Department of Distributed and Dependable Systems
Technical report no. D3S-TR-2012-02
August 2, 2012

Formal Verification of Annotated Use-Cases

Viliam Simko, Petr Hnetynka, Tomas Bures, Frantisek Plasil

Abstract: Textual use-cases have been traditionally used at the design stage of development process for describing software functionality from the user's view. Their advantage is that they can be easily understood by stakeholders and domain experts. However, since use-cases typically rely on a natural language, they cannot be directly subject to a formal verification. In this paper, we present **F**ormal Verification **o**f **A**nnotated Use-Case **M**odels (FOAM). 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 verifies temporal invariants associated with the annotations. This way, FOAM allows for harnessing the benefits of model-checking while still keeping the use-cases understandable for non-experts.

This work was partially supported by the Grant Agency of the Czech Republic project P103/11/1489 and by the Charles University institutional funding SVV-2012-265312.

Contents

1 Introduction

Specification of functional requirements using textual use-cases [7] is a well established technique in requirements engineering. The use of a 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 (actors). However, the natural language imposes the risk of ambiguity or contradiction in specification documents which can negatively impact later phases of the system development. With the increasing complexity of a use-case specification it becomes hard to ensure its validity. Additionally, in a changing environment, the original specification can get out-of-sync with the implementation artefacts. Thus a formalisation and automated validation of use-cases is desirable.

One of the few properties that can be checked in an automatized way is the correct sequencing of actions. In [13] we have proposed a method for verifying temporal constraints among use-case steps. The verification is based on semantic *annotations* attached to the use-case steps. This allows expressing temporal invariants in a way that is understandable to both domain engineers and stakeholders.

The method described in [13] works with predefined annotations. However, different application domains require a broader spectrum of properties to be verified. In this paper, we therefore introduce a general framework for specification of temporal constraints expressed as annotations based on temporal logics. We also show how these constraints can be verified in an automated way. In particular, we formalize our approach and show transformation of a set of annotated use-cases to an LTS with guards, which can be treated as a standard input formalism used by contemporary model-checkers. We provide a discussion on our experience with utilizing the NuSMV symbolic model checker for this purpose.

The remainder of this paper is structured as follows: In Section 2 we shortly overview the structure of textual use-cases while in Section 3 we introduce the annotations which are subject to formal verification. Section 4 is the core part of the paper which describes in detail the transformation of a use-case model into Symbolic Model Verifier (SMV) language through a set of Labeled Transition System (LTS) automata. The correspondence between a use-case model and SMV code is formally described by inference rules.

2 Textual use-cases

The main advantage of use-cases – the use of a natural language to achieve easy comprehension – makes it also a hurdle from the perspective of automated processing. This is not just because of the intricacies interpreting the text in the natural language, but also because, to date, there is no standardized form of use-cases. To circumvent this, we adhere to the widely accepted format proposed in [7]. To facilitate reading of the paper, we briefly summarize this use-case format below.

Typically the system under discussion 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 *use-case 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. 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 of **UseCase 1** in Figure 1). A branching scenario is either *variation* or *extension* of a particular use-case step. An extension enhances specification of the particular step while a variation is an alternative to the step's specification. The correspondence of a variation or an extension to a step is given by referring to the step's sequence number (e.g. variation $2a$ is an alternative to step 2 in the **UseCase 1**).

Use-cases can also be involved in a *precedence relation* [4, 3, 9], which constraints their sequencing (e.g. before the use-cases 2 or 3 can be executed, the use-case 1 has to be executed first).

3 Verification of use-cases with annotations

As mentioned in the introduction, it is hard to ensure correctness of large and complex specifications just by reviewing. Moreover, it would be a mistake to understand requirements documents as final and unchangeable (as emphasized in [10]), so that reviewing is typically a time-consuming process.

FOAM allows for an automated verification of use-cases' correctness. It comprises several steps shown in Figure 2. First, the use-cases are instrumented by annotations. An annotation is a tag appended to a particular use-case step.

UseCase 1: Select city on map
1. The user opens the map web page.
2. The system generates a map with available cities.
3. The user selects a city on the map. ⟨create:city⟩
Variation: 2a. No cities available.
2a1. System displays an empty map with message.
2a2. Use−case aborts. ⟨abort⟩

UseCase 2: Generate city
Preceding: "Select city on map"
1. The system asks MapServer to provide city information. ⟨use:city⟩
2. MapServer provides the requested information.
3. The system generates the map with default zoom settings. ⟨create:zoom⟩
4. User adjusts zoom settings. ⟨use:zoom⟩
Extension: 2a. City already generated
2a1. Use−case aborts. ⟨guard:create:zoom⟩ ⟨abort⟩
Variation: 2b. MapServer error occurred.
2b1. Use−case aborts. ⟨abort⟩

UseCase 3: Generate restaurant map for city
Preceding: "Select city on map"
1. Include use−case "Generate city". ⟨include:GenerateCity⟩
2. System validates the zoom settings. ⟨use:zoom⟩
3. System asks RestaurantServer for restaurants. ⟨use:zoom⟩ ⟨use:city⟩
4. RestaurantServer generates the restaurant layer information.
5. System generates restaurant map.
Variation: 1a. There was an abort in "Select city on map".
1a1. Use−case aborts. ⟨guard:abort⟩ ⟨abort⟩
Extension: 1b. There was an abort in "Generate city".
1b1. Use−case aborts. ⟨guard:abort⟩ ⟨abort⟩
Extension: 2a. Zoom settings are invalid
2a1. System display an error message to the user.
2a2. Goto step 1. ⟨goto:1⟩

The elements denoted as $\langle a{:}s \rangle$ are examples of annotations in FOAM.
– "a" is the name of the annotation
– "s" is the qualifier of the annotation

Figure 1: *Example of a use-case model with 3 annotated use-cases.*

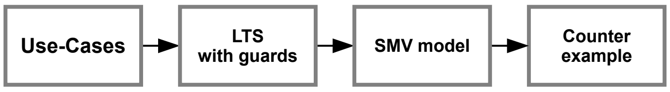


Figure 2: *Overview of the verification method*

Next, the annotated use-cases are programmatically transformed into an LTS. This model is passed to the NuSMV model checker for verification. The transformations are transparent to the user; the potential errors reported by NuSMV are presented in a natural language by translating the counter-example to the steps of the flawed use-case.

