Formal Verification of Annotated Textual Use-Cases

Viliam Simko\textsuperscript{1,2,*}, David Hauzar\textsuperscript{1}, Petr Hnetynka\textsuperscript{1}, Tomas Bures\textsuperscript{1,3} and Frantisek Plasil\textsuperscript{1,3}

\textsuperscript{1}Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University, Malostranské náměstí 25, 11800 Prague 1, Czech Republic
\textsuperscript{2}Institute for Program Structures and Data Organisation, Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany
\textsuperscript{3}Institute of Computer Science, Academy of Sciences of the Czech Republic, Pod Vodárenskou věží 2, 18207 Prague 8, Czech Republic
*Corresponding author: simko@d3s.mff.cuni.cz

Textual use-cases have been traditionally used in the initial stages of the software development process to describe software functionality from the user’s perspective. Their advantage is that they can be easily understood by stakeholders and domain experts. However, since use-cases typically rely on natural language, they cannot be directly subject to a formal verification. In this article, we present a method (called Formal Verification of Annotated Use-Case Models, FOAM) for formal verification of use-cases. This method features simple user-definable annotations, which are inserted into a use-case to make its semantics more suitable for verification. Subsequently, a model-checking tool is employed to verify temporal invariants associated with the annotations. This way, FOAM allows harnessing the benefits of model checking while still keeping the use-cases understandable for non-experts.

Keywords: specification; use-cases; behavior modeling; verification; temporal logic; formalization

Received 16 October 2013; revised 13 June 2014
Handling editor: Manfred Broy

1. INTRODUCTION

Specification of functional requirements using textual use-cases [1] is a well-established technique in requirements engineering. A use-case captures a particular functionality of the system textually, as a scenario of actions and responses written in a natural language. The use of the natural language makes textual use-cases an ideal approach for consulting the intended behavior of a developed system, i.e. System Under Discussion (SuD), with the users/stakeholders. However, the natural language brings the risk of ambiguity or contradiction in specification documents which can negatively impact later phases of the system development. Problems inherent to natural language, such as inconsistent wording, naming entities or other ambiguities can be partially avoided by using controlled languages, templates and restriction rules.\textsuperscript{1} Even though, the use-case specification can still be logically inconsistent and with the increasing complexity it becomes harder to ensure its validity. Examples include mutually incompatible behavior of specification fragments and violation of constraints assumed by the use-case authors [3]. Thus, a formalization and automated verification of use-cases is desirable and very important. The Formal Verification of Annotated Use-Case Models (FOAM) method described in this article addresses a particular problem related to use-cases—ensuring a correct sequencing of actions in a set of related use-cases.

The problem of correct sequencing of actions is shown by the use-cases given in Fig. 1. The use-cases $u_1$, $u_2$ and $u_3$ operate with location. The order, in which the steps are executed, is important because in step $u_2(2)$, \textit{location} is provided by the server while later it is being referred to in step $u_3(3)$. A correct sequencing of $u_2(2)$ and $u_3(3)$ requires that the former happens prior to the latter because \textit{location} is used in $u_3$ as a filter for obtaining a list of restaurants near the selected location. Furthermore, during evolution of the specification, new branching conditions can be added,
FIGURE 1. Example of use-cases sharing an artifact relevant for the sequencing of actions.

<table>
<thead>
<tr>
<th>UseCase: u₂ Generate city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preceding: u₁ “Select city on map”</td>
</tr>
<tr>
<td>1. The system asks MapServer to provide city information.</td>
</tr>
<tr>
<td>2. MapServer provides the requested information. <em>(providing the location)</em></td>
</tr>
<tr>
<td>3. The system generates the map with default zoom settings.</td>
</tr>
<tr>
<td>4. User adjusts zoom settings.</td>
</tr>
</tbody>
</table>

Variation: 2a. MapServer error occurred.  
2a1. The use-case aborts.

FIGURE 2. A variation causing an inconsistency is added to the specification.

<table>
<thead>
<tr>
<th>UseCase: u₃ Generate restaurant map for city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preceding: u₁ “Select city on map”</td>
</tr>
<tr>
<td>1. Include use-case “Generate city”.</td>
</tr>
<tr>
<td>2. System validates the zoom settings.</td>
</tr>
<tr>
<td>3. System asks RestaurantServer for restaurants near the location. <em>(near the given location)</em></td>
</tr>
<tr>
<td>4. RestaurantServer generates the restaurant layer information.</td>
</tr>
<tr>
<td>5. System generates restaurant map.</td>
</tr>
</tbody>
</table>

Extension: 1a. There was an abort in “Generate city”.  
1a1. System shows an error message to the user.  
1a2. Use-case aborts.

FIGURE 3. An extension resolving the inconsistency is added to the specification.

Changing the normal execution-flow, e.g. by terminating a use-case. In our example (Fig. 2), the branch Variation 2a in u₂, leading to an abort, has been introduced in a later project phase. However, in a case this branch is being executed, location has not been provided until u₃ starts execution, which would result in a problem inside u₃. Thus, in order to avoid the inconsistency, the Extension 1a has been added into u₃ (Fig. 3). Even from such a small example, it is obvious that manual detection of such inconsistencies is almost impossible (especially in projects where the use-cases are prepared by a group of analysts). Therefore, an automated verification mechanism is in fact a necessity.

Originally, use-cases were not intended for modeling of consequential effects across different use-cases; rather, they were invented for documentation purposes. However, the issue of not considering the sequencing of use-cases was soon criticized, first in the paper [4].

In this article, we propose a method (called FOAM) allowing verification of correct sequencing of actions in use-cases. Our method works with use-cases in their natural language form, requiring only a few basic annotations to be inserted in the use-cases to support the verification.

Apart from our earlier approaches [5, 6], this article covers the following topics: (i) detailed formal description of the...
method using inference rules, (ii) formal proof that the method has sufficient expressive power, (iii) implementation of the FOAM tool, (iv) evaluation of the learning curve and usability and (v) evaluation of the scalability with respect to real-life use-cases—our method scales linearly with the number of use-cases.

The structure of the article is as follows. Section 2 defines the used format of use-cases. Section 3 overviews our method while Section 4 describes its details. In Section 5, we provide proofs about expressiveness and correctness of the proposed method. In Section 6, we discuss scalability. Sections 7–9 are devoted to an evaluation of the learning curve. Section 10 compares our method with related work and, finally, Section 11 concludes the article.

2. TEXTUAL USE-CASES
A use-case describes a particular functionality of a system in natural language. This makes use-cases a very advantageous asset for communicating software specification with customers as well as with developers; however, the use of natural language is also the main obstacle in automated processing and verification of use-cases. This is not just because of the intricacies of interpreting the text in natural language, but also because, to date, there is no standardized form of use-case description. To overcome the latter, we adhere to the widely accepted format proposed in the book by Cockburn [1]. The former is, however, a much more serious problem, which we tackle in our method by introducing annotations as explained further in the article. To facilitate reading, we briefly summarize this use-case format below.

Use-cases: key concepts
Typically, the SuD is specified as a set of use-cases (further denoted as UCM, i.e. Use-Case Model). A single use-case always specifies the main scenario and a (potentially empty) set of branching scenarios. Each scenario comprises a sequence of steps. A use-case step, written as a simple sentence in a natural language (English in our case), expresses an interaction between SuD and actors (typically users and stakeholders). A use-case step is identified by its sequence number. The main scenario (also called success scenario) defines the sequence of interactions for achieving the goal of the use-case (e.g. steps 1–3 in Fig. 3). Branching scenarios refer to the sequence number of their parent step. For example, the variation 2a is an alternative to the step 2 in the use-case u2. Branching scenarios can be guarded to determine when a scenario can be non-deterministically chosen for execution.

Branching: variations vs. extensions
In FOAM, we distinguish two types of branching scenarios—extensions and variations—primarily due to convenience for the end-user. The difference between them is that an extension enhances specification of its parent step, while a variation is an alternative to the step’s specification. In other words, if there is an extension defined, then the parent step followed by its extension are executed. Whereas if a variation is defined, then exclusively the variation or its parent step is executed.

Relations among use-cases
Use-cases within a UCM, as understood in FOAM, can participate in precedence and include relations.

The include relation allows reuse of functionality. Jacobson in his paper [7] mentions that his original intention was only to allow fragments of text to be included. The reason was that he foresaw people misusing use-cases by applying them for functional decomposition. He feared that use-cases would be viewed as ‘just another way to do functions’. Nevertheless, in the paper [7] he acknowledges that extension and inclusion of a whole use-case is desirable. As an aside, we follow this acknowledgement in FOAM.

The intuition behind the precedence relation can be understood from the perspective of a single use-case. A use-case may require that other (preceding) use-cases should already have finished their execution, without enforcing any particular order (e.g. in Fig. 1, before the use-cases u2 or u3 can be executed, the use-case u1 has to be executed first). In general, a preceding use-case may have been executed once or repeated any number of times before reaching its successor. In practice, however, the most intuitive way is to consider the exactly-once-semantics as the default (as in the papers [8–10]) and to model other situations explicitly. In FOAM, we follow this practice and assume that each of the preceding use-cases is executed exactly once. When needed, a more-than-once-semantics has to be modeled explicitly case by case (e.g. by multiple usage of include relation and branching).

Finally, use-cases influence each other if there are temporal dependences among their use-case steps (as shown in Fig. 1).

3. OVERVIEW OF THE FOAM METHOD
As discussed above, ensuring the correctness of an evolving specifications is hard. It would be a mistake to understand requirements documents as final and unchangeable as emphasized in the book [11]. Automation is required especially for large and complex specifications where manual reviewing becomes tedious. Our goal in FOAM is to provide means for capturing and verifying temporal dependencies among steps in a use-case specification. This naturally leads to a solution involving temporal logics such as Linear Temporal Logic (LTL) or Computational Tree Logic (CTL). However, a temporal logic could cause problems for non-technical users. Therefore, we designed temporal annotations that hide the complexity of the temporal logic under an intuitive interface.
As a first step in the automation process, it is necessary to extract the control flow from the use-cases. Next, we need to formally define conditions to be verified. All this information is available in the use-cases but it is ‘hidden’ in the natural language of the text. To overcome this, we propose to enhance the use-cases with annotations, which are short tags appended to a particular use-case step sentence (written as #(a:s) in Fig. 4). These annotations can be divided into two groups: (1) flow annotations expressing control flow of use-cases and (2) temporal annotation expressing conditions to be verified. In an ideal case, these annotations would be automatically added to the use-case. However, such an automation is a complex and challenging task that would require an in-depth linguistic analysis the accuracy of which would be highly dependent on the problem domain as well as the writing style.

The verification itself then takes the annotated use-cases and automatically transforms them into a Labeled Transition System (LTS). This LTS (an automaton) encodes the execution of all scenarios expressed by the use-cases. The process continues further by transforming the LTS into the input for the NuSMV model-checker [12], which verifies the use-cases. Transformations are transparent to the user; the potential errors reported by NuSMV are presented in the natural language by translating the counter-example to the steps of the flawed use-case(s). We have implemented the transformation pipeline in the FOAM tool which is sketched in Fig. 6. In the rest of this section, we describe the annotations in detail.

### 3.1. Flow annotations

Execution of a use-case starts with the first step of its main scenario and then continues till the end, possibly, visiting optional branches. However, the control flow of the execution can be further altered by: (1) aborts which prematurely end the scenario—typically as a reaction to an error; (2) includes which incorporate (inline) another use-case in the place of a particular step, (3) jumps which move execution to a specified use-case step and (4) conditions of extensions and variations.
All these constructs are written in the natural language. FOAM considers them the core concepts influencing the control flow and captures them formally using annotations of the following form:

- #(abort): This annotation expresses an abort of the scenario. An abort causes the execution of a use-case to be terminated at the step where the annotation appears. However, if the use-case has been included into another use-case, only the included use-case is terminated and the execution returns to the parent use-case at the step immediately following the including step.
- #(include:u): This annotation specifies an inclusion (inlining) of another use-case u.

The following annotations #(mark) and #(guard) assume the existence of globally accessible boolean variables b_1, ..., b_n initialized to false.

- #(mark: b_i): This annotation sets b_i to true.
- #(guard: f (b_1, ..., b_k)): The f parameter of this annotation is a propositional logic formula over the boolean variables b_1, ..., b_k. The annotation serves as a guard for extensions and variations. That is, a variation/extension can be followed only if the formula f is true.

Further in the text (Section 1) we also discuss the restrictions of using flow annotations in use-case steps.

### 3.1.1. Expansion of expressions in guards

To simplify the usage of guard annotations, we support special expressions within guards that automatically inject new #(mark) annotations to some steps. This reduces the number of #(mark) annotations created manually.

