2012 38th Euromicro Conference on Software Engineering and Advanced Applications

FOAM : A Lightweight Method for Verification of Use-Cases

Viliam Simko*, Petr Hnetynka*
* *Charles University, Faculty of Mathematics and Physics*
Department of Distributed and Dependable Systems
Malostranské náměstí 25
Prague 1, 118 00, Czech Republic

Tomas Bures*†, Frantisek Plasil*†
† *Academy of Sciences of the Czech Republic*
Institute of Computer Science
Pod Vodárenskou věží 2
Prague 8, 182 07, Czech Republic

***Abstract*—The advantage of textual use-cases 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 Formal Verification of Annotated Use-Case Models (FOAM) method which features simple user-definable annotations, 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.**

***Keywords*- Requirements, Verification, CTL, LTL, NuSMV**

I. Introduction

Specification of functional requirements using textual use-cases [1] 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 [2] 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 [2] 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.

II. Textual use-cases

To date, there is no standardized form of use-cases. To circumvent this, we adhere to the widely accepted format proposed in [1].

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.

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

978-0-7695-4790-9/12 $26.00 © 2012 IEEE
DOI 10.1109/SEAA.2012.15

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* [3], [4], [5], 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).

III. Overview of the Solution

In FOAM, we define two sorts of annotations: (i) *flow annotations* expressing control flow of use-cases, and (ii) *temporal annotation* expressing temporal invariants to be satisfied by use-cases.

A. 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:

$\langle$goto:$s\rangle$: This annotation represents a jump to the target step s within the use-case.
$\langle$include:$u\rangle$: Specifies inclusion of another use-case u.
$\langle$abort$\rangle$: Expresses abort of the scenario.
The following two annotations assume existence of (global) boolean variables $b_1, \dots, b_n$ initialized to $false$.
$\langle$mark:$b_i\rangle$: This annotation sets b_i to $true$.
$\langle$guard:$f(b_1, \dots, b_k)\rangle$: The f parameter of this annotation is a propositional logic formula over the boolean variables $b_1, \dots, b_k$. The annotation serves as a guard for extensions and variations.

B. 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. This is possible because FOAM distinguishes two types of users:

```
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"
```

Figure 2. Example of a custom annotation (template) defined in TADL.

(a) experts in temporal logic who prepare templates of annotations in Temporal Annotation Definition Language (TADL) in the form illustrated by Figure 2,
(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.

In the UCM example from the Figure 1, there are annotations $\langle$create$\rangle$ and $\langle$use$\rangle$ attached to textual use-cases. Using the TADL template from Figure 2, these annotations would be transformed into temporal 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$_{city}$)]

C. Verification process

The automated verification process of FOAM presumes that annotations are already attached to use-case steps; either manually or semi-automatically. Annotated use-cases are then transformed into an Labeled Transition System (LTS). This model is passed to the NuSMV model checker [6] 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.

IV. 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 3. 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 IV-F).

A. 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

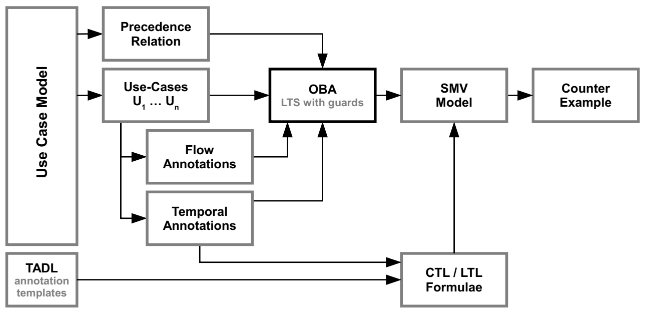

Figure 3. Verification method – an overview

means that we explicitly capture use-case steps (along with annotations attached to them), extensions and variations.

Def.1 (Annotated textual use-case). An 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.IV-A) 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.

Def.2 (Use-Case Model). A 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 U_M^p.

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

B. 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:

Def.3 (Overall Behavior Automaton). OBA is a tuple $A = (V, init_0, \tau, B, AP, Val, Lab, Guards)$

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

C. Building Overall Behavior Automaton – step #1

Now we show OBA construction from a UCM. This first step of the transformation process is formally described in [7] as a set of 12 inference rules which declaratively express the logical constraints on OBA based on the input UCM. Hence, the inference rules provide a logical theory the model of which is OBA. 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 4. There is an initial state $init_0$ with branches to particular sub-automatons, each corresponding to one of the use-cases $U_1, \ldots, U_n$.

In order to capture non-deterministic branching, we introduce a boolean variable $done_u$ for each primary use-case u and also a transition guarded by a predicate over $done_u$ variables, which reflects the precedence relation $Prec_M$. This way, OBA captures the non-determinism in sequencing the use-cases where each use-case is executed exactly once.