In FOAM, we define two sorts of annotations:

- *flow annotations* expressing control flow of use-cases, and
- *temporal annotation* expressing temporal invariants to be satisfied by use-cases.

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: (i) *aborts* which prematurely end the scenario – typically as a reaction to an error; (ii) *includes* which incorporate (inline) another use-case in the place of a particular step, (iii) *jumps* which move execution to a specified use-case step, and (iv) *conditions* of extensions and variations.

All these constructs are written in a 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 abort of the scenario.

⟨goto:s⟩ : This annotation represents a jump within the use-case. The parameter s indicates the target use-case step of the jump.

⟨include:u⟩ : This annotation specifies inclusion (inlining) of another use-case u.

The following annotations ⟨mark⟩ and ⟨guard⟩ assume existence of boolean variables $b_1, \ldots, b_n$ initialized to $false$ (globally accessible in UCM).

⟨mark:b_i⟩ : This annotation sets b_i to $true$.

⟨guard:$f(b_1, \ldots, b_k)$⟩ : The f parameter of this annotation is a propositional logic formula over the boolean variables $b_1, \ldots, b_k$. The annotation serves as a guard for extensions and variations.

3.2 Temporal annotations

Temporal annotations allow expressing temporal invariants among use-case steps in the whole Use-Case Model (UCM) without requiring an in-depth knowledge of the underlying temporal logic (CTL and/or LTL). This is possible because FOAM distinguishes two types of users:

(a) experts in temporal logic who prepare templates of annotations in Temporal Annotation Definition Language (TADL) in the form illustrated by Figure 3,

(b) domain engineers who refer to the names of these templates when associating use-case steps with annotations (Figure 1). For this activity detailed knowledge of 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.

We define these annotation in a domain specific TADL. Its syntax is straightforward as illustrated in Figure 3.

TADL defines a group of related temporal annotations along with their semantics expressed as a set of temporal logic formulae (typically Computational Tree Logic (CTL) and Linear Temporal Logic (LTL)). The particular logic depends on the model-checker used. In the case of our implementation, which relies on the NuSMV, we support both the logics. Temporal constraint expressed by the formula is also written down in a human-readable form for error reporting (when showing a counter-example to the user).

Let us now examine the example in Figure 3 in more detail. There are three temporal annotation groups defined here. The "create,use" annotations allow expressing constraints on ordering the use-case steps. For instance, in Figure 1, the step 1 of the use-case 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:x⟩ annotations having the same parameter x. The conversion of these annotations would result in the following set of formulae:

CTL AG(create$_{city}$ −> **EF**(use$_{city}$))
CTL AG(create$_{city}$ −> **AX**(**AG**(! create$_{city}$)))
CTL A[! use$_{city}$ **U** create$_{city}$ | ! **EF**(use$_{city}$)]

CTL AG(create$_{map}$ −> **EF**(use$_{map}$))
CTL AG(create$_{map}$ −> **AX**(**AG**(! create$_{map}$)))
CTL A[! use$_{map}$ **U** create$_{map}$ | ! **EF**(use$_{map}$)]

The "open,close" tuple is similar but there should be only one appearance of the *close* in a single execution.

The template "init,process,release" in Figure 3 illustrates how more complex annotations can be defined in TADL (strict ordering of 3 phases in this case).

Note that in FOAM we assume formulae without X (next) operator (i.e. $\mathrm{LTL}_{\text{-X}}$, $\mathrm{CTL}_{\text{-X}}$). This is to provide for semantics that does not impose a specific level of granularity of use-case steps. From the practical point of view, this restriction does not severely limit the expressibility. Rather it makes model-checking more efficient and also in our case it simplifies checking of the use-cases.

```
Annotations: create, use
 CTL AG( create −> EF(use) ) "Branch with use required after create"
 CTL AG( create −> AX(AG(!create)) ) "Only one create"
 CTL A[!use U create | !EF(use)] "First create then use"

Annotations: open, close
 LTL G(open −> F(close)) "After open, close is required"
 CTL AG(open −> AX(A[!open U close])) "No multi−open"
 CTL AG(close −> AX(A[!close U open | !EF(close)])) "No multi−close"
 CTL A[!close U open | !EF(close)] "First open then close"

Annotations: init, process, release
          →
 CTL A[!process U init | !EF(process)] "First init then process"
 CTL AG(init −> AF(process)) "After init there should always be process"
 CTL AG(init −> AX(A[!init U process])) "No multi−init without process"
 CTL AG(process −> AX(A[!process U init | !EF(process)]))
     "No multi−process without init"
               →
 CTL A[!release U process | !EF(release)] "First process then release"
 CTL AG(process −> AF(release)) "After process, release is required"
 CTL AG(process −> AX(A[!process U release]))
     "No multi−process without release"
 CTL AG(release −> AX(A[!release U process | !EF(release)]))
     "No multi−release without process"
```

Figure 3: *Examples of custom annotations (templates) defined in TADL. (The concrete syntax of TADL expressed in Xtext notation is depicted in Figure 9).*

4 From specification to verification

In this section we formally define a use-case with annotations and show how the annotations can be verified.

Our approach is depicted in Figure 4. The input to FOAM is a collection of annotated textual use-cases called UCM. Each Annotated Textual Use-Case (ATUC) is specified as a set of steps, variations and extensions. Additionally, UCM specifies precedence constraints among ATUCs in the model. Based on UCM, we build a non-deterministic automaton – called Overall Behavior Automaton (OBA) – representing the overall behavior. OBA 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 4.6). The verification of OBA is performed with respect to temporal logic formulae coming from the definition of temporal annotations.