These automated expansions are as follows:

- **Abort-handling**: Each #(guard:abort) annotation automatically causes the injection of #(mark: abort) into all steps that contain the #(abort) annotation (used, e.g. in Fig. 4).

**References to steps**: This is useful if we want to refer to a particular use-case step by its sequence number. An example is in Fig. A9. We wanted to follow the branch 4a if it has not been visited before, thus avoiding an infinite loop. The annotation #(guard: !visited(4a)) automatically injects #(mark: v_1) into the step MOD2_U1C1.4a and internally translates the guard into #(guard: !v_1).

### References to other annotations:

Sometimes it is useful to execute a branch if a particular annotation has already been visited on the trace. For example, in Fig. A7, the annotation #(guard: !create:) #(registeredMoney) injects #(mark: v_2) into all steps that contain the annotation #(create: registeredMoney) and translates the guard internally into #(guard: !v_2).

**Wild-cards**: We also support a simple wild-card-based expressions. Namely, the wild-card ‘*’ expands to a sequence of 0...∞ characters while ‘?’ represents a single character. An example is the annotation #(guard: create:*Money) that would inject #(mark: v_3) into all steps annotated with a matching annotation, e.g. #(create: withdrawMoney) or #(create: registerMoney).

### 3.2. Temporal annotations

Temporal annotations allow expressing temporal invariants among use-case steps in the whole UCM without requiring an in-depth knowledge of the underlying temporal logic (CTL or LTL).

In Fig. 4, these annotations are: #(create: city), #(use: city), #(create: zoom), #(use: zoom). FOAM allows these annotation to be user-defined. In particular, it distinguishes two types of users:

- (a) **experts in temporal logic** who prepare templates of annotations in the FOAM’s Temporal Annotation Definition Language (TADL), i.e. our language for template definition (for an example see Fig. 5).
- (b) **domain engineers** who refer to the names of these templates when associating use-case steps with annotations (Fig. 4). For this activity, a detailed knowledge of the temporal logic is not necessary.

Specifically, when an annotation #(x:y) appears in a specification, the TADL definition for x is used to convert it into a set of temporal formulae (where x is substituted by x_y). The transformation is described in detail in Section 4.5.

TADL defines a group of related temporal annotations along with their semantics expressed as a set of temporal logic formulae in CTL, LTL and PLTL. These formulae can be connected with usual propositional logic operators such as ¬ϕ (Negation), α | β (OR), α & β (AND), α → β (Implication). The temporal constraint expressed by a formula is also written down in a human-readable form for error reporting which is useful for showing a counter-example to the user.

The formulae need to be transformed into the representation supported by a particular model-checking back-end, in our case to NuSMV (see Section 4.7).

Here is the complete list of all the temporal operators supported by the FOAM verification tool:

---

2PLTL is a convenient extension of LTL that defines operators related to the past rather than the future.
Supported CTL operators:

$$AX(\varphi):$$ $\varphi$ holds on all paths in the next state.

$$AG(\varphi):$$ $\varphi$ holds on all paths in all states.

$$AF(\varphi):$$ On all paths there is some state in the future where $\varphi$ holds.

$$A[\alpha U \beta]:$$ On all paths, at some point $\beta$ holds while in the meantime $\alpha$ holds.

$$EX(\varphi):$$ There is a path on which $\varphi$ holds in the next state.

$$EG(\varphi):$$ There is a path on which $\varphi$ holds globally.

$$EF(\varphi):$$ There is a path with a state in the future where $\varphi$ holds.

$$E[\alpha U \beta]:$$ There is a path where at some point $\beta$ holds while in the meantime $\alpha$ holds.

Supported PLTL operators (related to the past):

$$Y(\varphi):$$ ‘Yesterday’, i.e. $\varphi$ holds in the previous state.

$$H(\varphi):$$ ‘Historically’, i.e. $\varphi$ holds on all states in the past.

$$O(\varphi):$$ ‘Once’, i.e. $\varphi$ holds in some state in the past.

$$\alpha S \beta:$$ ‘Since’ is a temporal dual of $U$ (until), so that $\beta$ holds somewhere in the past and $\alpha$ is true from then up to now.

$$\alpha T \beta:$$ ‘Triggered’ is the temporal dual for $R$, which means that: $\alpha T \beta \equiv \neg[\neg \alpha S \neg \beta]$. In other words, $\beta$ holds up until the present, starting from the time-step where $\alpha$ was true. If $\alpha$ never was true, then $\beta$ must have been true always in the past.

Let us now examine the example in Fig. 5 in more details. There are four temporal annotation groups defined here. The create, use annotations allow expressing constraints on ordering of the use-case steps. For instance, in Fig. 4, the step 1 of the use-case $u_2$ annotated with $\#(use:city)$ should be executed only if there was a previously executed step with the $\#(create:city)$ annotation. Additionally, there should not be an execution with several $\#(create:city)$ annotations.
The transformation of these annotations would result in the following set of formulae:

- LTL: G(use\(_{city}\) → O(create\(_{city}\))) 'There must be create before use'
- CTL: AG(create\(_{city}\) → EF(use\(_{city}\))) 'At least one branch with use required after create'
- CTL: AG(create\(_{city}\) → AX(AG(¬create\(_{city}\)))) 'Only one create'

Figure 5 illustrates how more complex annotations can be defined in TADL. The open, close template is an example of a strict ordering of two phases: phase\(_1\), #(open), phase\(_2\), #(close), phase\(_1\). The term phase\(_1\) denotes a sequence of steps outside of the open–close pair, while phase\(_2\) denotes the sequence inside the pair. Similarly, the init, process, release template is an example of a strict ordering of three phases: phase\(_1\), #(init), phase\(_2\), #(process), phase\(_3\), #(release), phase\(_1\). This can be generalized to any number of phases as demonstrated by the template \(a_1, a_2, \ldots, a_N\) showing a strict ordering of \(N\) phases: phase\(_1\), #(a\(_1\)), phase\(_2\), #(a\(_2\)), \ldots, phase\(_N\), #(a\(_N\)), phase\(_1\).

4. FORMALIZATION OF THE FOAM METHOD

In this section, we explain details of the transformations from annotated textual use-cases into formal structures that can be automatically verified. The whole process is depicted in Fig. 6. The input is the UCM—a collection of annotated textual use-cases.

The analyst decides which use-cases are ‘non-primary’ (denoted as Primary:FALSE) and which are ‘primary’ (this is the default). Non-primary use-cases are intended only for inclusion or as a precedence of other use-cases. Importantly, there exists no ordering of use-cases where a non-primary use-case would be executed as the last one. As an example, see the use-case MOD2_UC12 (Fig. A10) which describes a behavior shared by other use-cases in the system.

Primary use-cases, on the other hand, must be independently executable, i.e. there exists an ordering of use-cases where the primary use-case is executed as the last use-case (but there may be other orderings, where it is not the last one). As an example consider the use-case MOD1_UC1 (Fig. A4) describing how users register to the system. The behavior of this use-case is self-contained and therefore there exists an ordering such that no other use-case is executed after MOD1_UC1 finishes.

Based on precedence and inclusion, we define for each primary use-case \(u\) a restricted version of UCM (here denoted as \(rUCM\)) containing only such use-cases that can influence \(u\). For each \(rUCM\), we build a non-deterministic automaton—called Use-Case Behavior Automaton (UCBA). All UCBA's together represent the overall behavior of the whole UCM.

A UCBA is essentially an LTS with guards over boolean variables; thus, it can be straightforwardly encoded in specification languages of modern model-checkers (we discuss such an encoding for the NuSMV model-checker in Section 4.7). The verification of UCBA is performed with respect to the temporal logic formulae coming from the definition of temporal annotations.

4.1. Formalizing the input UCM

We start the formalization with the definition of an annotated textual use-case. This structure represents a use-case as close as possible to the way it is usually written down (e.g. as in Fig. 4). This means we explicitly capture use-case steps (along with annotations attached to them), extensions and variations.

Definition 1 (Annotated textual use-case). An annotated textual use-case is a tuple:

\( u = (S_u, W_u, w_u^m, Ext_u, Var_u, Flow_u, Temp_u) \),

where:

1. \( S_u \) is a set of all steps (sentences written in English).
2. \( W_u = \{w|w \subseteq S_u\} \) is a set of all scenarios of \( u \) where each scenario is a linearly ordered set with its total...
order $\leq_w$ such that scenarios do not share steps, i.e. $\forall u, w \in \mathcal{U}_w (w \neq w') \Rightarrow (u \cap w = \emptyset)$. Informally, a use-case is composed of scenarios, each scenario is a sequence of steps, each step is located in exactly one scenario.

(3) $w^0 \in \mathcal{W}_w$ is the main scenario.

(4) $Ext_u : \mathcal{W}_u \mapsto \mathcal{S}_u$ is a mapping function that assigns extensions to steps, i.e. $w' \in \mathcal{W}_u$ is an extension of $w \in \mathcal{W}_u$ from step $s \in \mathcal{W}_u$ if $Ext_u(w') = s$. Informally, extensions are attached to steps, forming a tree.

(5) $Var_u : \mathcal{W}_u \mapsto \mathcal{S}_u$ is a mapping function that assigns variations to steps, i.e. $w' \in \mathcal{W}_u$ is a variation of $w \in \mathcal{W}_u$ from step $s \in \mathcal{W}_u$ if $Var_u(w') = s$. Informally, variations are attached to steps, forming a tree.

(6) $Flow_u : \mathcal{S}_u \mapsto 2^\mathcal{S}$ is a function that assigns a set of flow annotations to each step ($2^\mathcal{S}$ denotes a set of all flow annotations; $2^\mathcal{W}$ denotes the set of all subsets of $\mathcal{W}$).

(7) $Temp_u : \mathcal{S}_u \mapsto 2^\mathcal{T}$ is a function that assigns a set of temporal annotations to each step ($\mathcal{T}$ denotes a set of all temporal annotations).

Further, we say that a use-case is well-formed if the following structural constraints are not violated. These rules follow the common practice of writing use-cases [1, 11, 13, 14], to help keep use-cases well-separated, comprehensible and of well-understood semantics.

(1) The annotations #(abort) and #(goto) can only be attached to the last step of a variation or extension. (Otherwise, there would be unreachable steps.)

(2) The annotation #(guard) is attached only to the first step of an extension or variation, while multiple #(guard) annotations for a single step are not allowed. (Because a guard defines the condition for executing a variation or an extension.)

(3) The main scenario of a primary use-case does not contain any #(goto), #(abort) or #(guard) annotations. (Guards are intended for branches only; the main scenario is also known as the main success scenario, thus no aborts and no loops are allowed. This is a good practice suggested by use-case practitioners such as Larman [11] and Cockburn [1].)

(4) The following types of flow annotations cannot be combined in a single step: #(abort), #(goto), and #(include). Annotations #(abort) and #(include) can only be used once in a single step. It is allowed to use multiple #(mark) annotations in a single step.

Now, we define UCM as a collection of use-cases accompanied with a precedence relation over use-cases. UCM thus represents the textually specified overall behavior of a system.

**Definition 2 (UCM).** A UCM is a tuple:

$$\mathcal{M} = (\mathcal{U}_\mathcal{M}, \mathcal{U}_\mathcal{M}^p, Prec_\mathcal{M})$$

(1) $\mathcal{U}_\mathcal{M}$ is a set of use-cases;

(2) $\mathcal{U}_\mathcal{M}^p \subseteq \mathcal{U}_\mathcal{M}$ is a set of primary use-cases;

(3) $Prec_\mathcal{M} \subseteq \mathcal{U}_\mathcal{M} \times \mathcal{U}_\mathcal{M}$ is a precedence relation on use-cases.

If a use-case $u$ precedes a use-case $u'$ (denoted as $(u', u) \in Prec_\mathcal{M}$), it means that whenever $u'$ is executed, $u$ must have been executed before.

Moreover, if it holds that $(u_3, u_2) \in Prec_\mathcal{M}$, $(u_3, u_1) \in Prec_\mathcal{M}$, $(u_2, u_1) \not\in Prec_\mathcal{M}$, and $(u_1, u_2) \not\in Prec_\mathcal{M}$, then both use-cases $u_1$ and $u_2$ have to be executed before use-case $u_3$ (in any order).

For each primary use-case $u$, there always exists an ordering of use-cases where $u$ is executed as the last use-case (but there may well be other orderings, where $u$ is followed by some use-case). However, for a non-primary use-case $v$, there is no such ordering that would execute $v$ as the last use-case. For example, consider $u_1$ is a non-primary use-case and $u_2, u_3$ are primary use-cases. If $(u_2, u_1) \in Prec_\mathcal{M}$, $(u_3, u_1) \in Prec_\mathcal{M}$, $(u_2, u_3) \not\in Prec_\mathcal{M}$ and $(u_3, u_2) \not\in Prec_\mathcal{M}$, then there are only four possible sequences allowed: $[u_1, u_2], [u_1, u_3], [u_2]$ and $[u_3]$.

