of resolution of includes – a transformation of *UCAI* to *RUCA*. This operation replaces include transitions with transitions of the included automata.

Definition 4 (Resolved use-case automaton–RUCA). Resolved use-case automaton (*RUCA*) *is UCAI that does not contain any include action.*

Definition 5 (Resolution of includes). *Let* P *be UCAI. Let* I *be the set of use-case automata included in automaton* P*. The operation of* resolution of includes (Res) *transforms* $P = \langle V_P, V_P^{init}, V_P^{abort}, V_P^{succ}, A_P, \tau_P \rangle$ *to RUCA* $Q = \langle V_Q, V_Q^{init}, V_Q^{abort}, V_Q^{succ}, A_Q, \tau_Q \rangle$ *in the following way:*

- $V_Q = V_P \bigcup_{U \in I} V_{\mathrm{Res}(U)}$,
- $V_Q^{init} = V_P^{init}$,
- $V_Q^{abort} = V_P^{abort}$,
- $V_Q^{succ} = V_P^{succ}$,
- $A_Q = A_P \setminus A_P^{include} \bigcup_{U \in I} A_{\mathrm{Res}(U)}$,
- $\tau_Q = \tau_P \cup \tau_A \setminus \{\tau_I\}, \tau_I = (s, inc, s') \in \tau_P, inc \in A_P^{inc}, s, s' \in V_P$.

τ_A *is defined as follows. Let* $t_i = (s_i, inc, s'_i) \in \tau_P, s_i, s'_i \in V_P$ *be a transition of the automaton* P *that contains an include action, let* Q_{inc} *be UCAI associated with the include action* inc *and* $R = \mathrm{Res}(Q_{inc})$ *be the corresponding resolved use-case automaton. For every such a transition* t_i*,* τ_A *contains:*

- $(s_i, \epsilon, s_0), s_0 \in V_R^{init}$
- $(s_{final}, \epsilon, s_i')\ s_{final} \in V_R^{succ} \cup V_R^{abort}$

Example 2. Figure 5a shows an example of UCAI U_2 from Figure 4 after the operation of resolution of includes.

5.3 Resolution of annotations

In textual use-cases, additional behavioral restrictions and consistence constraints are captured using annotations. Additional behavioral restrictions are captured using trace-on annotation pair and additional consistency properties are captured by create-use and open-close annotation pairs. We describe how these annotations define valuation of variables in transitions of the automaton, guard functions, and LTL formulae. Guard functions restrict sequences of transitions that the automaton captures and LTL formulae describe consistency requirements on the automaton.

Definition 6 (Valuation of states of RUCA). *Let P be RUCA and X a set of variables.* Valuation of transitions *of P over the set of variables X is a function* $\mathrm{Val}_P : \tau_P \mapsto 2^X$ *that maps each transition of P to a set of variables. We denote each variable $v \in \mathrm{Val}_P(s)$ as* satisfied in a transition $s \in V_P$.

The set of variables X_P is called variables of P *if $\forall x \in X_P$: $\exists v \in V_P$ such that $x \in \mathrm{Val}_P(v)$. By $X_P^s = X_P \setminus \mathrm{Val}(s)$ we denote the set of variables that are* not satisfied in the transition s.

Definition 7 (Guard functions). *Let P be RUCA and X_P a set of variables of P.* Guard functions $\mathrm{Guard}^+ : \tau_p \mapsto (2^{X_P})$ *and* $\mathrm{Guard}^- : \tau_p \mapsto (2^{X_P})$ *map each transition of P to a set of variables.*

The concept of guard functions is important for defining enabled transitions (Definition 13); how a guard function is constructed expresses the Definition 8.

Definition 8 (Correspondence of annotations to valuation of a use-case automaton). *Let P be a RUCA, N the set of all annotations of the transitions of P. We define* Val *(valuation function),* Guard^+ *and* Guard^- *(guard functions) as follows:*

If the annotation $an \in N$ attached to a transition $t = \{s_i, a, s_j\}$ is of the form:

- trace:id*, there is a variable c_{id} such that $c_{id} \in \mathrm{Val}(t)$,*
- on:id*, there is a variable $c_{id} \in \mathrm{Guard}^+(t)$ and for all transitions $t_u = (s_i, a_k, s_n), s_n \neq s_j$, it holds that $c_{id} \in \mathrm{Guard}^-(t_u)$,*
- create:id*, there is a variable cr_{id} such that $cr_{id} \in \mathrm{Val}(t)$,*
- use:id*, there is a variable u_{id} such that $u_{id} \in \mathrm{Val}(t)$,*
- open:id*, there is a variable o_{id} such that $o_{id} \in \mathrm{Val}(t)$,*
- close:id*, there is a variable cl_{id} such that $cl_{id} \in \mathrm{Val}(t)$.*

Consequently, for $t_i \in \tau_p$ is $\mathrm{Guard}^+(t_i) \cap \mathrm{Guard}^-(t_i) = \emptyset$.