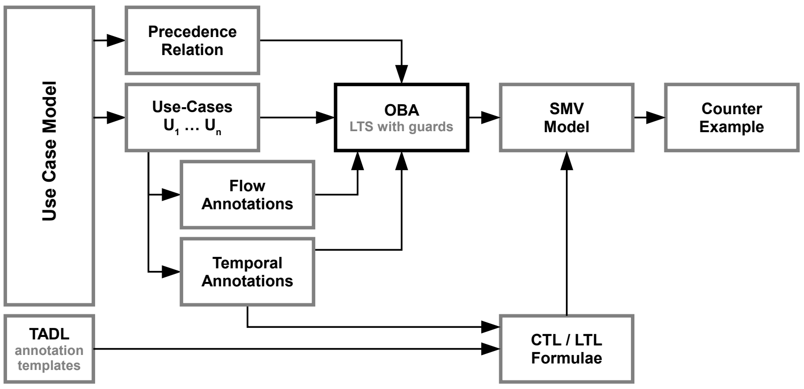


Figure 4: *Verification method in detail*

4.1 Formalizing the Input Use-Case Model

We start the formalization with definition of an ATUC. This structure represents a use-case as close as possible to the way it is usually written down (e.g. as in Figure 1). This means that we explicitly capture use-case steps (along with annotations attached to them), extensions and variations.

Definition 4.1 (Annotated textual use-case)**.** *An Annotated Textual Use-Case (ATUC) is a tuple:*

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

where:

- S_u *is a set of all steps (sentences written in English);*
- $W_u = \{w | w \subseteq S_u\}$ *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.* $\forall_{w,w' \in W_u}(w' \neq w) \Rightarrow (w \cap w' = \emptyset)$*.*
- $w_u^m \in W_u$ *is the main scenario;*
- $Ext_u : W_u \mapsto S_u$ *is a mapping function which assigns extensions to steps, i.e.* $w' \in W_u$ *is an extension of* $w \in W_u$ *from step* $s \in w$ *if* $Ext_u(w') = s$*;*
- $Var_u : W_u \mapsto S_u$ *is a mapping function which assigns variations to steps, i.e.* $w' \in W_u$ *is a variation of* $w \in W_u$ *from step* $s \in w$ *if* $Var_u(w') = s$*;*
- $Flow_u : S_u \mapsto 2^{\mathbb{F}}$ *is a function that assign a set of flow annotations to each step (*$\mathbb{F}$ *denotes a set of all flow annotations);*
- $Temp_u : S_u \mapsto 2^{\mathbb{T}}$ *is a function that assign a set of temporal annotations to each step (*$\mathbb{T}$ *denotes a set of all temporal annotations).*

Further, we say that an ATUC is **well-formed** if the following structural constraints below are not violated. These rules follow the common practice of writing use-cases 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.
2. The annotation ⟨guard⟩ is attached only to the first step of an extension or variation.
3. Main scenario of primary use-cases (Def.4.2) does not contain any ⟨goto⟩, ⟨abort⟩ or ⟨guard⟩ annotations.

Now, we define UCM as a collection of ATUCs accompanied with a precedence relation over the primary use-cases. UCM thus represents the textually specified overall behavior of a system. By a primary use-case we mean a use-case not included to any other use-case.

Definition 4.2 (Use-Case Model)**.** *A Use-Case Model (UCM) is a tuple:*

$$M = (U_M, U_M^p, Prec_M)$$

where:

- U_M *is a set of ATUCs;*
- $U_M^p \subseteq U_M$ *is a set of primary use-cases;*
- $Prec_M : U_M^p \times U_M^p$ *is a precedence relation on primary use-cases.*

In the rest of the paper, we assume only UCMs with well-formed ATUCs.

4.2 Formalizing the Overall Behavior Automaton

In FOAM, we transform UCM into OBA, which has well-defined semantics and can be rather directly used as an input to standard model-checkers. OBA is defined as follows:

Definition 4.3 (Overall Behavior Automaton)**.** *An Overall Behavior Automaton (OBA) is a tuple:*

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

where:

- *V is a set of states.*
- *$init_0 \in V$ is the initial state.*
- *$\tau \subseteq V \times V$ are transitions.*
- *B is a set of boolean variables.*
- *AP is a set of atomic propositions.*
- *$Val : \tau \mapsto 2^{(B \times \{true, false\})}$ are actions (valuations) on transitions which assign values to boolean variables in B.*
- *$Lab : V \mapsto 2^{AP}$ is labelling of states by temporal properties.*
- *$Guards : \tau \mapsto 2^{\mathcal{L}}$ are guards on transitions (a guard $g \in \mathcal{L}$ is a propositional logic formulae with variables from B).*

The semantics of OBA is the following:

- the execution starts in state $init_0$,
- the transition to another state is by non-deterministic choice among outgoing transitions, whose all guards are satisfied,
- upon the transition, the boolean variables of the automaton are updated based on the actions associated with the transition,
- for the sake of model-checking, the function Lab gives the atomic propositions that hold in a particular state.

4.3 Building Overall Behavior Automaton – step #1

Having provided the definition of UCM and OBA, we now show OBA construction from a UCM. This process is performed in two steps. In the first step below, we describe the automaton with the help of inference rules (in the form $\frac{premise}{conclusion}$). The rules put logical constraints on OBA based on the input UCM. In other words, the inference rules provide a logical theory, the model of which is OBA. In FOAM, we take the minimal model (with respect to inclusion) as the resulting OBA.

The basic OBA structure constructed from use-cases $U_1, \ldots, U_n$ is depicted in Figure 5. There is an initial state $init_0$ with branches to particular sub-automatons, each corresponding to one of the use-cases $U_1, \ldots, U_n$. The transitions to the sub-automatons are guarded by formulae that reflect the precedence constraints. This way, OBA captures the non-determinism in sequencing the use-cases. After the sequence is completed, OBA 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 $s \to s'$ to denote the existence of states s and s' and the existence of transition between them, i.e. $s, s' \in V \wedge (s, s') \in \tau$. Additionally we use the notation $s \xrightarrow{[G]} s'$ to additionally state that the $G \in 2^{\mathcal{L}}$ 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 subset of actions on t, i.e. that $V \subseteq Val(t)$.