In the rest of the text, we assume only UCMs with well-formed use-cases.

**Definition 3 (Include relation).** Let $\mathcal{U}_\mathcal{M}$ be a set of use-cases. Let $u_1 \in \mathcal{U}_\mathcal{M}$ and $u_2 \in \mathcal{U}_\mathcal{M}$ be two use-cases. We define the include relation $Inc_\mathcal{M} \subseteq \mathcal{U}_\mathcal{M} \times \mathcal{U}_\mathcal{M}$ as follows:

$$\exists s \in \mathcal{S}_u, \#(include: u_2) \in Flow_{u_1}(s).$$

In other words, the include relation is defined using the #(include) annotation attached to a use-case step.

**Definition 4 (Use-cases related by precedence and inclusion).** Let $Prec_\mathcal{M}^* \subseteq \mathcal{U}_\mathcal{M} \times \mathcal{U}_\mathcal{M}$ be a transitive closure of $Prec_\mathcal{M}$.

Let $Inc_\mathcal{M}^* \subseteq \mathcal{U}_\mathcal{M} \times \mathcal{U}_\mathcal{M}$ be a transitive closure of $Inc_\mathcal{M}$. Use-cases $u_1$ and $u_2$ are related by precedence and inclusion if and only if:

$$(u_1, u_2) \in Inc_\mathcal{M}^* \lor (u_1, u_2) \in Prec_\mathcal{M}^*.$$
Definition 6 (Use-cases sharing an identifier). Let \( u_1 \) and \( u_2 \) be two use-cases. We define a relation \( Sa_M \subseteq U_M \times U_M \) and we say that \( u_1 \) and \( u_2 \) are related through \( Sa_M \) if and only if they share an annotation identifier.

\[
(u_1, u_2) \in Sa_M \iff ID_{u_1} \cap ID_{u_2} \neq \emptyset.
\]

Definition 7 (Use-cases related by annotations). Let \( Sa^*_M \subseteq U_M \times U_M \) be a transitive closure of \( Sa_M \). Use-cases \( u_1 \) and \( u_2 \) are related by annotations, if and only if:

\[
(u_1, u_2) \in Sa^*_M.
\]

Note that use-cases can influence each other only if they are related by annotations. Sequencing of use-cases not related by annotations has no influence on validity of temporal invariants that are verified. Thus, they can be verified independently.

FOAM checks whether use-cases in a particular UCM that are related by annotations are also related by precedence and inclusion. If this does not hold, the error is reported prior to the actual verification. The reason is that if use-cases are related by annotations and not related by precedence and inclusion, FOAM would have to explore all possible orderings of such use-cases.

Definition 8 (rUCM to be verified). Let \( Prec^*_M \subseteq U_M \times U_M \) be a transitive closure of \( Prec_M \).

The rUCM for a use-case to be verified \( M \) for a primary use-case \( u \in M \) is defined as follows: \( M = (U_M, UC, Prec_M) \), where:

1. \( U_M = U^e_M \cup \{Inc^*_M(u_x), \exists u_x \in U^e_M\} \);
2. \( U^e_M = Prec^*_M(u) \cup \{u\} \);
3. \( Prec_M = \{(x, y) \in Prec_M, x \in U^e_M\} \).

Each rUCM \( M \) is used for verification of a single primary use-case \( u \) from UCM \( M \). It is an rUCM \( M \), that (i) contains only directly executed use-cases \( U^e_M \) and use-cases that are (transitively) included in these use-cases, (ii) the set of directly executed use-cases \( U^e_M \) contains the use-case \( u \) and those use-cases that (transitively) precede the use-case \( u \), all orderings of the use-cases from \( U^e_M \) will be explored during the verification, (iii) rUCM has a restricted version of the precedence relation in UCM \( M \).

As explained above, primary use-cases not related by precedence and inclusion can be verified independently. Moreover, we require that if a use-case \( u_i \) is included in a use-case \( u \), the use-case \( u \) must satisfy all precedences of the use-case \( u_i \). That is, the set of use-cases that must precede the use-case \( u_i \) must be a subset of the set of use-cases that must precede the use-case \( u \). Thus, precedences of included use-cases need not to be taken into the account explicitly.

rUCM for a primary use-case \( u \) includes all the information necessary for verification of scenario corresponding to \( u \). In such a scenario, all precedences of \( u \) must be executed before executing \( u \). These precedences may have multiple possible orderings. After executing \( u \), no subsequent steps follow.

The notion of rUCM is needed for better scalability of FOAM. Instead of building a single automaton from the whole UCM (as presented in [5, 6]), we construct multiple rUCM for each primary use-case. Without this restriction, the constructed automaton would also consider the permutations of use-cases that do not depend on each other. These redundant orderings would only slow down the verification without adding any additional benefit. As our tests show (see Section 6), the verification complexity grows exponentially with the number of use-cases. Unlike UCM, each rUCM is small for real-life specifications as it grows only with the number of preceding use-cases, which typically does not go over 4 (see Example A13). Thus, the method scales linearly with the size of UCM. Moreover, the verification can run in parallel.

### 4.2. Formalizing the UCBA

We can now focus on the verification of a single rUCM. In FOAM, we transform rUCM into UCBA, which has well-defined semantics and can be rather directly used as an input to standard model-checkers. UCBA is defined as follows.

Definition 9 (UCBA). A UCBA is a tuple:

\[
A = (V, init_0, \tau, B, AP, Val, Lab, Guards),
\]

where:

1. \( V \) is a set of states;
2. \( init_0 \in V \) is the initial state;
3. \( \tau \subseteq V \times V \) are transitions;
4. \( B \) is a set of boolean variables;
5. \( AP \) is a set of atomic propositions;
6. \( Val : \tau \mapsto 2^{(B \times \{true, false\})} \) are actions (valuations) on transitions that assign values to boolean variables in \( B \);
7. \( Lab : V \mapsto 2^{AP} \) is labeling of states by temporal properties;
8. \( Guards : \tau \mapsto 2^L \) are guards on transitions (a guard \( g \in L \) is a propositional logic formula with variables from \( B \)).

The semantics of UCBA is the following:

1. the execution starts in state \( init_0 \);
2. the transition to another state is by non-deterministic choice among outgoing transitions, whose all guards are satisfied;
3. upon the transition, the boolean variables of the automaton are updated based on the actions associated with the transition;
4. for the sake of model checking, the function \( Lab \) gives the atomic propositions that hold in a particular state.
4.3. Building UCBA: step #1

Having provided the definition of rUCM and UCBA, we now show UCBA construction from rUCM. It should be noted that by constructing UCBA, we give a precise semantics to the rUCM. This transformation is performed in two steps. In the first step below, we describe the automaton with the help of inference rules (in the form premise/conclusion). The rules put logical constraints on UCBA based on the input rUCM. In other words, the inference rules provide a logical theory, the model of which is UCBA.

The basic UCBA structure constructed from use-cases \(u_1, \ldots, u_n\) is depicted in Fig. 7. There is an initial state \(init_0\) with branches to particular sub-automatons, each corresponding to one of the directly executable use-cases \(u_1, \ldots, u_n\). The transitions to the sub-automatons are guarded by formulae that reflect the precedence constraints. This way, UCBA captures the non-determinism in sequencing the use-cases. After the sequence is completed, UCBA proceeds to the final state \(succ_0\), where a cycle is formed to generate infinite traces as typically required by model-checkers.

In the inference rules, we use for brevity reasons the notation \(s \rightarrow s'\) to denote the existence of states \(s\) and \(s'\) and the existence of a transition between them, i.e. \(s, s' \in V \land (s, s') \in \tau\). Additionally, we use the notation \(s \xrightarrow{G} s'\) to additionally state that the \(G \in 2^C\) is a subset of guards on transition \(t = (s, s') \in \tau\), i.e. that \(G \subseteq Guards(t)\); and we use the notation \(s \xrightarrow{V} s'\) to additionally state that the \(V \in 2^{(B \times \{true, false\})}\) is a subset of actions on \(t\), i.e. that \(V \subseteq Val(t)\).

The rules are as follows; each of them is accompanied by an explanation.

**Rule 1** (Representing steps). Every use-case step \(x\) is represented in the automaton \(A\) as exactly five states connected with transitions, forming a chain:

1. \(x^{in}\) represents the state before \(x\) has been executed.
2. \(x^{var}\) is the source state of all variations attached to \(x\).
3. \(x^{imp}\) is the target state of a \(#(\text{goto}:x)\) annotation.
4. \(x^{ext}\) is the source state of all extensions attached to \(x\).

![FIGURE 7. UCBA constructed from use-cases \(u_1, \ldots, u_n\).](image)

**Rule 2** (Representing scenarios). Let \(w \in W_{u}\) be a scenario containing steps \(x_1 \leq w \cdots \leq x_n\) linearly ordered using its total order \(\leq_w\). Then in \(A\) we connect the individual steps according to the order imposed by the \(\leq_w\) relation.

\[
u \in U_M, \; w \in W_{u}, \; x_1 \leq_w \cdots \leq_w x_n\]

Therefore, by applying this rule, we expand \(n\) use-case steps into \(5n\) states in the automaton \(A\) as depicted in Fig. 8. It is also important to stress that the actual ‘action’ of the original step \(x\) is represented as the transition \(x^{imp} \rightarrow x^{ext}\). For example, this transition is used for assigning boolean variables created from \(\#(\text{mark})\) annotations, as we show in Rule 8.

**Rule 3** (Handling variations). Let \(w \in W_{u}\) be a variation from step \(x\) which contains steps \(y_1, \ldots, y_n\). Then we connect \(w\) (state \(y_1^{in}\)) to its parent (state \(x^{var}\)). If the variation is conditioned by a \(#(\text{guard})\) annotation, we add this as a guard. See also Fig. 9.

\[
u \in U_M, \; w \in W_{u}, \; w = \{y_1, \ldots, y_n\}, \; Var_u(w) = x, \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \Quad

**Rule 4** (Handling extensions). Let \( w \in W_u \) be an extension from step \( x \) which contains steps \( y_1, \ldots, y_n \). Then we connect \( w \) (state \( y^n \)) to its parent (state \( x^\text{ext} \)). If the extension is conditioned by a \( \#(\text{guard}) \) annotation, we add this as a guard. See also Fig. 9.

\[
u \in U_M, w \in W_u, w = \{y_1, \ldots, y_n\}, Ext_u(w) = x,\]
\[
G_E = \{g | \#(\text{guard}:g) \in Flow_u(y_1)\}.
\]

\[x^\text{ext} \xrightarrow{[G_E]} y^\text{in}_1\]

**Rule 5** (Continuation from scenarios). In Rules 3 and 4, we redirected the execution from the step \( x \) to its variations and/or extensions. In this rule, we have to connect all scenarios back to \( x \) once they are finished. An exception is made, if the last step of a scenario contains either the \( \#(\text{goto}) \) or \( \#(\text{abort}) \) annotation. This situations are handled separately in Rules 6 and 7. In Fig. 9, we can see that the last state \( y_{n}^\text{out} \) within a scenario is connected back to the state \( x^\text{out} \). It also means, that if a variation is executed, then the execution continues at the end of the step \( x \), i.e. the step \( x \) itself is skipped together with all of its extensions. This is consistent with our definition that a variation replaces the behavior of the step. Extensions, on the other hand, are executed after the step \( x \) has already been executed.

\[
u \in U_M, w \in W_u, x = Var_u(w) \vee x = Ext_u(w),\]
\[
w = \{y_1, \ldots, y_n\}, \#(\text{abort}) \notin Flow_u(y_n),\]
\[
\#(\text{goto}:y) \notin Flow_u(y_n)\]
\[
y_{n}^\text{out} \rightarrow y_{\text{imp}}^\text{out}\]

**Rule 6** (Handling goto annotations). Let \( \#(\text{goto}:y) \) be a goto annotation of a use-case step \( x \) and \( y \) be another step in the same use-case. We handle the annotation by jumping to the \( y_{\text{jmp}} \) location as depicted in Fig. 10. This means that variations of the step \( y \) will be skipped, however, extensions are still executed. If a need be, the execution can jump directly to a particular variation (e.g. \( \#(\text{goto}:na) \) jumps to the variation \( na \) of the step \( n \)).

\[
u \in U_M, w \in W_u, x = S_u, \#(\text{goto}:y) \in Flow_u(x)\]
\[
x^\text{out} \xrightarrow{y_{\text{jm}p}} y_{\text{imp}}^\text{out}\]

**Rule 7** (Handling abort annotations). For each use-case step annotated with an annotation \( \#(\text{abort}) \), we add a transition which introduces an infinite loop. This is usual in temporal logic model checking. An temporal formula is satisfied by an infinite sequence of truth evaluations of variables. By introducing the infinite loop from a particular state, an infinite sequence without any progress from the state is introduced. This corresponds to aborting the computation in the state. Note that the automaton is not seen by the user, it is an intermediate structure between the user-understandable UCM and the input for a model-checker.
The abort-loop is skipped only when returning from an included use-case (see Rule 10). Both situations are depicted in Fig. 11.

\[
u \in U_M, x \in S_u, \#(\text{abort}) \in Flow_u(x) \quad x^{\text{out}} \rightarrow x^{\text{out}}\]

**Rule 8** (Handling mark annotations). We use \#(mark) annotations together with the \#(guard) annotations to control branching within use-cases. This is implemented using globally accessible boolean variables that are set to true when a \#(mark) annotation is visited during the execution. Since the actual ‘action’ of any step \(x\) happens on the transition \(x^{\text{imp}} \rightarrow x^{\text{ext}}\), also the boolean variable is set to true on this transition and remains true forever.

\[
u \in U_M, x \in S_u, \#(\text{mark}; b) \in Flow_u(x) \quad x^{\text{imp}} \xrightarrow{\{b \rightarrow \text{true}\}} x^{\text{ext}}\]

**Rule 9** (Resolution of includes (calling a procedure)). In FOAM, the include relationship among use-cases is expressed using \#(include) annotations. In order to implement include operations within UCBA, we add a set of boolean variables \(\text{incl}_{x,c}\), where \(\text{incl}_{x,c} = \text{true}\) if a use-case \(c\) has been called directly from a step \(x\) of a use-case \(u\). This is depicted in Fig. 12. Later, in Rule 10, we connect the use-case \(c\) back to \(x\), possibly from multiple scenarios of \(c\).