Example 3. Figure 5b shows RUCA with annotated transitions and Figure 5c shows this RUCA with valuations of transitions and guards. The transition $1a$ is annotated by a set of annotations $\{on : abort1\}$ and the other transition $i1$ from the input state of the transition $1a$ has no $on : id$ annotation. Hence, values of guard functions on these transitions are defined as follows: $Guard^{+}(1a) = \{c_{abort1}\}$, $Guard^{-}(1a) = \{\}$, $Guard^{+}(i1) = \{\}$, and $Guard^{-}(i1) = \{c_{abort1}\}$.

Definition 9 (Consistency properties). *Let P be RUCA, N the set of all annotations of the states of P. The set of* consistency properties *LTL_P associated with the automaton P is defined as follows:*
If N contains:

- open:id *or* close:id *then LTL_P contains the LTL formulae depicted in Figure 6a,*
- create:id *or* open:id *then LTL_P contains the LTL formulae depicted in Figure 6b.*

(a)	$G(op_{id} \rightarrow F(cl_{id}))$	After 'open' there should always be 'close'
	$\neg cl_{id}\ U\ op_{id}$	First 'open' then 'close'
	$G(cl_{id} \rightarrow X(\neg cl_{id}\ U\ op_{id}))$	No multi-close without open
	$G(op_{id} \rightarrow X(\neg op_{id}\ U\ cl_{id}))$	No multi-open without close
(b)	$G(cr_{id} \rightarrow X(G(\neg cr_{id})))$	Only one 'create'
	$\neg u_{id}\ U\ cr_{id}$	No 'use' before 'create'
	$\neg G(cr_{id} \rightarrow G(\neg u_{id}))$	After 'create' there must be a branch with 'use'

Fig. 6. LTL formulae generated from temporal annotations

Example 4. For RUCA P in Figure 5c we define the following LTL formulae: $LTL_P = \{G(cr_{city} \rightarrow X(G(\neg cr_{city}))), \neg u_{city} U cr_{city}, \neg G(cr_{city} \rightarrow G(\neg u_{city})), G(cr_{zoom} \rightarrow X(G(\neg cr_{zoom}))), \neg u_{zoom}\ U\ cr_{zoom}, \neg G(cr_{zoom} \rightarrow G(\neg u_{zoom}))\}$.

5.4 Resolution of precedence relationship

Now, we define how automata capturing behavior of individual use-cases are serialized according to the precedence relationship.

Definition 10 (Concatenated RUCA). *Let $s = (R_1, R_2, ..., R_k)$ be an sequence of RUCAs.* Concatenated RUCA *Q corresponding to s is defined as follows:*

- $V_Q = \bigcup_{R \in s} V_R$
- $V_Q^{init} = V_{R_1}^{init}$
- $V_Q^{abort} = V_{R_k}^{abort}$

- $V_Q^{succ} = V_{R_k}^{succ}$
- $A_Q = \bigcup_{R \in s} A_R$
- $\tau_Q = \bigcup_{R \in s} \tau_R \cup \tau_A$

τ_A is defined as follows. Let (R_i, R_{i+1}) be a pair of subsequent resolved use-case automata in the sequence s. Let $init_{i+1}$ be the initial state of the automaton R_{i+1}. For every such a pair and every final state $final_i \in V_i^{succ} \cup V_i^{abort}$ of the automaton R_i, there are transitions $(succ_i, \epsilon, init_{i+1})$ and $(final_i, \epsilon, final_i) \in \tau_A$.

Obviously, this definition stems from classical automata concatenation; the key enhancement here is the introduction of the transitions of the form $(final_i, \epsilon, final_i)$, which corresponds to the semantics of Prec. That is, U_i must occur before U_{i+1}, hence all traces that reach U_{i+1} must go through U_i. However, it is not required that U_{i+1} is executed after U_i. There exist infinite traces that go through U_i and loop using the transition $(final_i, \epsilon, final_i)$ thus never reaching U_{i+1}.