The rules are the following (each accompanied with brief explanation):

(Rule 1) Representing steps: Every use-case step x is represented in the automaton A as a fixed number of states connected with transitions:

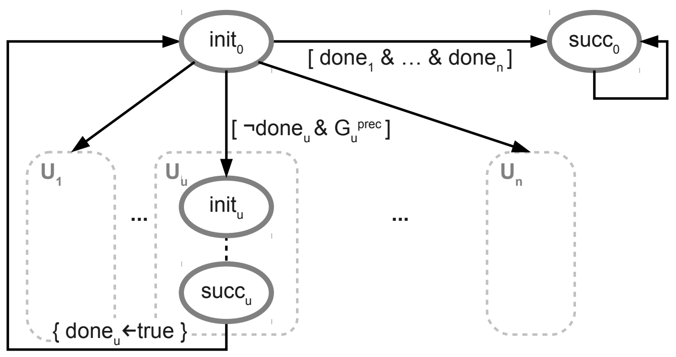


Figure 5: *OBA constructed from use-cases $U_1, \dots, U_n$*

- x^{in} represents the state before x has been executed.
- x^{var} is the source state of all variations attached to x.
- x^{jump} is the target state of a ⟨goto:x⟩ annotation.
- x^{ext} is the source state of all extensions attached to x.
- x^{out} represents the state after x and all its branching scenarios have been executed. Therefore it is the target state when continuing the execution from extensions and variations.

$$\frac{u \in U_M, x \in S_u}{x^{\mathsf{in}} \to x^{\mathsf{var}} \to x^{\mathsf{jump}} \to x^{\mathsf{ext}} \to x^{\mathsf{out}}}$$

(Rule 2) Representing scenarios: Let $w \in W_u$ be a scenario containing steps $x_1 \leq_w \dots \leq_w x_n$ linearly ordered using the total order $\leq_w \in Ord_u$. Then in A we connect the individual steps according to the order imposed by the $\leq_w$ relation.

$$\frac{u \in U_M, w \in W_u, x_1 \leq_w \dots \leq_w x_n}{(x_1^{\mathsf{out}} \to x_2^{\mathsf{in}}), \dots, (x_{n-1}^{\mathsf{out}} \to x_n^{\mathsf{in}})}$$

(Rule 3) Connecting variations to parents: Let $w \in W_u$ be a variation from step x which contains steps $y_1, \dots, y_n$. Then we connect w (state y_1^{in}) to its parent (state x^{var}). If the variation is conditioned by a ⟨guard⟩ annotation, we add this as a guard.

$$\frac{\begin{gathered} u \in U_M, w \in W_u, w = \{y_1, \dots, y_n\}, Var_u(w) = x, \\ G_V = \{g \mid \langle\text{guard:}g\rangle \in Flow_u(y_1)\} \end{gathered}}{x^{\mathsf{var}} \xrightarrow{[G_V]} y_1^{\mathsf{in}}}$$

(Rule 4) Connecting extensions to parents: Let $w \in W_u$ be an extension from step x which contains steps $y_1, \dots, y_n$. Then we connect w (state y_1^{in}) to its parent (state x^{ext}). If the extension is conditioned by a ⟨guard⟩ annotation, we add this as a guard.

$$\frac{\begin{gathered} u \in U_M, w \in W_u, w = \{y_1, \dots, y_n\}, Ext_u(w) = x, \\ G_E = \{g \mid \langle\text{guard:}g\rangle \in Flow_u(y_1)\} \end{gathered}}{x^{\mathsf{ext}} \xrightarrow{[G_E]} y_1^{\mathsf{in}}}$$

(Rule 5) Continuation from scenarios: Let w be a branching scenario (variation/extension) from step x which continues the execution in its parent scenario. Then we connect y_n (state y_n^{out}), the last step of w, to x (state x^{out}).

$$\frac{\begin{gathered} u \in U_M, w \in W_u, x = Var_u(w) \vee x = Ext_u(w), \\ w = \{y_1, \dots, y_n\}, \langle\text{abort}\rangle \notin Flow_u(y_n), \\ \forall_{s \in S_u} \langle\text{goto:}s\rangle \notin Flow_u(y_n) \end{gathered}}{y_n^{\mathsf{out}} \to x^{\mathsf{out}}}$$

(Rule 6) Handling goto annotations: Let x be a use-case step annotated with $\langle\text{goto:}y\rangle$, where y is another step in the same use-case. We handle the annotation by jumping to the y^{jump} location. This means that variations of the step y will be skipped, however extensions are still executed.

$$\frac{u \in U_M, x \in S_u, \langle\text{goto:}y\rangle \in Flow_u(x)}{x^{\text{out}} \rightarrow y^{\text{jump}}}$$

(Rule 7) Handling abort annotations: For each state annotated by $\langle\text{abort}\rangle$, we add a transition which introduces an infinite loop.

$$\frac{u \in U_M, x \in S_u, \langle\text{abort}\rangle \in Flow_u(x)}{x^{\text{out}} \rightarrow x^{\text{out}}}$$

(Rule 8) Resolution of includes (calling a procedure): The include relationship among use-cases is expressed using $\langle\text{include}\rangle$ annotations. In order to implement include operations within OBA, we add a set of boolean variables $incl_{x,c}$, where $incl_{x,c} = true$ if a use-case c has been called directly from a step x of a use-case u.