Let \(x\) be a use-case step annotated with \#(include:\(c\)). We disable execution of the transition \(x^{\text{imp}} \rightarrow x^{\text{ext}}\) by adding a [false] guard (because the use-case \(c\) will be called instead). Then we connect \(x^{\text{imp}}\) to the initial state \(y_1^{\text{in}}\) of the use-case \(c\) using a new transition, which sets the variable \(\text{incl}_{x,c}\) to true.

\[
u, c \in U_M, x \in S_u, \#(\text{include}; c) \in Flow_u(x), \quad w^m = \{y_1, \ldots, y_n\} \quad \begin{align*}
x^{\text{imp}} & \xrightarrow{\{\text{incl}_{x,c} \rightarrow \text{true}\}} y_1^{\text{in}},
x^{\text{imp}} \xrightarrow{\{\text{false}\}} x^{\text{ext}} \end{align*}\]

**FIGURE 11.** Rule 7 explained: an abort is represented as an infinite loop (normal execution) or returns the execution to the original use-case (when executing an included use-case).

**FIGURE 12.** Rule 9 explained: including a use-case \(c\) from step \(x\) of a use-case \(u\).

**Rule 10** (Resolution of includes (return)). Similarly to Rule 9, let us denote \(c\) as a use-case that has been included directly from a step \(x\) of a use-case \(u\), which is expressed in UCM using the \#(include:\(c\)) annotation. The purpose of this rule is to connect \(c\) back to the original use-case \(u\) which is depicted in Fig. 13.

Due to our assumption that a UCM contains only well-formed use-cases, the use-case \(c\) does not contain any \#(goto) or \#(abort) annotation within its main scenario \(w^m\). Therefore, we always need to connect the main scenario \(w^m\) back to the step \(x\) (state \(x^{\text{ext}}\)) and optionally other scenarios that abort the execution of the use-case \(c\). A scenario \(w\) aborts the use-case if it contains the \#(abort) annotations in its last step \(y_n \in w\).

Note, that the transition \((x^{\text{imp}} \rightarrow x^{\text{ext}})\) has been previously disabled by Rule 9. After returning to \(x^{\text{ext}}\), all the extensions of the step \(x\) have the chance to be executed.

Remember that in Rule 9 we set the \(\text{incl}_{x,c}\) variable to true when calling the included use-case. In this rule, we need to switch \(\text{incl}_{x,c}\) back to false.

Moreover, all the states that should be connected back to \(x^{\text{ext}}\) may already contain outgoing transitions to some states \(z_1, \ldots, z_k\). In fact, this situation occurs due to Rule 7 which adds an infinite loop to any aborting state. However, an abort in an included use-case should return the execution to the original use-cases instead of aborting the whole execution. Therefore, we add a new guard \([\neg \text{incl}_{x,c}]\) to each transition \((y_n^{\text{out}} \rightarrow z_1), \ldots, (y_n^{\text{out}} \rightarrow z_k)\), where \(y_n\) is a state that should be connected back to \(x^{\text{ext}}\); we also add a guard \([\text{incl}_{x,c}]\) on each returning transition, so that the transition is enabled just within the scope
It should be noted that the variable exec_u is also set to true when the use-case is called by inclusion from other use-case. It means, that if a use-case has already been called through inclusion, it is not executed again through precedence on the same trace. However, the same use-case can be called multiple times through inclusion.

\[
G = \{\text{exec}_u | \text{exec}_u \in U^P_M\} \quad \text{init}_0 \xrightarrow{[G]} \text{succ}_0 \rightarrow \text{succ}_0
\]

Rule 13 (Atomic propositions). Temporal annotations attached to use-case step \(x\) are translated to UCBA as atomic propositions attached to a corresponding \(x^{imp}\) state. The state \(x^{imp}\) is circumvented only when executing a variation attached to the step \(x\).

\[
x \in S_u, u \in U^P_M \quad Lab(x^{imp}) = Temp_u(x)
\]

4.4. Building UCBA: step #2

In this step, we address the issue of a different semantics of guards in use-case steps and guards in the constructed UCBA. If the user adds a guard to a variation/extension in a use-case, he/she intuitively assumes that other branches from the same step are not executed when the guard holds. It is natural that the use-cases specification mixes guarded and unguarded branches. This needs to be explicitly captured in UCBA.

**Summary of our assumptions on use-cases:**

1. A non-deterministic choice is assumed among the default step and its unguarded branches. This is the case of a step with unguarded variations and/or extensions only.
2. A non-deterministic choice is assumed among guarded branches that contain non-disjunctive guards. In other words, if there are multiple guarded branches the guards of which hold, they are executed non-deterministically.
3. Mutual exclusivity is assumed among the default step and unguarded branches on one hand, and the guarded branches on the other hand. More precisely, first, the guarded branches are checked whether any of them can be executed. If none of the guards hold, an unguarded branch is selected non-deterministically for execution.
The UCBA that we have constructed in step #1 follows the semantics of (1) and (2), but not of (3). To address (3), we need to additionally introduce the guards for the default step and the unguarded variations and extensions so as the mutual exclusivity holds.

The example depicted in Fig. 14 shows a single use-case step $x$ represented as five states $(x^{in}, x^{var}, x^{inp}, x^{ext}, x^{out})$. There are three variations branching from the state $x^{var}$ while only two of the transitions are guarded using the guards $[g_1]$ and $[g_2]$. Similarly, there are two guarded and one unguarded extensions from the state $x^{ext}$. Our solution computes new guards as a negation of the existing ones. In this case, we need to additionally introduce the guards for the default step and of computing the missing guards.

If $G^x_V \neq \emptyset$, we add to each unguarded transition from $x^{ext}$ a guard:

$$\bigwedge_{f \in G^x_V} (\neg f).$$

It should be noted that after applying this operation, there is no non-deterministic branching in the automaton that would mix guarded and unguarded transitions. Either there are no guards or all transitions have a properly defined guard.

### 4.5. Deriving temporal formulae from TADL

Now we show the instantiation of temporal logic formulae based on the rUCM and TADL (user-defined temporal annotations). Each temporal annotation used in rUCM has the form $\#(a:s)$, where $a$ is the name of the annotation and $s$ is the qualifier of the annotation in the use-case. Let $tadl$ be a TADL definition for the annotation name $a$. Such annotation therefore contributes a set of formulae $F^{\#(a:s)} = \bigcup_{i=1}^{n} F^{tadl}_{i}[_/ #(_(a:s))]$, where $F^{tadl}_{i}$ is the $i$th logical formula defined in the template $tadl$ and where $[/_ #(_(a:s))]$ denotes renaming of each variable (represented by placeholder _) in the formula to the form _s.

In other words, whenever an annotation $a$ appears in the text with the parameter $s$, we need to instantiate all the corresponding formulae from the $tadl$ template by replacing the template variables with new variables containing $s$ as a qualifier.

The temporal properties to be verified by the model-checker are obtained as union over all the sets $F^{\#(a:s)}$ contributed by annotations used in rUCM.

**Example:** Consider the following TADL template with annotations $\#(a)$ and $\#(b)$.

**Annotations:** $a, b$

**LTL** $\Box g \rightarrow O(a)$

**CTL AG** $a \rightarrow EF(b)$

If the textual specification contains annotations $\#(a(x))$, $\#(a(y))$ and $\#(b(y))$ the set of formulae will be constructed as a union of:

**LTL** $\Box g \rightarrow O(a(x))$

**CTL AG** $a \rightarrow EF(b)$

which yields the formulae:

**LTL** $\Box g \rightarrow O(a(x))$

**CTL AG** $a \rightarrow EF(b)$

**4.6. Refining operators**

A formula defined in TADL describes a temporal constraint on top of use-case steps. Obviously, Rule 1 causes an increase
in the granularity of states which subsequently affects the semantics of temporal operators. This can be demonstrated in Fig. 15. A single use-case step from the UCM corresponds to multiple LTS states. However, the original event (e.g. a communication between actors) corresponds only to the jmp state in the UCBA. The other states (in, var, ext and out) emerged during the construction of the UCBA. Therefore, we have to refine the temporal formulae from TADL by translating all temporal operators to a slightly longer form as defined in Table 1. In Fig. 15, the CTL formula \( a \land X(b) \), had to be refined to a new form \( a \land X[\neg jmp\ U\ jmp\ &\ b] \). The variable jmp is used as a marker to indicate, where the use-case ‘actions’ take place, which is in every \( x^{imp} \) (for any step \( x \)) and also in any looping state (i.e. state \( s \in V: \exists (s \rightarrow s) \)).

Also all the model-checking-related variables can only be set to true in the \( x^{imp} \) states. In Fig. 15, the model-checking-related variables \( tadl_a \) and \( tadl_b \), corresponding to annotations \#(a) and \#(b), are set to true only in the states \( x^{imp} \) and \( y^{imp} \).

### 4.7. Verification using NuSMV

We have implemented a verification of UCBA using the NuSMV model-checker [12]. Since UCBA is defined as an LTS structure, it is easy to employ any other state-of-the-art model checker for this task. We have opted for NuSMV because it supports analysis of synchronous and asynchronus systems using a rich set of temporal logics—CTL, LTL and PLTL.

Transformation of UCBA into the NuSMV input language is straightforward (Fig. 16). There is a NuSMV variable state, which corresponds to the current state. Transitions of UCBA are reflected as NuSMV rules setting the state variable based on the source state and guarding formulae.

**TABLE 1.** Translated temporal operators from TADL to UCBA.

<table>
<thead>
<tr>
<th>TADL</th>
<th>CTL in UCBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>( AX(\varphi) )</td>
<td>( AX A[\neg jmp\ U\ (jmp\ &amp;\ \varphi)] )</td>
</tr>
<tr>
<td>( AG(\varphi) )</td>
<td>( AG(jmp \rightarrow \varphi) )</td>
</tr>
<tr>
<td>( AF(\varphi) )</td>
<td>( AF(\varphi) )</td>
</tr>
<tr>
<td>( A[a U \beta] )</td>
<td>( A[(jmp \rightarrow a) U \beta] )</td>
</tr>
<tr>
<td>( EX(\varphi) )</td>
<td>( EX E[\neg jmp\ U\ (jmp\ &amp;\ \varphi)] )</td>
</tr>
<tr>
<td>( EG(\varphi) )</td>
<td>( EG(jmp \rightarrow \varphi) )</td>
</tr>
<tr>
<td>( EF(\varphi) )</td>
<td>( EF(\varphi) )</td>
</tr>
<tr>
<td>( E[a U \beta] )</td>
<td>( E[(jmp \rightarrow a) U \beta] )</td>
</tr>
<tr>
<td>TADL</td>
<td>LTL in UCBA</td>
</tr>
<tr>
<td>( X(\varphi) )</td>
<td>( X[\neg jmp\ U\ (jmp\ &amp;\ \varphi)] )</td>
</tr>
<tr>
<td>( G(\varphi) )</td>
<td>( G(jmp \rightarrow \varphi) )</td>
</tr>
<tr>
<td>( F(\varphi) )</td>
<td>( F(\varphi) )</td>
</tr>
<tr>
<td>( aU\beta )</td>
<td>( (jmp \rightarrow a) U \beta )</td>
</tr>
<tr>
<td>( aR\beta )</td>
<td>( aR(jmp \rightarrow \beta) )</td>
</tr>
<tr>
<td>TADL</td>
<td>PLTL in UCBA</td>
</tr>
<tr>
<td>( Y(\varphi) )</td>
<td>( Y[\neg jmp\ S\ (jmp\ &amp;\ \varphi)] )</td>
</tr>
<tr>
<td>( H(\varphi) )</td>
<td>( H(jmp \rightarrow \varphi) )</td>
</tr>
<tr>
<td>( O(\varphi) )</td>
<td>( O(\varphi) )</td>
</tr>
<tr>
<td>( (aS\beta) )</td>
<td>( (jmp \rightarrow a) S\beta )</td>
</tr>
<tr>
<td>( (a\beta) )</td>
<td>( aT(jmp \rightarrow \beta) )</td>
</tr>
</tbody>
</table>

Note that the only operators unaffected by the refinement are \( F(\varphi) \), \( O(\varphi) \), \( AF(\varphi) \) and \( EF(\varphi) \).