Example 5. Figure 5d shows an example of concatenation of RUCA U_1 from Figure 4 and $\mathrm{Res}(U_2)$ from Figure 5a. The initial state of the resulting automaton is the initial state of U_1, abort and succeeded states of the resulting automaton are the same as abort and succeeded states of the automaton $\mathrm{Res}(U_2)$. The two automata are connected using transitions $p1$ and $p2$ going from the final states of the automaton U_1 to the initial state of the automaton $\mathrm{Res}(U_2)$. Then, there are looping transitions $s1$ and $s2$ going from each final state of U_1 back to this state. All these transitions contain the ϵ action.

Definition 11 (Precedence Relation). Precedence relation *defined on a set of RUCA* U $\mathrm{Prec} : U \times U$ *is an antisymmetric and irreflexive relation, whose transitive closure* $\mathrm{Prec}^\star$ *is antisymmetric and irreflexive as well. We say that* U_i^R precedes U_j^R *if* $(U_i^R, U_j^R) \in \mathrm{Prec}$. *We say that* U_k^R must be executed before U_l^R *if* $(U_k^R, U_l^R) \in \mathrm{Prec}^\star$.

Definition 12 (Overall-behavior–OB). *Let U be a set of RUCAs, let* Prec *be a precedence relation, and let S be the set of all permutations of RUCAs from U ordered according to* Prec. *The* overall-behavior OB *set with respect to U and* Prec *is the set of concatenated RUCAs corresponding to members of S.*

Example 6. There are two permutations of use-cases in use-case specification in Figure 4 ordered according to specified precedences. That is, (U_1, U_2, U_3) and (U_1, U_3, U_2). Hence, the set OB for this specification consists of two automata.

It should be noted that our approach does not tackle the problem of parallel execution of use-case steps. Instead, it focuses on verification of temporal properties of all use-case sequences which could be defined by the *precede* relation (these sequences are captured in the OB set).

5.5 Verification algorithm

In this section, we define the verification algorithm and related concepts.

Definition 13 (Enabled transition). *Let P be RUCA. Let $tr = v_0, a_0, v_1, a_1, ..., v_n$ be an alternating sequence of states and actions such that $t_i = (v_i, a_i, v_{i+1}) \in \tau_P$. The transition t_i is* enabled *on tr if all the transitions $t_j, j < i$ are enabled, for all $v^+ \in \text{Guard}^+(t_i)$ there exists $t_k, k \leq i$ such that $v^+ \in \text{Val}(t_k)$, and there is no $t_l, l \leq i$ such that for some $v^- \in \text{Guard}^-(t_i)$ it holds $v^- \in \text{Val}(t_l)$. If the transition is not enabled on tr, we say that it is* disabled *on tr.*

Example 7. Consider the use-case automaton in Figure 5d and the sequence of transitions $(p1, i1, 1', 2a', 2a1', f2, 1b, 1b1)$. For the transition $p1$ both guard functions return the empty set and this transition is trivially enabled on sq_1. Next, $Guard^+(i1) = \{\}$ and $Guard^-(i1) = \{c_{abort1}\}$ and there is no predecessor t_j of a transition $i1$ in the sequence sq_1 such that $c_{abort1} \in \text{Val}(t_j)$. Hence, the transition $i1$ is enabled on sq_1. Values of guard functions on transition $1'$ are the same and therefore this transition is also enabled on sq_1. Transitions $2a'$, $2a1'$, and $f2$ are trivially enabled on sq_1. Transition $1b$ is enabled on sq_1 because $Guard^+(1b) = \{c_{abort2}\}$ and $Guard^-(1b) = \{\}$ and for the transition $2a1'$ it holds $\text{Val}(2a1') = \{c_{abort2}\}$.

Now, consider a sequence of transitions $(p1,\ 1a,\ 1a1)$. Similar to the previous example, the transition $p1$ is trivially enabled on sq_2. $Guard^+(1a) = \{c_{abort1}\}$ and there is no predecessor s_j of the transition $1a$ in the sequence sq_2 for that $c_{abort1} \in \text{Val}(s_j)$. Hence, a transition $1a$ is disabled on sq_2. Both guard functions for a transition $1a1$ return the empty set, however, because a transition $1a$, which precedes the transition $1a1$, is disabled on sq_2; the transition $1a1$ is also disabled on sq_2.

