Automatic Formal Model Derivation from Use Cases

Amine Raji — Philippe Dhaussy

Laboratoire d'Informatique des Systèmes Complexes EA3883
Développement des Technologies Nouvelles–ENSIETA
Brest, France
{amine.raji ; dhaussy}@ensieta.fr

ABSTRACT. *In industrial development practices, verification process takes a great part of the development time and budget to satisfy quality and reliability. The use of formal verification at industrial scale still difficult, expensive and requires lot of time. This is due to the size and the complexity of manipulated models, but also, to the important gap between requirement models manipulated by different stackholders and formal models required by existing verification tools. In this paper, we fill this gap by providing a set of model transformations to automatically generate, from requirement models, behavioral models used by formal verification tools. Starting from an extended form of use cases, we generate behavior models of environment actors interacting with the system.*

RÉSUMÉ.
En génie logiciel, le processus de vérification requiert beaucoup de temps et d'efforts pour faire face aux contraintes de qualités et de sûreté, ce qui limite le taux d'utilisation des méthodes formelles dans les processus de développement industriels. L'une des principales causes de ce constat est l'écart qui sépare les modèles manipulés par différentes parties prenantes et ceux nécessaires à la vérification formelle. Dans ce papier, on propose de réduire cet écart par le biais d'une suite de transformations de modèles visant à générer les modèles comportementaux, utilisés par les outils de vérification formelle, directement à partir des modèles d'exigences. Notre approche consiste à générer des modèles comportementaux des différents acteurs de l'environnement, interagissant avec le système, directement à partir des cas d'utilisations du système.

KEYWORDS: *requirement models, use cases, model transformation, activity diagram, formal verification.*

MOTS-CLÉS : *modèles d'exigences, cas d'utilisations, transformation de modèles, diagrammes d'activités, vérification formelle*

1. Introduction

Verification of software systems is an important task that aims to check whether design meets intended requirements. Several researchers have proposed valuable techniques and tools to ease and automate the verification process. Formal methods have demonstrated their potential in this area especially through the so called model checking. However, the application of such techniques in industrial practices still limited w.r.t the growing need of quality and reliability of developed software.

In this paper we overcome this shortcoming by presenting a model based approach to bridge the gap between high level requirement models and models required by existing formal verification tools. We focus on the context description part in order to facilitate the derivation of behavioral models of environment entities, referred to as *actor* in the rest of the article, directly from requirements.

Automating the extraction process of environment actors behavior considerably facilitates subsequent verification phases when checking the coherence of the system behavior against environment stimuli. The approach presented in this paper is complementary to our work proposed in [DHA 09]. In this latter, we propose a verification process based on context description and property definition patterns [DWY 99]. Through this paper, we illustrate our approach using the the Crisis Management System (CMS) case study. Figure 1 shows an example of a partial view if the CMS use cases [1].

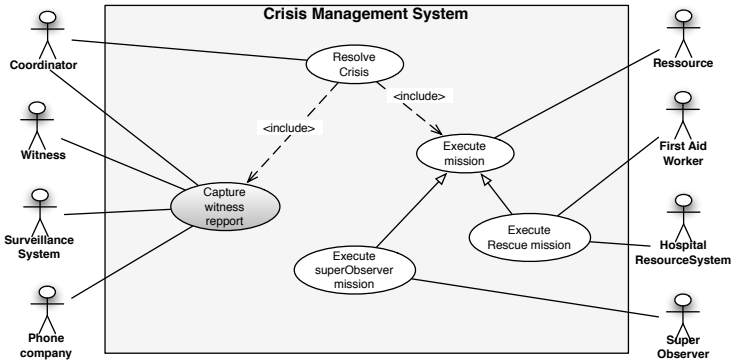

Figure 1. *Excerpt of the CMS Use Case Diagram*

The rest of the paper is organized as follow: Section 2 shows our model based process to fill the gap between requirement specification and formal verification activities. Section 3 presents related works and Section 4 discusses future work and draws some conclusions.