**Boolean variables:** The transformation from LTS to NuSMV generates variables that represent: (i) temporal annotations (set to true at some point and immediately to false in the next state), (ii) variables controlling inclusion of use-cases (incl\(_{i,i'}\)) and scheduling (exec\(_{i,i'}\)), (iii) \#(mark) annotations (set to true at some point, remaining true until the final state).

**Non-deterministic guards:** NuSMV does not support a guarded non-deterministic choice between rules. However, an unguarded non-deterministic choice is supported (e.g. state \( = s0 : \{ s1, s2, s3 \} \)). Therefore, we emulate non-deterministic guards in the following way: (i) target state is non-deterministically chosen among all target states (regardless of the guards), (ii) transition to the selected target state is taken if the guard holds, if it does not, transition back to the source is taken and the process is repeated. Such operation requires splitting the original transition \( x \rightarrow y \) into \( x \rightarrow x_{guard} \rightarrow y \) and also adding the back-transition \( x_{guard} \rightarrow x \). The new state \( x_{guard} \) is used in NuSMV to deterministically check the negated version of an original guarding condition. In order to avoid infinite loops, fairness is enforced using a dedicated FAIRNESS condition featured by NuSMV.
5. EXPRESSIVENESS OF FOAM

To reflect the common guidelines in creating use-cases, FOAM features a number of restrictions on the control flow annotations—rules regarding the placement of \#(goto), \#(abort) and \#(guard) (e.g. guards are allowed only at the beginning of variations and extensions). In this section, we show that these restrictions do not actually impact the overall theoretical expressive power of the formalism. We show this by proving that a Kripke structure [15] and a related temporal logic formula, which form a typical input in model-checking theory, can be transformed into a use-case and an annotation group while still preserving satisfiability. In particular, we show this for Kripke structures that have one initial state and a set of states in which all computation eventually ends in an infinite cycle. The first assumption does not cause any loss of generality, as we can always add a single initial state. The second assumption restricts us to describing functionality that eventually ends, which is one of the main characteristics of scenarios that are being described by use-cases.3

The claim showing the expressive power is formalized by the theorem below.

**Theorem 1.** Let \( K \) be a Kripke structure such that it has only one initial state and a set of states, in which all computations eventually end in an infinite cycle. Let \( F \) be an LTL or CTL formula. Then there exists a UCM \( M \) and a set of related annotation groups \( G \) such that \( F \) is satisfied in \( K \) if and only if \( M \) is correct with respect to \( G \).

**Proof.** We split the proof into two parts—the expressiveness part and the correctness part. In the expressiveness part, we construct the UCM \( M \) and the set of related annotation groups \( G \) given the Kripke structure \( K \) and the formula \( F \). In the correctness part, we show how \( M \) and \( G \) is transformed to UCBA \( K' \) (a Kripke structure), and how the formula \( F \) is transformed to the formula \( F' \). We state that it holds that \( F \) in \( K \) and \( F' \) in \( K' \) are equisatisfiable. This follows from the construction of \( M \), the construction of \( G \), and the correctness of transformation described in Section 4. For brevity, we focus here on the basic idea of the proof.

We denote the initial state of \( K \) and set of states, in which all computation eventually end \( O \). We construct the UCM \( M \) with one primary use-case \( u \), no precedence constraints and the set of annotation groups \( G \) with one annotation group \( g \). We introduce a synthetic step \( s_{\text{start}} \) as the first step of the main scenario of \( u \). The remaining steps of \( u \) are defined using the edges in \( K \) (note that we treat \( K \) as an oriented graph) as follows. We choose a path \( p_m \) in \( K \), which starts in \( i \) and ends in a state \( o \in O \) (such path has to exist due to our assumptions) and for each edge of \( p_m \) we add a step to the main scenario of \( u \).

Now we iterate the following steps until all vertices in \( K \) have been processed (we deem vertices in \( p_m \) as already processed): We select a path \( p = v_1 \rightarrow \cdots \rightarrow v_n \) in \( K \) such that (i) it starts in some of the processed vertices, (ii) when not

---

3Allowing for valid use-cases with infinite cyclic functionality would be also possible (along with transformation from the Kripke structure), but it would make no sense to speak about a set of use-cases and use-case precedences; also the UCBA would have to be constructed differently, thus we treat use-cases in this article to be valid only when they have finite execution.
considering the last vertex of \( p \), the vertices of \( p \) are disjunctive,\(^4\) (iii) when not considering the first and last vertex of \( p \), the vertices have not yet been processed, (iv) the path cannot be made longer without violating (i)–(iii).

We define a variation of a step \( s \), which is found as follows: (i) \( s \) was added to \( \mathbb{M} \) because of an edge of \( K \) which originates in \( v_1 \) and (ii) \( s \) is not the first step in its scenario. For each edge of \( p \), we add a step to the variation. If it holds that \( v_i \in O \), we add an \( #(\)abort\() \) annotation to the last step. Finally, we mark all vertices in \( p \) processed.

When all vertices are processed, we connect variations in \( \mathbb{M} \) corresponding to branchings in Kripke structure \( K \) that do not end with the infinite loop but are connected to other branchings. These are the variations that do not have the last step annotated with an \( #(\)abort\() \). To connect variations, we use \#(goto) annotations. Note that in Kripke structure, a vertex to which a branch connects can introduce another branching (the vertex can have both more ingoing and more outgoing edges). In this case, the last step of each a such variation will be annotated by more \#(goto) annotations. Formally, let \( s \) in \( \mathbb{M} \) be a step that is the last in a variation and does not have an \#(abort) annotation. Let \( e_s = (v, v') \) be the corresponding edge in \( K \). We add an annotation \#(goto;e) to the step \( s \) for each step \( e \) that corresponds to an edge in \( K \) that goes from the vertex \( v' \).

The annotation group \( g \) is constructed as follows. A formula \( F \) is used as the corresponding temporal logic formula in \( g \). An annotation is introduced to the group for each distinct atomic proposition in the formula. A temporal annotation is attached to a step in the use-case \( u \), on condition that a corresponding atomic proposition has been associated with a vertex in \( K \) such that the vertex was the target of the step.

UCM \( \mathbb{M} \) has one \( rUCM = (u, u, \emptyset) \). From \( rUCM \), UCBA \( K' \) is built using the Rules 1...13 in Section 4.3. Next, from the input formula \( F \), the formula \( F' \) is built using the transformation described in Table 1. From the correctness of transformation UCM to UCBA and the correctness of transformation of temporal formula in Table 1 it holds that \( F \) in \( K \) and \( F' \) in \( K' \) are equisatisfiable.

6. EVALUATION OF SCALABILITY

In this section, we discuss scalability of FOAM with respect to industrial-size specifications. Obtaining industrial specifications is difficult due to intellectual property issues. Thus, we used freely available reference specifications to get a realistic idea about the size and complexity of the specification usually encountered. In particular, we have relied on the benchmark conducted in [13, 14]. The authors derived a referential specification based on the common characteristics of such specifications.

\(^4\)Vertices of given path are disjunctive if no vertex in the path repeats.

<table>
<thead>
<tr>
<th>Property analyzed</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>The average number of steps in main scenario</td>
<td>4.82, stdev = 2.41</td>
</tr>
<tr>
<td>Use-cases with extensions</td>
<td>72.1%</td>
</tr>
<tr>
<td>Number of extensions in a use-case</td>
<td>1.57, stdev = 1.88</td>
</tr>
<tr>
<td>Number of steps in extension</td>
<td>2.46, stdev = 1.61</td>
</tr>
<tr>
<td>Steps with validation actions</td>
<td>3.4%</td>
</tr>
<tr>
<td>Use-cases with preconditions</td>
<td>37.4%</td>
</tr>
<tr>
<td>Steps with reference to use-cases</td>
<td>6.4%</td>
</tr>
</tbody>
</table>

Based on the work of Alchimowicz et al. [13].

The benchmark covers 16 industrial specifications comprising 524 use-cases. The purpose was to create a referential specification reflecting typical patterns found in software projects. According to the authors, such specification can be used for presenting, testing and verifying methods and tools for use-case analysis. The published referential specification is available on-line.\(^5\)

Relevant properties are listed in Table 2 from which we selected the following: The average number of steps in the main scenario is 4.8, the average number of branches in a use-case is 1.6, the average size of a branch is 2.5 and the number of steps of validation nature is 4%.

The main factors influencing the complexity of a use-case specification can be characterized by the following five parameters:

1. \( u \) the number of use-cases in rUCM;
2. \( m \) the number of steps within the main scenario;
3. \( b \) the number of branches;
4. \( b' \) the length of a branch;
5. \( a \) the number of temporal annotations to be verified (e.g. create-use pairs) in the rUCM.

We conducted four experiments, each highlighting the dependency between the verification time \( t \) and two of the parameters. Each dependency is rendered as a 3D surface plot accompanied with a 2D projection. In other words, our experiments show several projections of the 6D space:

\[ (t \times u \times m \times bc \times b'l \times a'). \]

As our experiments indicate, the time required for the verification depends exponentially on the number of use-cases in a specification. Fortunately, real specifications, such as [10] depicted in Fig. A13, tend to be divisible into fairly small rUCM chunks. The complexity of the precedence relation within these chunks is usually very low. Typically, the use-cases are linearly ordered. Therefore, in our experiments, we decided to order our use-cases linearly with just a single cluster.

\(^5\)http://ucdb.cs.put.poznan.pl/benchmark/2.f.n/srs/.
of three parallel use-cases within the rUCM (3! = 6 possible orderings in every experiment).

For a given tuple of parameters \((u, m, bc, bl, a)\) in an experiment, we generate an rUCM with the following properties:

1. All generated use-cases are identical, i.e. yielding the same number of states, transitions and branches. The number of steps within a use-case is therefore: \(bc \times bl + m\).
2. All branches are attached to the step #2 in the main scenario.
3. Each branching scenario continues its execution in the main scenario instead of aborting.
4. Each temporal annotation added to the rUCM is either \#(create) annotation in step #1 of use-case #1 or \#(use) annotation in the last step of a randomly picked use-case. This way, the model-checker has to verify both consistent and inconsistent specifications which should mitigate the bias toward one of these alternatives.
5. We presume a precedence relation which yields \(\min(3, u)! \leq 6\) possible orderings of use-cases within the rUCM.

The generated NuSMV code is similar to the example in Appendix (Fig. A1). We used the following hardware setup for our test: CPU: Core2 Duo CPU P9600 2.53GHz; RAM: 4GiB RAM; OS: 64bit Linux with kernel 3.5.0; NuSMV version: NuSMV-zchaff-2.5.4-x86_64.

In each run of the experiment, the model-checker either successfully finished or we terminated it after 30min. We visualize these results in Figs 17–20, which scale to show the first 60s of the run. This makes it possible to see in a greater detail the breaking point after which the execution times start to quickly rise because of the exponential nature of the verification process. In fact, most of the experiments, the duration of which exceeded 60s, had to be forcibly terminated (given the 30 min timeout).

6.1. Experiment 1

This experiment shows how the verification time depends on the number of use-cases in rUCM (parameter \(u\)) and the number of states within the main scenario (parameter \(m\)). We set the parameters as follows: number of branches \(bc = 2\), length of each branch \(bl = 3\) (according to [14] where the average number of branches is 2.5) and number of properties to be verified \(a = 5\).

We can see in Fig. 17 that the verification time depends exponentially on both parameters \(u\) and \(m\). For example, given the memory constraints in our setup (4 GB), and scenarios with 20 steps, NuSMV successfully verified rUCM up to 16 use-cases.