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

$$\frac{\begin{array}{c} u, c \in U_M, x \in S_u, \langle\text{include:}c\rangle \in Flow_u(x), \\ w_c^m = \{y_1, \ldots, y_n\} \end{array}}{x^{\text{jump}} \xrightarrow{\{incl_{x,c} \leftarrow true\}} y_1^{\text{in}}, x^{\text{jump}} \xrightarrow{[false]} x^{\text{ext}}}$$

(Rule 9) Resolution of includes (return): The last step of the included use-case is connected back to the calling use-case, assuming that the included use-case does not end by looping.

$$\frac{\begin{array}{c} u, c \in U_M, x \in S_u, \langle\text{include:}c\rangle \in Flow_u(x), \\ w_c^m = \{y_1, \ldots, y_n\}, \forall_{s \in S_c} \langle\text{goto:}s\rangle \notin Flow_c(y_n) \end{array}}{y_n^{\text{out}} \xrightarrow{\{incl_{x,c} \leftarrow false\}} x^{\text{ext}}}$$

(Rule 10) Scheduling of use-cases: The execution of use-cases may be arbitrarily sequenced with respect to the precedence relation $Prec_M$. In this rule, we create a mechanism that non-deterministically executes each use-case exactly once, while obeying the precedence relation. To do so, we introduce boolean variables $done_u$, where $done_u = true$ if a primary use-case $u \in U_M$ has been completed. Furthermore, we introduce a global initial state $init_0$ with a transition to the initial state and from the final state of each primary use-case. Each transition to the initial state is guarded by a predicate over $done_u$ variables, which reflects the precedence relation. Each transition from the final state sets the corresponding $done_u$ variable to $true$.

$$\frac{\begin{array}{c} u \in U_M^P, w_u^m = \{x_1, \ldots, x_n\}, \\ G_u^{\text{prec}} = \{done_v | \exists_{v \in U_M^P} (v, u) \in Prec_M\} \end{array}}{init_0 \xrightarrow{[G_u^{\text{prec}}, \neg done_u]} x_1^{\text{in}}, x_n^{\text{out}} \xrightarrow{\{done_u \leftarrow true\}} init_0}$$

(Rule 11) Final state: After the sequence of all primary use-cases has been executed successfully (indicated by $\forall_{u \in U_M^P} done_u = true$), OBA ends up in its final state $succ_0$ in an infinite cycle.

$$\frac{G = \{done_u | u \in U_M^P\}}{init_0 \xrightarrow{[G]} succ_0 \rightarrow succ_0}$$

(Rule 12) Atomic propositions: Temporal annotations are translated to OBA as atomic propositions attached to a corresponding x^{jump} state. Note that x^{jump} is a state that is always visited when a step in the use-case is taken (it is circumvented only when variation is used instead of the default step).

$$\frac{x \in S_u, u \in U_M}{Lab(x^{\text{jump}}) = Temp_u(x)}$$

4.4 Building Overall Behavior Automaton – step #2

In this step, we address a kind of peculiarity in semantics of guards on variations and extensions. The typical interpretation, which we also stick to in our paper, is the following:

(i) A non-deterministic choice is assumed among the default step and its unguarded branches (i.e. variations and extensions without guards).

(ii) A non-deterministic choice is assumed among guarded branches with non-disjunctive guards.

(iii) Mutual exclusivity is assumed among the default step and unguarded branches on one hand and the guarded branches on the other hand.

The OBA constructed in step #1 follows the semantics in terms of (i) and (ii), but not of (iii). To address (iii), we need to additionally introduce the guards for the default step and the unguarded variations and extensions so as the mutual exclusivity holds. This is done in the following way:

For each step x, we compute the union of guarding formulae on variations as:

$$G_V^x = \bigcup_{\forall (x^{\mathsf{var}}, y) \in \tau} \bigwedge Guards((x^{\mathsf{var}}, y))$$

If G_V^x is a non-empty set, we add to each unguarded transition from x^{var} a guard computed as:

$$\neg \bigwedge G_V^x$$

We apply a similar addition for unguarded extensions:

$$G_E^x = \bigcup_{\forall (x^{\mathsf{ext}}, y) \in \tau} \bigwedge Guards((x^{\mathsf{ext}}, y))$$

if $G_E^x \neq \emptyset$, add to each unguarded transition from x^{ext} a guard:

$$\neg \bigwedge G_E^x$$

4.5 Temporal properties

Now we show a construction of temporal logic formulae based on the UCM and TADL (user-defined temporal annotations). Each temporal annotation used in UCM has the form $\langle a{:}s \rangle$, 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 $F_{\langle a:s \rangle} = \bigcup F_i^{tadl}[\_/\langle a{:}s \rangle]$, where F_i^{tadl} is the i-th logical formula defined in the template $tadl$ and where $[\_/\langle \_{:}s \rangle]$ denotes renaming of each variable (represented by placeholder _) in the formula to the form "$\_{:}s$". The temporal properties to be verified by the model-checker are obtained as union over all the sets $F_{\langle a:s \rangle}$ contributed by annotations used in UCM.

4.6 Verification using NuSMV

We have implemented a verification of OBA using the NuSMV model checker [5] (as OBA is defined as an LTS structure, it should be easy to employ any other state-of-the-art model checker for this task). NuSMV supports analysis of synchronous and asynchronous systems using CTL and LTL, thus we allow for both in defining temporal annotations.

Transformation of OBA into the NuSMV input language is straightforward (Figure 6). There is a NuSMV variable $state$, which corresponds to the current state. Transitions of OBA are reflected as NuSMV rules setting the $state$ variable based on the source state and guarding formulae. Each atomic proposition attached to a state is reflected by introduction of a boolean variable which is set to true upon entering the state and set to false upon leaving the state.