1. A detailed description of the used case study can be found at http://www.cs.mcgill.ca/ joerg/taosd/TAOSD/TAOSD.html

2. Our Approach

We propose a model-driven approach to capture and formalize system requirements in order to ensure that the SUS meets intended safety and reliability requirements as defined by stackholders. We use extended use cases to capture system requirements as well as possible exceptions and corresponding handlers if any. The idea behind this step is to gather all useful information about the system behavior and its interactions with environment actors. Constructed use cases are then used as input in our model transformation to generate formal models directly that can be processed by a model checker. The benefits of such approach consists in bridging the gap between informal models manipulated by different stackholders with existing formal verification tools.

2.1. *Capturing system requirements using extended use cases*

Extending traditional use cases with exception handling was originally proposed in [MUS 08]. In use cases, an exception occurrence endangers the completion of the actor's goal, suspending the normal interaction temporarily or for good. To guarantee reliable service or to ensure safety, special interactions with the environment might be necessary. These special interactions can be described in *handler use cases*.

Use Case: Capture Witness Report

Intention: The Coordinator intends to create a crisis record based on the information obtained from witness. Workers to execute appropriate missions.
Level: User-goal
Main Success Scenario:
1) Witness report crisis to the coordinator.
2) Coordinator requests Witness to provide his identification.
3) Witness provide his ids to the coordinator.
4) Coordinator provides witness information to System as reported by the witness.
5) System contacts PhoneCompany to verify witness information.
6) PhoneCompany sends information to System.
7) System validates information received from the PhoneCompany.
8) Coordinator informs System of location and type of crisis as reported by the witness.
9) System provides Coordinator with a crisis-focused checklist.
10) Coordinator provides crisis information to System as reported by the witness.
11) System provides crisis checklist
12) System assigns an initial emergency level to the crisis and sets the crisis status to active.
Use case ends in «Success».
Extensions:
2a,3a. Exception {CallDisconnected}
The base use case terminates.
Use case ends in «failure»
7a. Exception{InformationMismatch}
//information does not match information received from Witness.
The base use case is terminated.
Use case ends in «failure»
8a. Exception{CallDisconnected}
Use case continues at step 11 without crisis information.

Figure 2. *Capture Witness Report Use Case*

Figure 2 shows a sample exceptional use case of the CMS example to illustrate how exceptions are integrated in textual use cases. Identified exception situations are associated with the occurring interaction, specified in the *extension* section of the use case. Handlers are invoked when an exception is triggered during the execution or the

use of the associated context or nested context to set the system back to a coherent state.