6.2. Experiment 2

In this experiment (Fig. 18), we measured how FOAM deals with temporal annotations within the specification. We set the parameters as follows: size of the main scenario \(m = 5\), number of branches \(bc = 2\) and length of each branch \(bl = 3\). Then, we varied the parameter \(a\), which determines the number of generated create-use pairs. According to Fig. 5, each create-use pair is represented by three LTL/CTL
formulæ. It should be noted that we did not varied the length of temporal formulæ during this experiment. More complex formulæ would certainly further increase the verification time.

Our results depicted in Fig. 18 show that the verification time depends exponentially on the number of use-cases \( u \) as well as on the number of temporal annotations \( a \). For example, 13 use-cases with 100 temporal annotations can be verified under 1 min. However, we do not expect to find so many temporal annotations in real-life specifications.

6.3. Experiment 3

The goal of this experiment was to estimate how FOAM deals with use-cases containing multiple branches. Although it is not common for a use-case to have too many branches, we also tested such a scenario (Table 2). The surface-plot in Fig. 19 shows that model checking of a relatively rich rUCM is feasible. For example, we could model-check 15 use-cases containing 26 branches or 6 use-cases containing 80 branches under one minute. This leads us to the conclusion that the
number of branches is not a critical factor influencing the verification.

6.4. Experiment 4

For a firm number of use-cases ($u = 8$ in this experiment), we measured the verification time depending on the number of steps. Therefore, we combined parameters $m$ and $bc$ into a single plot (Fig. 20).

6.5. Summary of the experimental results

To summarize all experiments, an exponential growth of verification time can be seen in all diagrams. We can approximate each diagram using a surface:

$$f(x, y) = a \cdot e^{(bx+cy+d)},$$

for certain values of $a, b, c, d > 0$ and $c < 0$ that are different for each diagram.

The results show that by assuming three parallel use-cases in the precedence relation, we can successfully verify an rUCM with up to 17 use-cases. However, by increasing the value of other parameters, the boundary of a rapid exponential increase in verification time would be reached even sooner.

6.6. Limitations of the experiment

As mentioned earlier, all the generated use-cases in our experiments have the same characteristics (the same number of steps and branches), which may, even though not significantly, influence the results when a symbolic model-checker is used (the boundary in Fig. 19 is not continuous, nevertheless the overall behavior is not affected).

We also did not consider any flow annotations in the experiment. This should not be an issue for $(\text{abort})$, $(\text{include})$, $(\text{mark})$ and $(\text{guard})$ annotations. Aborts would only simplify the model by decreasing the length of some traces. Similarly, guards would decrease the number of non-deterministic choices. Using $(\text{include})$ annotations would have the same effect as increasing the number of steps in a use-case because the included use-case is simply inlined to the original use-case. However, at the moment, we cannot estimate the impact of $(\text{goto})$ annotations in the specification.

7. EVALUATION OF LEARNING CURVE

This section focuses on the ease of use in terms of learning curve.

We asked three first-year computer science postgraduate students without any prior knowledge of the FOAM method to annotate a set of use-cases. We measured the time required to learn the basic concepts of FOAM and the time required for annotating a typical use-case. More precisely, we split the whole process into a sequence of stages measured independently. Before performing the experiment, one of the FOAM authors selected a set of suitable use-cases and created an annotated referential specification which was then compared with the testers’ answers.
7.1. Selection of use-cases for the test

Just like in Section 6, we chose the specification [14] (Admission System version 2.0F quantitative) because it represents 16 industrial specifications distilled into 34 use-cases. The goal was to demonstrate all important aspects of the FOAM method using a minimal set of use-cases. Therefore, our selection criteria were as follows:

1. The set of use-cases should be closed on precedence and inclusion (transitively).
2. At least one use-case should contain the include relation.
3. There should be extensions or variations in most of the use-cases.
4. Some branches should contain aborts and jumps (goto).
5. Some scenarios should demonstrate the use of the open–close and create-use annotations.

For the sake of simplicity, we selected the following nine use-cases:

<table>
<thead>
<tr>
<th>UC1: Login to the system</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOD1_UC1: Register in the system</td>
</tr>
<tr>
<td>MOD1_UC2: Provide personal and education information</td>
</tr>
<tr>
<td>MOD1_UC3: Choose a major</td>
</tr>
<tr>
<td>MOD1_UC4: Assign an application fee to a major</td>
</tr>
<tr>
<td>MOD1_UC5: Check application status</td>
</tr>
<tr>
<td>MOD2_UC1: Create a new admission</td>
</tr>
<tr>
<td>MOD2_UC8: Add a new user</td>
</tr>
<tr>
<td>MOD2_UC12: Import admissions fees</td>
</tr>
</tbody>
</table>

Each of them can be found in Appendix. (Here we have included the final corrected version of the use-cases—details about the corrections are found in Section 8. We also provide a diagram, Fig. A12, showing the precedence and include relations among the use-cases.)

7.2. Method applied by testers

Three testers (TesterA, TesterB and TesterC) were asked to apply FOAM on the set of use-cases selected in Section 7.1 but without any annotations and precedence relation specified.

Additionally, they received another already annotated use-case6 as an example and an explanation of the flow and temporal annotations (Fig. A2).

Then, each of them was exposed to an oral explanation session about the method. The explanation was held without the presence of the other testers so that they could not influence each other by questions, etc. Finally, the testers applied the method in a sequence of stages without knowing the use-case order beforehand. We measured the time spent on each stage: (1) reading the use-cases, (2) ordering the use-cases according to the precedence relation, (3) adding flow annotations, (4) adding temporal annotations and (5) reporting on identified inconsistencies.

7.3. Feedback from testers

Table 3 summarizes the measured time spent by each tester on different stages of the use-case analysis.

### Table 3. Results of the case study.

<table>
<thead>
<tr>
<th></th>
<th>Author (min)</th>
<th>TesterA (min)</th>
<th>TesterB (min)</th>
<th>TesterC (min)</th>
<th>AVG(A,B,C) (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading</td>
<td>n/a</td>
<td>10</td>
<td>30</td>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>Precedence</td>
<td>12</td>
<td>6</td>
<td>5</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>Flow annot.</td>
<td>7 + 3</td>
<td>11</td>
<td>8</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>Temp. annot.</td>
<td>25</td>
<td>23</td>
<td>24</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>Total analysis</td>
<td>47</td>
<td>50</td>
<td>67</td>
<td>60</td>
<td>59</td>
</tr>
<tr>
<td>Analysis per UC</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Learning</td>
<td>n/a</td>
<td>13</td>
<td>15</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>47</td>
<td>63</td>
<td>82</td>
<td>78</td>
<td>74</td>
</tr>
</tbody>
</table>

7.3.1. Adding flow annotations

As can be seen from the table, there are minor differences between the testers. When we compared the annotations made by all the testers, we noticed significant similarities in their choice and placement of annotations. As expected, the precedence relation and flow annotations were the same for all of them and also the same as in the reference specification.

There were slight differences in the use of the #include (it is actually ambiguous whether the extension ‘user decided to contact admin’ in UC1 is an inclusion of MOD2_UC2 or an abort).

7.3.2. Adding temporal annotations

More interesting are the results of temporal annotations. The create-use pair was the most challenging one while almost all open-close annotations matched the referential solution. For example, comparing to the referential solution, none of the testers identified the #create:chosenMajor annotation. It seems that the notion of ‘transactions’ was more natural to them than a slightly vague notion of ‘data dependency’. The testers obviously avoided capturing data dependencies across multiple use-cases. Since a transaction does not usually cross boundaries of a single use-case, it was easier for them to correctly identify the #open and #close annotations.

In our case study, we showed them only MOD2_UC6 (Fig. A2) without demonstrating data dependencies across multiple use-cases.

7.3.3. Summary

In an interview after the experiment, all our testers gave us a positive feedback. They claimed that the process of annotating...
use-cases forced them to think about the dependencies from a different perspective compared with just simple reading. After annotating the text, they could easily spot important places in the text with respect to the temporal dependencies among use-case steps.

According to the testers’ responses, the time invested in learning FOAM can be considered negligible. Table 3 shows that annotating a single use-case takes approx. 6 min. This number is based on our sample-set of 36 use-cases (9 use-cases analyzed by 4 people). Unfortunately, we could not compute the standard deviation, because we did not measure the required time for each use-case individually.

We are aware that these results obtained from three testers are not statistically significant. Our intention was mostly to provide an initial estimate of the learning curve. We plan to pursue a more extensive study as a future work.

8. INCONSISTENCIES DETECTED DURING THE VERIFICATION

After collecting the responses from testers, we compiled a new version of the specification. Using our verification tool, we were able to detect inconsistencies that slipped through our manual review. (The referential specification in the Appendix already contains the updated version together with markings showing the corrections applied after the verification.)

Inconsistencies detected by visual inspection:

(1) The extensions 1a and 1b in MOD1_UC2 had to be attached to a different step compared with the original. Also the sequencing of actions in MOD1_UC2 had to be redefined properly by adding variations 4a2a and 4b2a (Fig. A5).
(2) MOD1_UC4 (Fig. A7)—instead of the variation 5a, the original specification defined an extension 5a. It is obvious that the ‘money transfer’ scenario is a variation of the ‘credit card payment’ scenario because either the former or the latter is executed, never both.

Inconsistencies detected by the tool:

(1) MOD1_UC1 (Fig. A4)—the tool detected the missing #(close:registration) which is necessary due to the infinite loop introduced by the #(goto:4) annotations in Extensions 5a, 5b, 5c. This type of error is not very serious. However, this way, FOAM framework guides us to explicitly state the boundaries of an ‘open–close’ transaction which increases clarity of the specification.
(2) MOD1_UC3 (Fig. A6)—in this case, our verification tool helped us detect that MOD1_UC3 cannot be executed independently because the annotation #(create:chosenMajor) from the step 3 would not have been followed by any #(use:chosenMajor). In other words, this use-case is actually a non-primary use-case.

The behavior of the use-case is included from other use-cases, yet it does not make sense to execute it independently. Nevertheless, it is not always possible to discover such inconsistencies. In our case, if there were an annotation #(use:chosenMajor) in step 4, the inconsistency would not have been detected.

(3) MOD1_UC4 (Fig. A7)—a missing Extension 5a3a was identified. The problem was that the money transfer may fail due to the included MOD2_UC12 (Extension 2a). If it fails, #(create:registeredMoney) will not be visited and therefore the subsequent #(use:registeredMoney) will cause a verification error.

It can be assumed that such a pattern is common in large specifications if the structure of use-cases changes over time (e.g. Extension 2a in MOD2_UC12 might have been added later in the project by a different analyst). When the dependencies are captured formally, such issues can be spotted easily.

(4) MOD2_UC1 (Fig. A9)—this use-case also contained an interesting type of error. The main goal of this use-case is to perform a transaction concerning the admissionForm. However, there were two Extensions 3a, 4a introducing an infinite loop. When writing a use-case specification, we normally assume that the execution eventually reaches either the success state or some abort state. Sometimes, however, we would like to limit the number of executions of a given loop (e.g. a password may be entered maximum N times). In FOAM, this can be achieved with a guard that can disable the execution of an already visited branch.

In MOD2_UC1, the first problem identified by the verification tool was a missing annotation #(close:admissionForm) in case the execution loops forever. However, tempting it might seem, we cannot just add this annotation to steps 3a1 and 4a1. In that case, #(close:admissionForm) would be visited repeatedly, which is forbidden by the definition of open–close (Fig. 5). The best option here was to break the infinite loop by introducing a guarded variation for each of the looping extensions.

(5) MOD2_UC12 (Fig. A10)—here, we also detected that this use-case is not primary, similarly to the use-case MOD1_UC3 (due to data dependency) because the annotation #(create:registeredMoney) is not followed by any #(use:registeredMoney) annotation.

(6) MOD2_UC8 (Fig. A11)—again, the tool identified that an open–close transaction needs to be properly closed in the looping extension 3a.
9. LIMITS OF VERIFICATION

Detection of non-primary use-cases:
As seen on the example of the use-case MOD1_UC3, the FOAM method can sometimes detect which use-cases should be non-primary. This is possible if the use-case contains a #create annotation which is not followed by any #use annotation. Such a use-case cannot be executed alone, yet it may be included from another use-case or used as a precedence.

Global variable names:
Variables that are used in temporal annotations have global names in the whole UCM. This may lead to problems with variable names in large specifications. However, naming conventions can help resolve the uniqueness problem easily.

Parallelism:
Currently, the FOAM tool does not support specification of parallelism. For example, it is not possible to specify that in some step, another process should be executed in parallel or at some step, the execution should wait until another process finishes execution.

Preconditions and postconditions:
In contrast to other approaches discussed in Section 10, FOAM does not support specification of pre/postconditions. However, some authors [16] argue, that pre/postconditions can cause problems to analysts.