Definition 14 (Execution fragment). *An* execution fragment *of RUCA P is an alternating sequence of states and actions $v_0, a_0, v_1, a_1, ...$ such that all transitions in the sequence $t_i = (v_i, a_i, v_{i+1}) \in \tau_P$ are enabled on P.*

Definition 15 (Execution trace). *An* execution trace *of RUCA P is an execution fragment of use-case automaton P that starts in the initial state of P and is infinite or end in some final state $v^{final} \in V_C^{succ} \cup V_C^{abort}$ of the automaton P.*

Definition 16 (Consistent use-case). *A resolved use-case automaton P is* consistent *if for all execution traces of the automaton P all formulae from LTL_P are satisfied.*

Verification algorithm. The verification algorithm takes a set U of use-cases (already parsed textual use-cases encoded in an internal form). and a precedence relation $Prec$ describing the precedence relationship among use-cases in U as input. First, a static check of precedences and includes is done. If a cyclic dependency is found, the algorithm stops and returns *not consistent*.

Second, model-checking of temporal properties (using UCMC algorithm) is performed: UCAI is built for each use-case in U (Definition 3); the set of RUCAs is created by resolving all UCAIs (Definition 5), then valuation of variables, guard functions (Definition 8), and consistency properties (Definition 9) are generated from annotations of RUCAs, the set OB is built (Definition 12) and then each RUCA in OB is model checked for consistency with generated LTL formulae (Definition 16). If all such automata are consistent, the algorithm returns *consistent*. If there is an inconsistent RUCA in OB, there is an execution trace for which the LTL formula corresponding to a consistency property of the RUCA does not hold. In this case, the algorithm returns *not*

consistent and provides further details comprising (1) the steps of use-cases from U that correspond to this execution trace, and (2) the ordering of use-cases corresponding to the inconsistent RUCA.

RUCA defines formal behavior of use-cases. We are able to model RUCA using the SMV system and then use the SMV system to check all the generated LTL formulae. In the future work, we consider to let a user to specify an arbitrary temporal dependencies by defining new annotations and mapping of these annotations to valuation of variables and LTL formulae.

Theorem 1 (Correctness of the verification algorithm). *Let U be the set of textual use-cases, G_{prec} be the graph describing a precedence relationship, and G_{incl} be the graph describing an include relationship. Assume that G_{prec} and G_{incl} do not contain cycles. Then, the algorithm returns* consistent *iff all the sequences of the use-cases corresponding to the specification consisting of U and complying with G_{prec} does not contain any incorrectly used* create, use, open *or* close *annotation.*

Proof. The proof is based on the fact that the standard algorithm for checking LTL formula in a Kripke structure returns *consistent* iff the Kripke structure satisfies the given LTL formula.

Because RUCA (an element of the OB set) corresponds to a Kripke structure, it is sufficient to show that the semantics of annotated textual specification corresponds to semantics of the generated OB set and LTL formulae.

This can be done in three steps proving that:

1. traces[6] of transitions captured by the elements of OB (RUCAs) exactly correspond to the sequences of steps captured by U with G_{prec} when the annotations are not considered,
2. execution traces of RUCAs in OB correspond to the sequences of steps captured by U with G_{prec} when the *trace-on* annotations are considered.
3. based on (1) and (2), from the Definition 9, it follows that LTL formulae generated from the *create-use* and *open-close* annotations correspond to semantics of these annotations. Specifically from this fact and the step (2), it follows that the sequences of steps captured by U and G_{prec} with correctly used annotations exactly correspond to the execution traces where all the generated LTL formulae are satisfied. Since there are no cyclic include dependencies and the number of variables is finite, the number of traces to explore is also finite and the algorithm eventually terminates. And thus the algorithm is correct.

Let us prove now the step (1). From the Definition 3 and the Definition 5 it follows that there is a sequence of steps that a use-case $u \in U$ describes iff there is a trace in RUCA corresponding to u. From the Definition 10 it follows that the semantics of concatenation of RUCA corresponds to the semantics of G_{prec}. From the Definition 12 it follows that for each possible order of executions of the use-cases in U determined

[6] The term *trace* in this context is defined in the same way as the *execution trace* (see Definition 15) with the modification that all the possible transitions are considered (not just the enabled ones).

by G_{prec} there is a RUCA in OB such that it consists of the RUCAs concatenated in compliance with G_{prec}.

Finally, let us prove the step (2). From the Definition 3 it follows that the annotations of steps of a use-case $u \in U$ correspond to the annotations of traces of the RUCA corresponding to u. The *trace-on* annotations restrict the sequences of steps captured by the specification. From this fact and (1), it follows that for each sequence of steps captured by U and G_{prec} when the *trace-on* annotations are considered, there is a trace of a RUCA from OB. The trace is the execution trace if for each transition annotated with *on:id* there is a transition annotated with *trace:id* before this transition. That is, there is no execution fragment which would not correspond to a sequence of steps captured by U and G_{prec}.