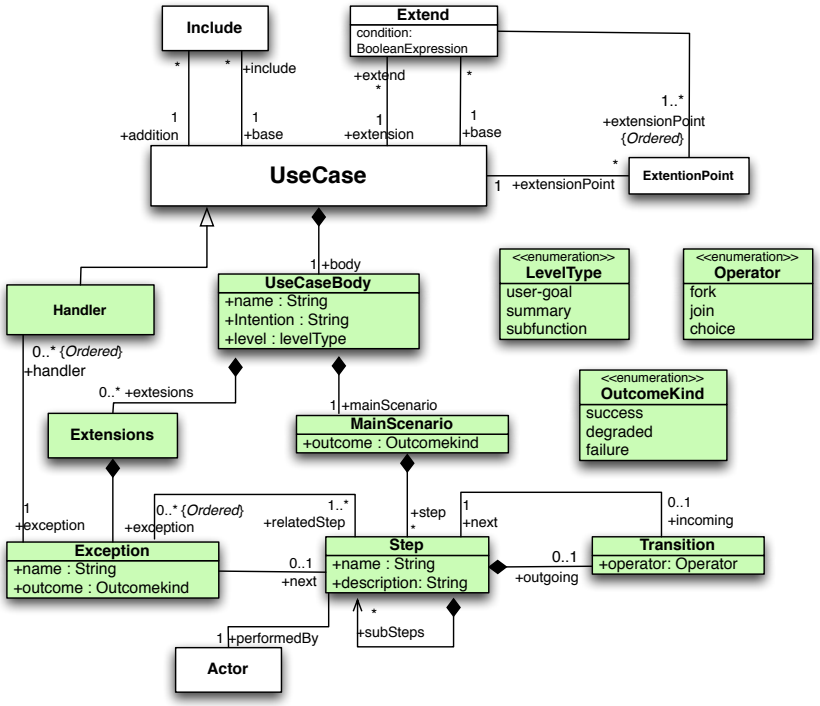

Figure 3. *Metamodel of extended use case integrated in the metamodel of UML*

We propose in figure 3 a metamodel of the extended use cases. The classes with a white background are imported from the UML metamodel, and should be related to the identical ones presented in [OMG 07]. The classes with the filled background was introduced to deal with exceptions and handlers in traditional use cases.

2.2. *Visualization of use case behavior with activity diagrams*

In this section we focus on mapping semi-formal specification expressed in term of extended use cases to a more formal specification, activity diagrams. Our model transformation is partially inspired form the work presented in [GUT 08]. Authors in the cited paper propose a method for representing functional requirements by automatically transforming use cases to activity diagrams. However, proposed use cases doesn't support the handling of identified exceptions. In this paper, we propose a model transformation of extended use cases with handler to UML2 activity diagrams. Before to attempting the transformation of any use case to activity diagrams following

our rules, it is important to pre-process the use cases to ensure that they are conform to the metamodel defined in figure 3. Main scenario steps have to be described in a structured natural language to be transformed into actions in the generated activity diagram. The used structured language is described in RDL (Requirement Description Language) introduced in [NEB 03].

Then, the process of transforming extended use cases to activity diagrams consists in applying transformation rules listed bellow:

1) generate an *activity* for each use case,

2) generate an *activity partition* for each actor and for the system,

3) generate an *action* for each *Step* in the main scenario,

4) add generated *actions* to the *activity partition* of the corresponding actor,

5) generate a *decision node* for each exception and a new *activity* for each handler,

6) generate an *activity final node* for each outcome in the use case,

7) link all generated elements using control flow.

Details about listed transformation rules can be found in [GUT 08].

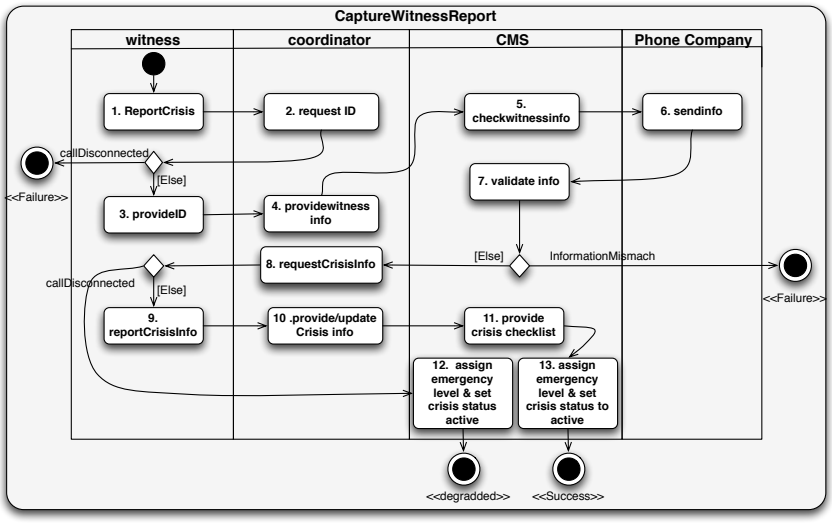

Figure 4. *Generated activity diagram from the CaptureWitnessReport use case*

2.3. *Extracting environment actors behavior*

We propose an algorithmic approach to generate separate activity diagram for each identified actor in the considered use case diagram (Table 1). The algorithm extracts nodes and edges related to the each actor then links them together using control flow.

It consists in applying four rules to transform each activity group into a single activity diagram : *R1:* Create the initial node *R2:* Process activity region owned elements *R3:* Process crossing edges *R4:* Create the activity final node.