Nested branches:
The number of levels for extensions and variations in FOAM is not explicitly limited. However, as the depth increases, it becomes harder to find the parent step for a branch. Some methodologies even prevent users from defining hierarchies deeper than 1 level.

Temporal dependencies between arbitrary use-cases
The way how construction of rUCM currently works in FOAM implies that use-cases have to be related by precedence/include relations in order to be together verified within the same rUCM. If FOAM detects that there is a use-case in the constructed rUCM that shares some annotation identifier with a use-case outside of the rUCM, an error is reported prior to the actual verification. Thus, in order to verify that certain temporal invariants hold when any sequencing of involved use-cases is considered, users would need to explicitly add additional precedences to UCM. We plan to further improve the construction of rUCM which will enhance the scalability of FOAM and also remove the aforementioned limitation.

10. RELATED WORK

10.1. Use-case templates
To structure textual use-cases, simplify their analysis and avoid ambiguities, various templates have been proposed in the past to structure textual use-cases. The comparison of several approaches has been nicely presented in the paper [17]. We extended the feature-matrix and compared FOAM to the other approaches in Table 4 which can be summarized as follows:

1. FOAM does not rely on use-case descriptions when capturing the behavior of a use-case. It is an optional field which is preserved during the transformation and may be a subject to further linguistic analysis.
2. Instead of preconditions and postconditions, FOAM utilizes an explicitly defined precedence relation among use-cases.
3. FOAM focuses on sequencing of actions without considering actors. The necessary semantics of actions is encoded by the use-case template and annotations attached to use-case steps.
4. Custom user-defined annotations can be specified in FOAM and attached to use-case steps. They are also preserved during all stages of the transformation. This way, it is possible to capture cross-references to high-level requirements, which is useful when external tools are involved in the verification process.

10.2. Extended UCMs
The Unified Modeling Language (UML) standard defines ‘include’, ‘extend’ and ‘generalize’ relations between use-cases. However, we can also find other relations in the literature that make use-case specification clearer. For example, Doug Rosenberg proposed in the book [25] the ‘precedes’ and ‘invoke’ relations in requirements specifications. The precedence was also adopted in the paper [26] and in the example specification [10]. In the papers [8, 9], the authors discuss control flow of common use-cases, variant use-cases, component use-cases, specialized use-cases, ordered use-cases and their relations such as the uses-relation, the extends-relation and the precedes-relation. In their terminology, our FOAM method can be characterized as using variant and ordered use-cases because we focus on verification of use-cases that are related using the ‘include’ and ‘precede’ relations and allow extension and variation of use-case scenarios.

10.3. Extracting dynamic structures from requirements
In the work of Kof [27], a method for deriving message sequence charts (MSCs from International Telecommunication Union similar to UML sequence diagrams) from textual scenarios is described. The process has been extended by the same author [28] with the aim to avoid sentences that should...
TABLE 4. Summary of supported features in various works based on use-case templates.

<table>
<thead>
<tr>
<th>Field</th>
<th>Use-case name</th>
<th>Description</th>
<th>Precondition</th>
<th>Postcondition</th>
<th>Basic flow</th>
<th>Alternative flow</th>
<th>Primary actor</th>
<th>Secondary actor</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Some</td>
<td></td>
<td></td>
<td>Drazan and</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Some</td>
<td></td>
<td></td>
<td>Mencl [24]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Some</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Some</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Some</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Some</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Some</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Some</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Some</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is an extended version of the matrix from the paper by Yue et al. [17].

In the paper by Yue et al. [17], the authors also tackled the issue of sentences in passive voice. He showed on a case study (Instrument Cluster8), how to treat compound sentences and passive voice. These studies aim at extracting the formal model of the behavior encoded in textual form, but not at all actual verification of its consistency. On the other hand, we decided to capture the behavior using FOAM annotations and rather focus on the verification aspect.

In the paper by Ambriola and Gervasi [30], an environment for analysis of natural language requirements is presented. The result of all the transformations is a set of models for documents, and the requirements-writing process. Also transformation to ER diagrams, UML class diagrams and Abstract State Machines specifying the behavior of modeled subsystems (agents) is provided. Similar approach is described in the paper [31] (called Metamorphosis) for transforming textual use-cases to interaction models (sequence diagrams). A specification is represented by three models describing the syntactic, semantic and conceptual aspects. The transformation is driven by predefined patterns applied to the models. Each sentence yields an ‘interaction fragment’. By combining these fragments a complete interaction diagram is constructed.

In the paper [32], a notation for representing use-case specification is presented. This model is more complicated than in FOAM. Similarly to FOAM, the notation describes use-cases, their scenarios with steps, branching, inclusion and sequencing constraints. However, it also models lifelines, messages, pre/postconditions, activities, actors and other elements that are commonly used in requirements specifications. The main focus of the paper [32] is to capture various use-case specification in a single model, while FOAM focuses on verification of control flow in use-cases.

10.4. Formal semantics for use-cases

In the paper [33], a formal semantics based on LTS is proposed for use-cases containing extensions and includes. The authors utilize LTSs to automatically detect livelocks. They also propose a method for verifying refinement of UCMs, namely checking their equivalence and deterministic reduction. A tool UCM Analyzer is mentioned which assists the developer in the automated verification. All of the checks are focused on global properties of UCMs. The same authors also wrote a number of papers about mapping use-cases into different formalisms—partially ordered sets (POSETs) [34], finite state machines [35] and LTS + POSETs [36]. As opposed to FOAM, the properties to be verified are predefined and branching scenarios are not considered.

A method for modeling sequencing constraints of use-cases was presented by Some [16]. In his approach, the textual-use-case notation is enhanced with the ‘follow’ and ‘enables’

8http://www.empress-itea.org/deliverables/D5.1_Appendix_B_v1.0_Public_Version.pdf.
keywords to define precedence relation and synchronization rules for dependent use-cases. The paper also provides a comparison with pre/postconditions, where the author argues that pre/postconditions are often too broad or restrictive and that modelers have difficulty finding them at the early stages of UCM. Our precedence relation $\text{Preceding: } u_1, \ldots, u_n$ maps to the Somé’s notation: $\text{Follows: } u_1 \text{ AND } \ldots \text{AND } u_n$. However, the notation $\text{Follows: } u_1 \text{ OR } \ldots \text{ OR } u_n$ in a use-case $u$, which represents a precedence relation where some of $u_1, \ldots, u_n$ have to finish prior to the execution of $u$, is not currently supported by FOAM.

In another paper by Somé [37], textual use-cases are formalized via reactive Petri nets, taking into account the include and extend UML relationships and sequencing constraints using pre/postconditions. The method assumes that use-case steps comply with a restricted English grammar. The approach does not allow expressing other relationships and constraints. This is similar to our FOAM method because we use an explicitly modeled precedence relation as a sequencing constraint. Instead of verifying petri nets, we verify LTSs using a model-checker (currently a symbolic model-checker).

Related to FOAM are also the methods that map use-cases into the UML activity or sequence diagrams [17, 38–41]. The main focus of these works is on model-driven development, code generation, traceability, safety verification, while automated verification of temporal constraints in not discussed. In order to verify temporal constraints within the generated diagrams, they would need an additional transformation to a model understood by a model-checker, which is generated. From the modeling point of view, these papers operate with the generalization, include and extend UML relations, which partially overlaps with the include relation and the notion of extensions and variations in FOAM.

There are also many approaches aiming at formalizing UML models in general. For instance, in [42] the authors propose an automated method for translating UML sequence diagrams into Petri nets for evaluating reliability of software architectures. Their method uses annotations in the form of stereotypes based on the UML profile for QoS and fault tolerance [43]. In contrast to FOAM, these approaches operate on UML diagrams rather than textual use-cases.

In the paper [44], a toolkit SPIDER for analysis of UML models is presented. The authors designed a method for verifying temporal properties in UML activity diagrams using the SPIN model-checker. The temporal properties are specified in natural language (English) which is mapped by the tool to a predefined set of patterns (as proposed by Dwyer et al. [45]) representing LTL properties to be verified. The tool aims at novice users who are not experts in temporal logic. If they follow the presented patterns, the tool can automatically derive and verify the properties. In a similar way, FOAM encapsulates LTL properties by more readable temporal annotations. Moreover, we do not have a predefined set of patterns. Temporal annotations can be defined by experts in temporal logic as needed.

Another approach for model-checking use-cases using SPIN is proposed in [46]. This method assumes that a UCM is expressed as: (1) UML use-case diagram containing relations among actors and use-cases, (2) use-case descriptions describing the main scenario and alternative flows, (3) pre/postconditions that specify how use-cases are sequenced. To formalize such a model, a notation based on first-order-logic predicates is introduced. Once the user specifies all the steps in use-cases and all the pre/postconditions formally, the specification is transformed into PROMELA code and verified in SPIN. The underlying formalism is thus exposed to the end-user, as opposed to FOAM. It is assumed that the user understands first-order logic and also temporal logic (LTL), whereas in FOAM, we aim at exposing to the end-user the specification in its original textual form. Further comparison with FOAM shows that the method [46] does not distinguish between extensions and variations (only extensions are supported) and that the include relation is not supported.

The authors of the paper [47] demonstrate an model-driven approach to development of autonomous ground control system for NASA directly from textual requirements, called Requirements-to-Design-to-Code (R2D2C) approach. Their method relies on a special English-like syntax of the input text defined using the ANTLR grammar. User requirements are first transformed to CSP (Hoare’s language of Communicating Sequential Processes) and then Java code is generated. The R2D2C is more suitable for modeling the behavior of autonomous agents while FOAM targets directly the specification and verification of textual use-cases.

In the papers [48, 49], the authors describe an object-oriented methodology based on so-called Task Models. The method uses the UML activity diagrams as a graphical notation and a process algebra as a formalism. The method provides means for model checking the traces obtained from the expressions in process algebra that are generated from the graphical representation. Temporal properties to be verified can be written as formulae in LTL or CTL, which may be difficult for people without a formal background. Users interact with the method only through the GUI which is different from FOAM, where we decided to focus on the textual notation. The method allows users to specify parallel execution (fork, join) and postconditions, which is not yet possible in FOAM. No information is available as to how the proposed method scales with the size of input specification.

11. CONCLUSION

In this article, we presented FOAM, a novel method for verifying correct sequencing of actions in use-cases. The main advantage of the method is its ease of use. In particular, it works with use-cases in their natural language form enhanced
by a few basic annotations. Thus, it can be easily integrated with existing development processes. Further, by allowing user-defined annotations, it can be customized for domain-specific properties to be verified.

We have developed a command-line-based tool that performs verification of UCMs instrumented with FOAM annotations. The architecture of our tool is modular and extensible. Each transformation phase is clearly separated and based on well-defined meta-models describing its inputs and outputs.

As to future work, there are two main directions of our research. First, currently the annotations have to be added manually during the preparation of a specification; nevertheless, we have already started work on a FOAM extension which, in an automated way, will propose addition of annotations using a machine-learning approach. This is a difficult linguistic task and a controlled natural language would help achieve good accuracy (a review of transformation approaches between requirements and models is described in the paper [2]).

Second, we aim at improving the user interface of FOAM to make it easily fitting into a software development process. The usual verification speed we encountered so far has been feasible. Most of the time is spent in Java virtual machine initialization, model-to-model transformations and serialization/deserialization of models. The actual NuSMV verification was capable of use in practice.

In future, we plan to evaluate the usability of FOAM on large specifications (100+ use-cases) and also to implement another model-checking back-ends. We also plan to evaluate the usability of FOAM on a larger group of testers. The NuSMV model-checker we are currently using, cannot utilize multiple CPU cores, so that SPIN-based implementation might benefit from SPIN’s multi-core model-checking algorithm.

FUNDING

This work was partially supported by the Grant Agency of the Czech Republic project P103/11/1489 and by the European Union within the FP7 Marie Curie Initial Training Network ‘RELATE’ under grant agreement number 264840.