6 Summary and Future Work

We have developed means for verifying consistency of textual use-cases useful especially when use-cases are written iteratively by multiple authors. By introducing annotations to use-case steps, we can capture temporal dependencies among use-cases which is a foundation for further verification of temporal properties (based on LTL). As a key contribution, we have defined a formal behavior model (based on LTS) and defined its correspondence to textual use-case specification. A formal behavior model satisfying LTL formulae inferred from user annotations corresponds to a consistent use-case specification. Even though we have considered just two annotation pairs, the *create-use* and *open-close* pairs, our approach is applicable for other annotations as well, the semantics of which can be described by LTL. This is because we internally use LTL formulae to capture desired temporal properties. It should be noted that most of the examples in the text were taken from case studies of real-life use-cases [5].

Currently, we continue the development of REPROTOOL which integrates the verification method with analysis of natural language. As a future work we plan to tackle the following challenges:

- We plan to extend the palette of annotations in future and potentially to let users define their own annotations using arbitrary LTL-formulae.
- We could also implement asynchronous events in use-case specification. As pointed out by Larman [9] these events can be attached to multiple steps, e.g. “at any time” or “within a range of steps”.
- Our method would work even if we did not use any tools for processing natural language. Users could manually mark sentences as *goto-*, *abort-* or *include-actions*. However, due to the restrictions of the natural language in use-case specifications [3,9,19], we can benefit from NLP tools and thus automate this process. It should be also possible to infer the use-case step annotations from the text automatically. We intend to improve the currently employed NLP tools in REPROTOOL.

Acknowledgements. This work was partially supported by the Grant Agency of the Czech Republic project P103/11/1489, by the Ministry of Education of the Czech Republic (grant MSM0021620838) and by the grant SVV-2011-263312.

References

1. de Alfaro, L., Henzinger, T.A.: Interface Automata. SIGSOFT Softw. Eng. Notes 26(5), 109–120 (2001)
2. Boehm, B.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs (1981)
3. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, Boston, MA, USA (2000)
4. Drazan, J., Mencl, V.: Improved processing of textual use cases: Deriving behavior specifications. In: Proc. of SOFSEM '07. pp. 856–868. Springer (2007)
5. Firesmith, D.: Global personal marketplace system requirements specification (2003), http://www.it.uu.se/edu/course/homepage/pvt/SRS.pdf
6. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall Int. (UK) Ltd. (1985)
7. Kof, L.: From textual scenarios to message sequence charts: Inclusion of condition generation and actor extraction. In: Proc. RE 2008. pp. 331–332. IEEE CS (2008)
8. Kofron, J., Poch, T., Sery, O.: TBP: Code-Oriented Component Behavior Specification. In: SEW '08: Proceedings of the 2008 32nd Annual IEEE Software Engineering Workshop. pp. 75–83. IEEE CS, Washington, DC, USA (2008)
9. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall PTR, Upper Saddle River, NJ, USA (2004)
10. Luisa, M., Mariangela, F., Pierluigi, I.: Market research for requirements analysis using linguistic tools. Requir. Eng. 9, 40–56 (February 2004)
11. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architectures. In: Fifth European Software Engineering Conference, ESEC '95 , Barcelona (1995), http://pubs.doc.ic.ac.uk/SpecifyDistributedArchitectures/
12. Mencl, V.: Deriving behavior specifications from textual use cases. In: Proc. of WITSE'04 (September 2004)
13. Milner, R.: Communication and Concurrency. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1995)
14. OMG: Unified Modeling Language (2008), http://www.uml.org
15. Plasil, F., Mencl, V.: Getting 'Whole Picture' Behavior In A Use Case Model. Journ. of Integrated Design and Process Sci. 7(4), 63–79 (2003)
16. Pow-Sang, J.A., Nakasone, A., Imbert, R., Moreno, A.M.: An approach to determine software requirement construction sequences based on use cases. In: Proc. of ASEA'08. pp. 17–22. IEEE CS, Washington, DC, USA (2008)
17. Simko, V.: Patterns in specification documents. Tech. Rep. 2011/6, Charles Uni. (2011), http://d3s.mff.cuni.cz/publications/download/tr2011-6.pdf
18. Yue, T., Briand, L., Labiche, Y.: An automated approach to transform use cases into activity diagrams. In: Proc. of ECMFA 2010, LNCS, vol. 6138, pp. 337–353. Springer (2010)
19. Yue, T., Briand, L.C., Labiche, Y.: A use case modeling approach to facilitate the transition towards analysis models: Concepts and empirical evaluation. In: Proc. of MODELS '09. pp. 484–498. Springer, Berlin, Heidelberg (2009)