The only difficulty stems from the fact that NuSMV does not support non-deterministic choice between rules (the non-deterministic choice has to be done by assigning a random value to a variable). Thus we model each non-deterministic choice among guarded transitions by two steps: (i) target state is non-deterministically chosen among all target states (regardless 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. In order to avoid infinite loops, fairness is enforced in step (i) using a dedicated FAIRNESS condition featured by NuSMV.

```
MODULE main
    VAR state : {$s_1$, ..., $s_n$} −− all states of OBA
    ASSIGN init(state) := init_0; −− initial state of OBA
        next(state) := case
            state=$x$ : {$y_1$, ..., $y_n$}; −− transitions $x \to y_1$, ..., $x \to y_n$
            state=$y_i$ & !($g$) : $x$; ... −− guarded transition $x \xrightarrow{g} y_i$
        esac;
    FAIRNESS ! guardloop −− avoids infinite loops when testing guards
    DEFINE guardloop := state in {$x_1$, ..., $x_m$} −− states in guards
    VAR $v$ : boolean; −− variable $v$ from OBA
    ASSIGN init($v$) := FALSE; −− valuation function $Val_A$
        next($v$) := case
            state = $s^v$ : $b^v$; ... −− assigns value $b^v$ to $v$ in state $s^v$
            TRUE : $v$; −− preserves the current value of $v$
        esac;
    −− LTL/CTL formula $f \in F_A$ which uses variables $t_1, \ldots, t_j$
    LTLSPEC $f(t_1, \ldots, t_j)$ ... CTLSPEC $f(t_1, \ldots, t_j)$
```

Figure 6: *A template NuSMV code used in the transformation from OBA.*

5 Expressiveness of FOAM

To reflect the common guidelines in creating use-cases, FOAM features a number of restrictions on the control flow annotations – guards allowed only at the beginning of a branching (variations and extensions), goto is allowed only at the one of a non-primary scenario), etc. 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 sufficiently general Kripke structure [6] and a related temporal logic formula, which form a typical input used in model-checking theory, can be transformed to a use-case and an annotation group while keeping the semantics. In particular, we show this for Kripke structures that have one initial state and one state 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.[1]

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

Theorem 5.1. *Let K be a Kripke structure without unreachable states such that it has only one initial state i and one state f, in which all computation eventually ends in an infinite cycle. Let F be an $LTL_{\text{-}X}$ or $CTL_{\text{-}X}$ 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. To prove the theorem, we construct UCM with one primary use-case u and with no precedence constraints. Further we construct G with just one annotation group g. We introduce a synthetic step s_{start} as the first step of the main scenario of u. We define the remaining steps of u as the edges in K (note that we treat K as an oriented graph). We add a particular path p_m in K, which starts in i and ends in f (such path has to exist due to our assumptions), as the remaining steps of the main scenario.

Now we iterate the following steps until all vertices and edges in K have been processed (we deem vertices and edges in p_m and the edge forming the infinite cycle $f \to f$ as already processed): We select the path $p = v_1 \to \ldots \to v_n$ in K such that (i) it starts in some of the processed vertices, (ii) when not considering the last vertex of p, the vertices of p are disjunctive, (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 the path p as a variation and the last edge of the path as a step annotated with goto. The variation is attached to the already processed edge (i.e. step) which originates in v_1 and which is not the first step in its scenario.

The annotation group g is constructed as follows. Formula F is used as the temporal logic formula in g. An annotation is introduced to the group for each distinct atomic proposition in the formula. A

[1]Allowing 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 OBA would have to be constructed differently, thus we treat use-cases in this paper to be valid only when they have finite execution.

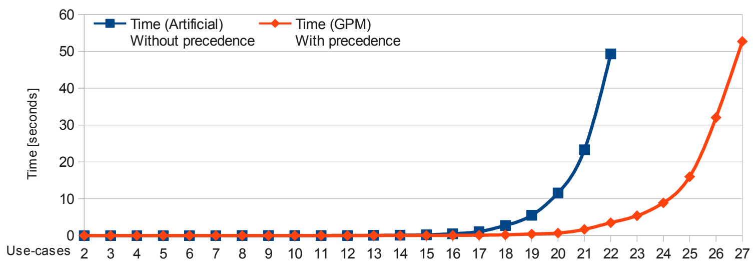


Figure 7: *Evaluation results*

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. □

6 Related Work

There are several other approaches related to FOAM that also formalize and/or verify use-cases.

Model-checking of the Unified Modeling Language (UML) use-cases using SPIN is proposed in [12]. This method assumes that pre/post-conditions of use-cases are expressed in first-order predicate logic. A graph representing the possible sequences of use-cases is constructed from the pre/post-conditions similarly to our *precede* relation. Even though their method supports branching in scenarios, it is restricted to extensions only. Also the *include* relation is not supported. Moreover in contrast to FOAM, the method assumes that use-cases are already provided in a formal notation (predicates). Also the LTL formulae to be verified by SPIN are constructed from the pre/post-conditions only.

In [15], a formal semantics based on LTSs is proposed for use-cases containing *extensions* and *include steps*. The authors utilize LTS to automatically detect *livelocks*. They also propose a method for verifying *refinement* of use-case models, namely checking their equivalence and deterministic reduction. All of the checks focus on global properties of use-case models. The same authors wrote a number of papers about mapping use-cases into several formalisms – POSETs in [17], finite state machines in [14] and LTS+POSETs in [16]. As opposed to FOAM, properties to be verified are pre-defined (FOAM allows for user-defined properties) and branching scenarios are not considered.

In [18], textual use-cases are formalized via reactive Petri nets, taking into account the *include* and *extend* UML relationships and sequencing constraints using pre/post-conditions. The method assumes that use-case steps comply with a restricted English grammar. The approach does not allow expressing other relationships and constraints.

Related are also the methods that map use-cases into the UML activity or sequence diagrams [1, 19, 21, 20, 2]. (Activity diagrams are basically transition systems similar to LTS formalism used in FOAM) These works focus on the *generalization*, *include*, and *extend* relationships in UML. However, such diagrams are not suitable for verifying temporal constraints in use-cases.

There are also many approaches aiming at formalizing UML models in general. For instance in [8] 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 [11]. In contrast to FOAM, these approaches rely on a model in UML that already provides a semi-formalized input in the form of annotations.

7 Evaluation

The complexity of OBA can significantly benefit from the fact that the precedence relation lowers the number of possible use-case sequences, since without it the model-checker would have to traverse all the $n!$ possible orderings for n use-cases.