REFERENCES


APPENDIX. DATA RELATED TO EVALUATION

MODULE main —— NuSMV code generated automatically from an annotated specification

—— The fairness constraint prevents infinite loops in the following states
FAIRNESS guardloop;
DEFINE guardloop := s in \{init.0, succ.0, init.1, init.2, init.3\};

VAR exec_1 : boolean; —— indicates that the UC1 has been executed
ASSIGN
Init(exec_1) := FALSE;
next(exec_1) := case
  s=succ.1 : TRUE;
  TRUE : exec_1;
  esac;

VAR exec_2 : boolean; ... —— indicates that the UC2 has been executed
VAR exec_3 : boolean; ... —— indicates that the UC3 has been executed

VAR s : \{init.0, init.1, init.2, init.3, u1.s0, u1.s1, ... u3.s2.b1.s2, succ.1, succ.2, succ.3, succ.0, succ.00\};
ASSIGN
init(s) := init.0;
next(s) := case
  s=init.0 : \{succ.0, init.1, init.2, init.3\};
  s=succ.0 & !\{exec.1 & exec.2 & exec.3\} : init.0;
  s=succ.0 : succ.0; —— we need this due to fairness constraint
  s=succ.00 : succ.00; —— finally, loop here forever
—– UC1
  s = init.1 & !\{exec.1\} : init.0; —— when UC1 is not scheduled
  s = init.1 : u1.s0;
  s = u1.s0 : u1.s1; ... s = u1.s5 : succ.1;
  s = succ.1 : Init.0; —— returning from UC1
—– UC2 ...
—– UC3 ...
  esac;

VAR v_create0 : boolean; —— temporal annotations create–use
ASSIGN
init(v_create0) := FALSE;
next(v_create0) := case
  s = u1.s0 : TRUE;
  s = u1.s1 : FALSE;
  TRUE : v_create0;
  esac;

VAR v_use0 : boolean;
ASSIGN
init(v_use0) := FALSE;
next(v_use0) := case
  s = u2.s3 : TRUE; —— u2 picked randomly
  s = u2.s4 : FALSE;
  TRUE : v_use0;
  esac;

LTLSPEC G(v_use0 -> O(v_create0)) —— comment truncated
CTLSPEC AG(v_create0 -> EF(v_use0)) —— comment truncated
CTLSPEC AG(v_create0 -> AX(AG(v_create0))) —— comment trunc.

FIGURE A1. An instance of NuSMV code in our experiment generated from three use-cases without precedence relation (u = 3, m = 5, bc = 2, bl = 3, a = 1: see Section 6).
MOD2 UC6: Transfer Candidates’ data

Summary: When the admission is finished, data of qualified candidates should be transferred to the Students management system.

Preconditions:
- Administrator is logged in to the system
  Preceding: “UC???...”

Trigger
- Admission end date has passed and Selection committee has qualified Candidates.
  Preceding: “UC???...”

Main Scenario:
1. Administrator chooses all qualified Candidates who provided paper version of their application form.
2. Administrator chooses the transfer option. #create:transferOption
3. System transfers Candidates data to the Students management system. #open:transfer
   - Extension: System is unable to transfer some of the accounts.
   - System informs that some accounts have not been transferred.
   - Administrator selects detailed information about the error.
   - System presents details concerning errors during transfer process.
   - Use case finishes. #close:transfer, #abort
   - Variation: Is a recoverable error.
     - System tries the transfer operation again in step 3. #goto:3
4. System authorizes in the Students management system. #include:UC???...
5. System sends Candidates data. #use:transferOption
6. Students management system verifies the data.
7. Students management system sends an acknowledge message to the System.
8. System displays information about a successful transfer. #close:transfer

"Flow Annotations"
- #(abort)
  - This annotation expresses abort of the scenario. It can only be added to the last step of a variation or an extension.

- #(goto:x)
  - This annotation represents a jump within the use-case.
  - The parameter x indicates the target use-case step of the jump.

- #(includeu)
  - This annotation specifies inclusion (inlining) of another use-case u.

"Temporal Annotations"
- #(open:x) ... #(close:x)
  - something like a transaction:
    - there open—close has to be properly paired
    - when x is opened, it has to be closed
    - x cannot be closed before being opened first

- #(create:x) ... #(use:x)
  - useful for marking data dependency:
    - x cannot be used before being created
    - x can be used multiple times
    - if x is created, there has to be at least one branch where x is used
    - (some branches can well be without it)

FIGURE A2. Annotations explained using a use-case example.
FIGURE A3. UC1: login to the system.

UC1: Login to the system

Level: User
Main Actors: User
Summary: In order to use system one has to authenticate.

Preconditions:
- User is not logged in.

Main Scenario:
1. User opens main page.
2. System presents main page with a login form.
3. User fills the login form with the authentication data.
4. System verifies the given data.
5. System welcomes Candidate.

Extension: 4a. Not all obligatory data was given.
4a1. System points which data is missing.
4a2. Go back to step 3. #(goto:3)

Extension: 4b. No account with the certain login exists in the system.
4b1. System informs the User that there is no account with the given user name in the system.
4b2. System suggests the User to register in the system as a Candidate or contact Administrator to create new account.
4b3. Go back to step 3. #(goto:3)

Extension: 4b2a. User decided to register as a Candidate.
4b2a1. Include MOD1_UC1. #(include:MOD1_UC1)

FIGURE A4. MOD1_UC1: register in the system.

MOD1_UC1: Register in the system

Level: User
Main Actors: Candidate
Summary: Candidate has to register in the system in order to apply for studies.

Preconditions:
- Candidate is not logged in

Main Scenario:
1. Candidate opens system main—page.
2. Candidate chooses registration option.
3. System presents a registration data form and asks to enter the registration data.
4. Candidate fills the registration data form and submits the registration data form. #(open:registration)
5. System verifies if data is correct.
6. System informs that account has been created. #(close:registration)

Extension: 5a. Some obligatory fields were not filled.
5a1. Systems highlights the missing fields. #(close:registration) ·········· FIXED
5a2. Back to step 4. #(goto:4)

Extension: 5b. Account with the given user name already exist.
5b1. System informs that the user name is in use. #(close:registration) ·········· FIXED
5b2. Back to step 4. #(goto:4)

Extension: 5c. Given passwords don’t match.
5c1. System informs Candidate that passwords don’t match. #(close:registration) ·········· FIXED
5c2. Back to step 4. #(goto:4)
**MOD1_UC2: Provide personal and education information**

Level: User  
Main Actors: Candidate  
Summary: Candidate has to provide personal information as well as some facts concerning his/her previous education.

Preconditions:
- Candidate is logged in to the system.  
  **Preceding:** "UC1: Login to the System"

Triggers:
- Either Candidate has logged to the system for the first time or has chosen to enter his/her application data.

Main Scenario:
1. Candidate provides personal information.  
2. Candidate chooses to provide information concerning former education.  
3. System presents the education data form.  
4. Candidate fills the education data form and confirms.  
5. System stores the data.  
6. System displays a confirmation message.

**Extension:** 4a. Some obligatory data was not provided.  
4a1. System informs that required some data was not provided and highlights the missing fields.  
4a2. Go back to step 2. **(goto:2)**

**Variation:** 4a2a Candidate logged to the system for the first time  
4a2a1. Go back to step 1. **(goto:1)**

**Extension:** 4b. Some data was provided in wrong format.  
4b1. System informs that some data was not provided correctly and highlights the fields that were consider as wrongly formatted.  
4b2. Go back to step 2. **(goto:2)**

**Variation:** 4b2a. Candidate logged to the system for the first time  
4b2a1. Go back to step 1. **(goto:1)**

**FIGURE A5.** MOD1_UC2: provide personal and education information.

**MOD1_UC3: Choose a major**

Primary: FALSE .......... **FIXED**  
Level: User  
Main Actors: Candidate  
Summary: Candidate would like to choose one or more majors he/she would like to apply for.

Preconditions:
- Candidate is logged in to the system  
  **Preceding:** "UC1: Login to the System"  
- Candidate provided personal and education information  
  **Preceding:** "MOD1_UC2: Provide personal and education information"

Main Scenario:
1. Candidate chooses the adding—new—major option.  
2. System presents a list of majors for which admission is available.  
3. Candidate chooses a major. **(#create:chosenMajor)**  
4. System presents a list of majors chosen by Candidate.

**Extension:** 3a. Candidate would like to apply for more majors.  
3a1. Candidate chooses many majors.  
3a2. Continue with step 4. **(goto:4)**

**FIGURE A6.** MOD1_UC3: choose a major.
MOD1_UC4: Assign an application fee to a major

Level: User
Main Actors: Candidate
Summary: Candidate has to pay an application fee for each major he/she chooses.

Preconditions:
– Candidate is logged in to the system
  Preceding: “UC1. Login to the System”
– Candidate has chosen at least one major
  Preceding: “MOD1_UC3. Choose a major”

Main Scenario:
1. Candidate proceeds to the chosen—majors view.
2. System presents list containing chosen majors.
3. Candidate chooses a major that he/she wants to pay for. #(use:chosenMajor)
4. System presents a payment form and asks about the method of payment. #(open:payment)
5. Candidate chooses to use a credit card.
6. Candidate provides credit card data and confirms payment. #(close:payment)
7. System presents updated list of the chosen majors.

Variation: 5a. Candidate chooses to pay by money transfer.
5a1. System presents Candidate’s individual account number.
5a2. Candidate performs money transfer (outside the system).
5a3. Money is registered by the System (MOD2 UC12) #(include:MOD2_UC12)
5a4. After money is registered candidate assigns the payment to a major. #(use:registeredMoney)
5a5. Use cases finishes. #(abort), #(close:payment) ------- FIXED

Extension: 5a3a. Error occurred while registering. #(guard:create:registeredMoney) ------- FIXED
5a3a1. Transaction terminated. #(abort), #(close:payment) ------- FIXED

Variation: 5a4a. If he/she don’t do that
5a4a1. the payment will be assigned automatically according to priorities.

FIGURE A7. MOD1_UC4: assign an application fee to a major.

MOD1_UC5: Check application status

Level: User
Main Actors: Candidate
Summary:
System should provide information about current status of the Candidate’s application.
For example Candidate can check whether his/her application is being processed or was accepted by Selection committee.

Preconditions:
– Candidate is logged in to the system
  Preceding: “UC1. Login to the System”
– Candidate has chosen at least one major
  Preceding: “MOD1_UC3. Choose a major”

Main Scenario:
1. Candidate chooses an option of presenting current status of his applications.
2. System presents a chosen—major list with information concerning current status of each application.
   #(use:chosenMajor)

FIGURE A8. MOD1_UC5: check application status.
MOD2_UC1: Create a new admission

Level: User
Main Actors: Administrator
Summary: Administrator has to create a new admission in the system and configure it before the whole process can start.

Preconditions:
- Administrator is logged in to the system
  
  Preceding: “UC1. Login to the System”

Main Scenario:
1. Administrator chooses the creating—a new—admission option.
2. System presents the new—admission form. #(open:admissionForm)
3. Administrator provides basic information concerning the admission.
4. Administrator chooses starting and ending dates of the admission.
5. System stores the admission. #(close:admissionForm)
6. System informs that admission has been stored.

Extension: 3a. Some obligatory data not provided. #(guard:visited(3a)) .......... FIXED
3a1 System informs that some required data is missing. #(close:admissionForm) .......... FIXED
3a2 Go back to step 3. #(goto:3)

Extension: 4a. The given dates are not valid. #(guard:visited(4a)) .......... FIXED
4a1 System informs that starting or ending dates are not valid. #(close:admissionForm) .......... FIXED
4a2 Go back to step 4. #(goto:4)

Extension: 3b. Catching an infinite loop. #(guard:visited(3a)) .......... FIXED
3b1. Use case aborted. #(abort) .......... FIXED

Variation: 4b. Catching an infinite loop. #(guard:visited(4a)) .......... FIXED
4b1. Use case aborted. #(abort) .......... FIXED

FIGURE A9. MOD2_UC1: create a new admission.

MOD2_UC12: Import admissions fees

Primary: FALSE .......... FIXED
Level: User
Main Actors: Administrator
Summary: Information about payments which are done via money transfer procedure (outside the system) have to be imported from the Bank system.

Preconditions:
- Administrator is logged in to the system.
  
  Preceding: “UC1. Login to the System”

Main Scenario:
1. Administrator chooses an option to import payments from the bank system.
2. System imports payment entries from the bank.
3. System displays a list containing information about all imported admission fees.
   
   #(create:registeredMoney)

Extension: 2a. Error occurred during the import.
2a1. System displays error message with the detailed information concerning the source of the failure.
2a2. Use case finishes. #(abort)

FIGURE A10. MOD2_UC12: import admissions fees.
**MOD2.UC8: Add a new user**

**Level**: User

**Main Actors**: Administrator

**Summary**: Administrator can add users.

**Preconditions**:
- Administrator is logged in to the system.

  **Preceding**: "UC1. Login to the System"

**Main Scenario**:
1. Administrator chooses an option to add user.
2. System presents the new—user form.
3. Administrator fills the form. *(open:newUserForm)*
4. Administrator grants roles to the user in the system.
5. System stores the user data. *(close:newUserForm)*
6. System grants the user roles.
7. System displays confirmation message.

**Extension**: 3a. Not all obligatory data was given.
   3a1. System informs that some data is missing. *(close:newUserForm)* ———— FIXED
   3a2. System highlights the missing fields.
   3a3. Go back to step 3. *(goto:3)*

**FIGURE A11.** MOD2.UC8: add a new user.

**FIGURE A12.** Precedence and include relations of the UCM in our evaluation.
FIGURE A13. An example specification with a precedence relation. This example is taken from global personal marketplace, system requirements Specification [10].