Table 1. *Synthesis algorithm rules*

Rule	Description
R1	In the source AD only one activity group can contains the initial node, it corresponds to the actor who performs the first action within the AD. For the remaining activity groups, the rule consists in creating a new initial node.
R2	This rule consists to parse the activity region owned elements to add them to the corresponding newly created activity diagram. Activity elements are grouped into activity nodes (A), control nodes (C) and edges (E). Activity nodes are actions while possible control nodes are : final nodes, fork nodes, join nodes, merge nodes and decision nodes. As stated before, we consider that an edge is owned by an activity group Gi only if its source element is also owned by Gi.
R3	For edges that link two nodes owned by two different regions the algorithm consists in creating an instance of *AcceptEventAction* class. The created instance represents the acceptance of a signal coming from outside the considered activity group. We consider that the execution of each action is equivalent to a SendSignalAction. The created AcceptEventAction waits for the SendSignalAction of the source action of the crossing edge. The created AcceptEventAction is labeled with a concatenation of the name of the group containing the edge (the actor's name) and the name of the source action.
R4	Each created AD have to contain at least one activity final node. Activity groups that doesn't contain any activity final node, finishes their behavior with an outgoing crossing edge (the activity final node is owned by another group in the source activity diagram). In the case the last edge targets an activity final node, we create a new activity final node in the considered AD labeled with the same *outcomeKind* stereotype. But if the last edge is targeting an action in a different group, the actor finished its behavior in success.

The result of applying our synthesis algorithm on the activity diagram of figure 4 is shown in figure 5. We have generated an activity diagram for each actor participating in the capture witness report use case (Fig. 2).

3. related works

Bastide proposes in [BAS 09] an integration of user tasks model (named CTT model) to provide an unambiguous model of the behavior of UML use cases. However, exception definition and handler was not supported in the proposed metamodel. The work presented in [ALM 04] describes an approach to translate use case-based functional requirements to activity charts. The source models are use cases diagrams with support of inclusion and generalization relationships. However the method is restricted to model sequences of use cases and not the behavior of each. Authors in [GUT 08]

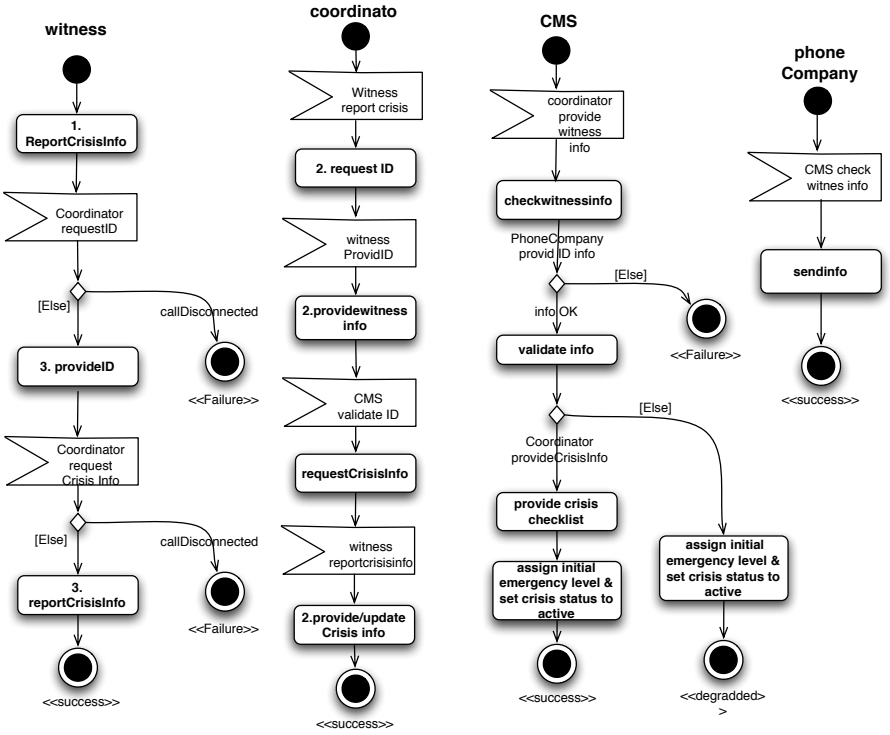

Figure 5. *Generated activity diagram for actors of the CaptureWitnessReport use case*

presents a model based approach to generate an activity diagrams modeling the use case scenario. The proposed metamodel does not take into account how the system could retrieve a normal state after the occurrence of an exception.