Our experimental configuration was as follows:

- **CPU:** Core2 Duo CPU P9600 @2.53GHz
- **RAM:** 4GiB RAM
- **OS:** 32bit Ubuntu 10.04 LTS with kernel 2.6.32

- **NuSMV version:** NuSMV-zchaff-2.5.4-i386

First we tested a set of artificial use-cases without any precedence. As it can be seen from the Figure 7, for up to 22 use-cases the required time was under one minute (actually, 49 seconds). With each added use-case, time required to process them increased by factor of 2.2. In the second experiment, we used use-cases (taken from [9]) with a precedence relation defined as depicted in Figure 10.

The generated NuSMV code for both experiments were similar to the example in Figure 11. We executed the following set of commands in order to obtain the results:

```
read_model -i generated-nusmv-code.smv
flatten_hierarchy
encode_variables
build_model
print_usage
quit
```

As expected, with precedence specified required time is smaller and even the graph is not as steep (required time grows with factor of 1.8).

These experiments show that trying all orderings would results in a state-explosion even though FOAM does not interleave steps of use-cases. FOAM assumes a rich usage of precedence to limit the possible use-case sequencing. It is also worth noting that most of the use-cases in a specification are usually independent, so that a partial order reduction can be successfully employed (e.g. just a particular sequencing can be considered). We are currently working on such a method which can identify groups of dependent use-cases and reflect the independence relation in creation of OBA.

8 Discussion and Conclusion

Comparing to our earlier work [13], we have moved from a fixed predefined set of temporal annotations to a system which provides sufficient variability for different application domains. We have also unified the notion of annotations attached to use-case steps. Unlike [13], where *goto*, *include*, *abort* "actions" were considered as special concepts, in FOAM we encode them as annotations.

Currently, annotations have to be added manually during the preparation of a specification; nevertheless we are working on a FOAM extension which, in an automated way, will propose addition of annotations.

References

[1] J. Almendros-Jimenez and L. Iribarne. Describing Use-Case Relationships with Sequence Diagrams. *TCJ*, 50(1):116–128, October 2006.

[2] Jesús Manuel Almendros-Jiménez and Luis Iribarne. Describing use cases with activity charts. In *Proc. of MIS'04, Salzburg, Austria*, pages 141–159. Springer, 2004.

[3] K Berg and M Aksit. Use Cases in Object-Oriented Software Development. *Language*, 1999.

[4] K G Van Den Berg. Control-Flow Semantics of Use Cases in UML. *Science*, pages 1–18, 1999.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. of CAV 2002*, volume 2404 of *LNCS*. Springer, July 2002.

[6] Edmud M Clarke, Orna Grumberg, and Doron A Peled. *Model checking*. MIT Press, Cambridge, MA, 1999.

[7] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, Boston, MA, USA, 2000.

[8] Sima Emadi. Mapping annotated sequence diagram to a Petri net notation for reliability evaluation. *Technology and Computer (ICETC)*, pages 57–61, 2010.

[9] D. Firesmith. GPM SRS, 2003. .

[10] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[11] OMG. UML for modeling QoS and fault tolerance characteristics and mechanisms. OMG document formal/2008-04-05, 2008.

[12] Yoshiyuki Shinkawa. Model Checking for UML Use Cases. In *SERA*, volume 150 of *SCI*. Springer, 2008.

[13] Viliam Simko, David Hauzar, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Verifying temporal properties of use-cases in natural language. In *Postproc. of FACS'2011*, LNCS. Springer, September 2011.

[14] Daniel Sinnig, Patrice Chalin, and Ferhat Khendek. Consistency between task models and use cases. In *EIS'08*, volume 4940 of *LNCS*, pages 71–88. Springer, 2008.

[15] Daniel Sinnig, Patrice Chalin, and Ferhat Khendek. LTS semantics for use case models. In *Proc. of SAC'09, Honolulu, Hawaii*, pages 365–370. ACM, 2009.

[16] Daniel Sinnig, Ferhat Khendek, and Patrice Chalin. Partial order semantics for use case and task models. *FAC*, 23(3):307–332, June 2010.

[17] D Sinning, P Chalin, and F Khendek. Towards a Common Semantic Foundation for Use Cases and Task Models. *ENTCS*, 183:73–88, July 2007.

[18] Stéphane S. Somé. Formalization of Textual Use Cases Based on Petri Nets. *IJSEKE*, 20(05):695, 2010.

[19] T. Yue and L. Briand. An automated approach to transform use cases into activity diagrams. *Modelling Foundations and Applications*, pages 337–353, 2010.

[20] Tao Yue, Shaukat Ali, and Lionel Briand. Automated Transition from Use Cases to UML State Machines to Support State-Based Testing. In *MFA*, volume 6698 of *LNCS*, pages 115–131. Springer, 2011.

[21] Tao Yue, Lionel C. Briand, and Yvan Labiche. Facilitating the Transition from Use Case Models to Analysis Models:Approach and Experiments. *TOSEM*, 2011.

9 Appendix

```
AnnotatedUseCaseModel:
    (useCases += UseCase)*;

UseCase:
    "UseCase" ":" (useCaseName = NON_ANNOTATION_STRING)
    (precedingUseCases += PrecedingUseCase)*
    (mainScenario = Scenario)
    (branches += Branching)*;

PrecedingUseCase:
    "Preceding" ":" (useCaseName = NON_ANNOTATION_STRING);

Branching:
    (Extension | Variation);

Extension:
    "Extension" ":" (label = ID) "." (text = NON_ANNOTATION_STRING)
    ScenarioSteps;

Variation:
    "Variation" ":" (label = ID) "." (text = NON_ANNOTATION_STRING)
    ScenarioSteps;

ScenarioSteps:
    (steps += UseCaseStep)+;

UseCaseStep:
    (label = ID) "." (text = NON_ANNOTATION_STRING)
    (annotations += Annotation)*;

Annotation:
    '#' (name = ID) (':' (param = NON_ANNOTATION_STRING) )?;

terminal NON_ANNOTATION_STRING : [^#\n]*;
```