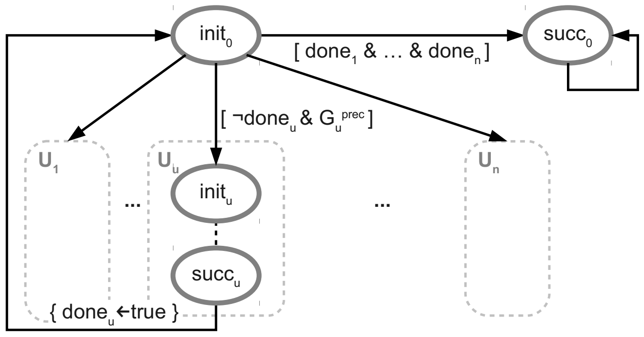

Figure 4. OBA constructed from use-cases $U_1, \ldots, U_n$

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.

An inclusion of a use-case expressed as $\langle$include:$c\rangle$ attached to a step x introduces a boolean variable $incl_{x,c}$ representing the "procedure call" from x to the use-case c.

Temporal annotations from a use-case translate to OBA as atomic propositions attached to corresponding states.

D. 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). We need to additionally introduce the guards for the default step and the unguarded variations and extensions so as the mutual exclusivity holds.

E. 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.

F. Verification using NuSMV

We have implemented a verification of OBA using NuSMV. Thanks to the usage of NuSMV, both CTL and LTL formulae are allowed in defining annotations.

Transformation of OBA into the NuSMV input language is straightforward. 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.

The only difficulty stems from the fact that NuSMV does not support non-deterministic choice between rules which had to be "emulated" in our implementation. Further details can be found in [7].

V. 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 [8]. 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 [9], 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 [10], finite state machines in [11] and LTS+POSETs in [12]. As opposed to FOAM, properties to be verified are pre-defined (FOAM allows for user-defined properties) and branching scenarios are not considered.

In [13], 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 [14], [15], [16], [17], [18]. 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 [19] the authors propose an automated method for translating UML sequence diagrams into Petri nets for evaluating reliability of software architectures. In contrast to FOAM, these approaches rely on a model in UML that already provides a semi-formalized input in the form of annotations.

VI. Evaluation

We evaluated the complexity of our verification method in an experiment (details in [7]). In the model-checking phase our results show sensitivity to the structure of the precedence relation. Even though the complexity grows exponentially with the number of use-cases in general, the growth is slower when the precedence relation is employed. Without the precedence relation, we were able to verify just 22 use-cases under one minute. Then we took use-cases from [5] with precedence defined, which allowed us to verify 27 use-cases under one minute.

The numbers may sound quite limiting for a real project (usually with hundreds of use-cases), but they only reflect a situation, where we need to verify several dependent use-cases together. However, 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 the construction of OBA.

VII. Conclusion and Future Work

In this paper we presented the FOAM method for formal verification of use-cases annotated by user-definable temporal annotation. Although FOAM is fully automated (construction of OBA, transformation to NuSMV, model-checking), the 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 offer annotations in a given context.

Acknowledgments

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.

References

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

[2] V. Simko, D. Hauzar, T. Bures, P. Hnetynka, and F. Plasil, "Verifying temporal properties of use-cases in natural language," in *Postproc. of FACS'2011*, ser. LNCS. Springer, September 2011.

[3] K. G. V. D. Berg, "Control-Flow Semantics of Use Cases in UML," *Science*, pp. 1–18, 1999.

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

[5] D. Firesmith, "GPM SRS," 2003, http://www.it.uu.se/edu/course/homepage/pvt/SRS.pdf.

[6] 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*, ser. LNCS, vol. 2404. Springer, July 2002.

[7] V. Simko, P. Hnetynka, T. Bures, and F. Plasil, "Formal verification of annotated use-cases," Charles University in Prague, Tech. Rep. 2012/2, June 2012.

[8] Y. Shinkawa, "Model Checking for UML Use Cases," in *SERA*, ser. SCI. Springer, 2008, vol. 150.

[9] D. Sinnig, P. Chalin, and F. Khendek, "LTS semantics for use case models," in *Proc. of SAC'09, Honolulu, Hawaii*. ACM, 2009, pp. 365–370.

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

[11] D. Sinnig, P. Chalin, and F. Khendek, "Consistency between task models and use cases," in *EIS'08*, ser. LNCS. Springer, 2008, vol. 4940, pp. 71–88.

[12] D. Sinnig, F. Khendek, and P. Chalin, "Partial order semantics for use case and task models," *FAC*, vol. 23, no. 3, pp. 307–332, June 2010.

[13] S. S. Somé, "Formalization of Textual Use Cases Based on Petri Nets," *IJSEKE*, vol. 20, no. 05, p. 695, 2010.

[14] J. Almendros-Jimenez and L. Iribarne, "Describing Use-Case Relationships with Sequence Diagrams," *TCJ*, vol. 50, no. 1, pp. 116–128, Oct. 2006.

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

[16] T. Yue, L. C. Briand, and Y. Labiche, "Facilitating the Transition from Use Case Models to Analysis Models:Approach and Experiments," *TOSEM*, 2011.

[17] T. Yue, S. Ali, and L. Briand, "Automated Transition from Use Cases to UML State Machines to Support State-Based Testing," in *MFA*, ser. LNCS. Springer, 2011, vol. 6698, pp. 115–131.

[18] J. M. Almendros-Jiménez and L. Iribarne, "Describing use cases with activity charts," in *Proc. of MIS'04, Salzburg, Austria*. Springer, 2004, pp. 141–159.

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