Mustafiz *et. al* [MUS 09] propose an algorithm that transforms dependability-focused use cases with handlers into activity diagrams. The transformation takes textual use cases description as source to produce activity diagram model respecting the use case hierarchy source model. Our approach differs in the sense that we begin with informal requirements specification, namely extended use cases, apply a model-driven process to map requirements to activity diagrams, and then, automatically extract formal behavioral models of actors that interact with the system using our synthesis algorithm.

4. Conclusion and future work

In this paper we have proposed a metamodel of extended use cases with handlers that address detected exceptions. Exceptional situations are less common and hence

the behavior of the system in such situations is less obvious. Therefore, the proposed metamodel represents a good starting point for the identification of environment actors that might interact with the system. We have also proposed an approach to automatically synthesis environment entities behavior directly from constructed use cases. We have generated an activity diagram that describe the behavior of each use case using our model transformation rules. Then, we extract the behavior of each actor participating to the activity in a separate activity diagram. The motivation behind this contribution is to ease the use of formal verification techniques by providing early context descriptions with enough precision to feed formal verification tools. To the best of our knowledge, there is no similar work dealing with this particular problem.

However some drawbacks have appeared at the time of generating activity diagrams from extended use cases. A possible loss of flexibility may occur during use case description because we have based our generation rules on simple structured sentences to describe use case scenario steps. Also, our generation process handles only the *Decision Nodes* among different control nodes proposed by the UML2 metamodel.

As a future work, we firstly plan to extend our work to deal with the rest of activity control nodes in order to produce more precis activity diagrams. Secondly, we would like to implement and to incorporate our method in a CASE tool. And finally, we will integrate our approach in the whole verification process.

5. References

[ALM 04] ALMENDROS-JIMENEZ J., IRIBARNE L., "Describing use cases with activity charts", *Metainformatics*, vol. 3511 of LNCS. Springer, 2004, p. 141–159.

[BAS 09] BASTIDE R., "An Integration of Task and Use-Case Meta-models", *Human-Computer Interaction. New Trends*, , 2009.

[DHA 09] DHAUSSY P., PILLAIN P.-Y., CREFF S., RAJI A., TRAON Y. L., BAUDRY B., "Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation", *MoDELS*, vol. LNCS 5795, 2009, p. 438–452.

[DWY 99] DWYER M. B., AVRUNIN G. S., CORBETT J. C., "Patterns in Property Specifications for Finite-State Verification", *ICSE*, , 1999, p. 411–420.

[GUT 08] GUTIÉRREZ J., NEBUT C., ESCALONA M., MEJÍAS M., "Visualization of use cases through automatically generated activity diagrams", *MODELS*, , 2008.

[MUS 08] MUSTAFIZ S., KIENZLE J., BERLIZEV A., "Addressing degraded service outcomes and exceptional modes of operation in Behavioural Models", *Proceedings of the RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, , 2008.

[MUS 09] MUSTAFIZ S., KIENZLE J., VANGHELUWE H., "Model transformation of dependability-focused requirements models", *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, , 2009.

[NEB 03] NEBUT C., FLEUREY F., LEÂ TRAON Y., JÉZÉQUEL J.-M., "A requirement-based approach to test product families", *5th Â Intl. Â Workshop Â on Â Product Family Engineering Â (PFE-5)*, , 2003.

[OMG 07] OMG, "UML 2.1.2 Superstructure", , 2007, p. 1–738.