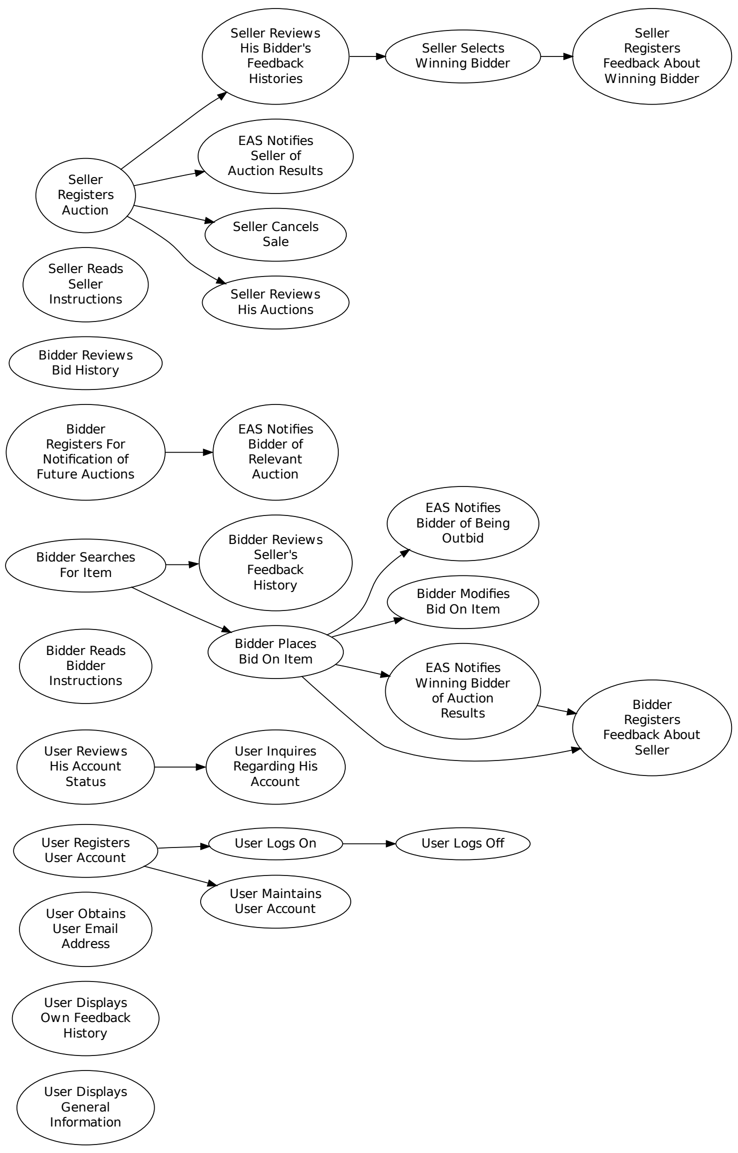
Figure 8: *Concrete syntax for annotated textual use-cases. (Xtext notation)*

```
CustomTemporalAnnotationsModel:
    (groups += TemporalAnnotationGroup)*;

TemporalAnnotationGroup:
    "Annotations" ":"
    (aliases += NSNAME ","?)+
    (formulas += Formula)+;

Formula:
    ("CTL"|"LTL") (formula=EXPRESSION)
    (comment=STRING);
```

Figure 9: *Concrete syntax of TADL. (Xtext notation)*



Seller Reviews His Bidder's Feedback Histories
Seller Selects Winning Bidder
Seller Registers Feedback About Winning Bidder
EAS Notifies Seller of Auction Results
Seller Registers Auction
Seller Cancels Sale
Seller Reads Seller Instructions
Seller Reviews His Auctions
Bidder Reviews Bid History
Bidder Registers For Notification of Future Auctions
EAS Notifies Bidder of Relevant Auction
EAS Notifies Bidder of Being Outbid
Bidder Searches For Item
Bidder Reviews Seller's Feedback History
Bidder Modifies Bid On Item
Bidder Places Bid On Item
Bidder Reads Bidder Instructions
EAS Notifies Winning Bidder of Auction Results
Bidder Registers Feedback About Seller
User Reviews His Account Status
User Inquires Regarding His Account
User Registers User Account
User Logs On
User Logs Off
User Maintains User Account
User Obtains User Email Address
User Displays Own Feedback History
User Displays General Information

Figure 10: *Precedence relation of 28 use-cases in our experiment taken from the GPM Specification [9].*

```
MODULE main
    VAR done_1 : boolean;
    ASSIGN
        init(done_1) := FALSE;
        next(done_1) := case
            s=succ_1 : TRUE;
            TRUE : done_1;
        esac;
    VAR done_2 : boolean;
    ASSIGN
        init(done_2) := FALSE;
        next(done_2) := case
            s=succ_2 : TRUE;
            TRUE : done_2;
        esac;
    VAR done_3 : boolean;
    ASSIGN
        init(done_3) := FALSE;
        next(done_3) := case
            s=succ_3 : TRUE;
            TRUE : done_3;
        esac;

    VAR s : {init_0, succ_0, init_1, init_2, init_3, succ_1, succ_2, succ_3 };
    ASSIGN
        init(s) := init_0;
        next(s) := case
            s=init_0 : {succ_0, init_1, init_2, init_3}; -- initial state
            s=succ_0 & !( done_1 & done_2 & done_3 ) : init_0;
            s=succ_0 : succ_0; -- final state
            s = init_1 & !(!done_1) : init_0; -- scheduling UCA(1)
            s = init_1 : succ_1;
            s = succ_1 : init_0; -- returning from UCA(1)
            s = init_2 & !(!done_2) : init_0; -- scheduling UCA(2)
            s = init_2 : succ_2;
            s = succ_2 : init_0; -- returning from UCA(2)
            s = init_3 & !(!done_3) : init_0; -- scheduling UCA(3)
            s = init_3 : succ_3;
            s = succ_3 : init_0; -- returning from UCA(3)
        esac;
```

Figure 11: *An instance of NuSMV code in our experiment generated from 3 use-cases without precedence relation.*