Automating Context Description for Software Formal Verification

Amine Raji
LISyC-ENSIETA
Université Européenne de Bretagne
2 rue François Verny
29806, Brest Cedex 9, France
amine.raji@ensieta.fr

Philippe Dhaussy
LISyC-ENSIETA
Université Européenne de Bretagne
2 rue François Verny
29806, Brest Cedex 9, France
philippe.dhaussy@ensieta.fr

Bruno Aizier
LISyC-ENSIETA
Université Européenne de Bretagne
2 rue François Verny
29806, Brest Cedex 9, France
bruno.aizier@ensieta.fr

ABSTRACT

Formal methods have increasingly been recognized as effective techniques for automating software verifications to satisfy quality and reliability. However, using such techniques within industrial development processes takes an important part of the development time and budget due to the complexity of developed software. Context aware techniques can circumvent this complexity by reducing the scope of the verification to some precise system configurations. Unfortunately, few existing approaches provide support for this crucial task and mainly rely on significant effort and expertise of the engineer. In this paper, we propose a tool-based framework that automate the description of the system contexts to improve the integration of formal verification techniques into industrial engineering methods. Models describing contexts are automatically generated from use cases and scenarios using model transformations. Then, requirements are checked considering generated contexts to reduce the complexity of the proof.

Keywords

Formal verifications, Context Description Language, model transformation, property patterns, use cases, UML activity diagram

1. INTRODUCTION

Software verification is an integral part of the software development lifecycle, whose goal is to assure that software fully satisfies all the expected requirements. Researchers have developed and investigated a wide variety of approaches, but techniques based on mathematical modeling of program behavior have been a particular focus since they offer the promise of both finding errors and assuring important program properties. Formal software analysis, *aka.* formal methods, have demonstrated their potential in this area especially through the so called model checking.

Over the past decade, software applications have grown significantly larger and more complex in terms of the sheer size of their descriptions and the capabilities they are designed to implement. Despite the best efforts of software developers and researchers, the techniques available to validate software systems are not keeping pace with challenges of modern software systems [8]. Thus, formal methods and there related tools are sometimes seen to be too complicated and time consuming to be integrated to the development lifecycle considering projects' constraints (cost and time).

Several researchers have proposed valuable techniques and tools to support the process of formal verification, but only few of them have focused on their integration within today software engineering practices. Nakagawa *et al.*[13] have developed an approach to bridge the gap between formal methods and requirements analysis based on model transformation techniques. They developed a tool to be integrated to the software development process which allows the derivation of VDM++ formal specifications from KAOS models (requirement specifications). KAOS [5] is a method for requirements analysis, whereas VDM++ [9] is a formal specification language for object-oriented systems. However, generating formal specifications from high level requirements is insufficient to prove or disprove a given property because of the combinatorial explosion problem. Consequently, the undecidability limitation reduces the integration of existing model checking tools into industrial engineering practices. The availability of a more systematic and automated approach that takes into account the scalability issues would therefore benefit the process of model checking requirements.

In this paper, we propose a formal, tool supported framework that combines context descriptions and model transformations to improve the integration of formal verification techniques into industrial engineering methods. The framework is defined as an iterative process that consists of tree phases. First, in the *context specification* phase, we have proposed an extension to the UML metamodel to capture important interactions between the system[1] and its environment. This extension consists of a UML profile, named *eXtended Use Case* (*XUC*), that combines use cases and scenarios to model precise contexts. In the *activities derivation* phase, *XUCs* are used to automatically generate an activity diagram describing the complete behavior of each use case using our model transformation rules. Finally, in the

[1]By system we refer to the model to be validated.

behavior synthesis phase, behavioral models of environment actors are synthesized from obtained activities by means of the proposed synthesis algorithm.

In summary, we provide a tool-supported approach for producing context models at early phase of the development process. The steps producing these artifacts are fully automated and generate behavioral models of identified environment actors directly from use cases and scenarios.

This paper is organized as follows. Section 2 illustrates the problem using the *S_CP* system described in [6]. Section 3 details main features of our approach. Section 4 provides an illustration case study on a real event-driven system, a military submarine Mine Hunting System and reports on our experiments and results. Section 5 discusses how the approach relates to other applications into industrial engineering processes and presents some related works. Section 6 provide a summary and some remarks about future work.

2. MOTIVATION

Consider the *S_CP* case study described in [6]. *S_CP* is the software part of an anti-aircraft system developed by one of our industrial partners. It controls the internal modes of the system, its physical devices (radars, sensors, actuators...) and their actions in response to incoming signals from the environment. Suppose that engineers want to check if the following requirement is satisfied on their conceptual models to early detect possible inconsistencies.

R1 = "On reception of an evtRequestLogin *message from an unlogged* HMI[2]*: if the pair userName/password is invalid the* S_CP *shall send an* evtAck *message to the* HMI *with result to* FALSE *to refuse operator login, else it shall send an* evtAck *message the* HMI *with result to TRUE to authorize operator to continue setting up the mission."*

The *S_CP* system can be in one of several modes during its lifecycle, i.e. *Off - Initialization - standby - active*, each mode might have submodes (*standby: Idle, Logged - Active: Preparation, Operational*), and for each mode, interactions between *S_CP* and external equipments might be completely different. Consequently, checking *R1* over the whole system automaton using an existing model checker might lead to a combinatorial explosion due to the large number of possible inter-leavings between the executions of the (concurrent) threads in the system. Which leads to undecidable results about the validity of the property.

An engineer, faced with this example of combinatorial explosion could make use of some available algorithmic techniques that reduce the size of the state space, e.g. partial order reductions and/or reduces the scope of the verification. This latter can be made by selecting specific execution configuration of the system according to the property to check, e.g. checking R1 during the *S_CP* Initialization mode. In this paper we propose an automated approach to generate behavioral models of all environment actors interacting with the system in each execution use case. These context models are then composed with the system automata to reduce the corresponding state space.

In next sections, we shall demonstrate how contexts are captured and specified using XUC (Section 3.1), how activities are generated for each use case (Section 3.2), and finally, how behavioral models of identified contexts are synthesized from generated activities (Section 3.3).

[2]HMI stands for Human Machine Interface

3. APPROACH

This approach is complementary with the one presented in [6]. In this latter, we proposed a context aware verification process that make use of a Context Description Language. CDL was proposed to fill the gap between user models[3] and formal models required to perform formal verifications. CDL is a Domain Specific Language presented in the form of UML like graphical diagrams (subset of activity and sequence diagrams) to capture environment interactions. Additionally, a textual syntax is proposed to formalize properties to be checked using property description patterns [7, 11]. CDL is designed so that formal artifacts required by existing model checkers could be automatically generated from it. This generation is currently implemented in our prototype tool named OBP (*Observer Based Prover*). When it is supplied with the system automaton and a CDL model, OBP generates a global Labeled Transition System (LTS) representing the system model composed with the observer automaton of the property to be checked and the LTS representing behavior of the context in which the verification is restricted to. This verification technique has demonstrated its effectiveness trough several aeronautic and military case studies [6].

Figure 1 shows an overview of the approach. The shaded rectangle highlights the contribution of this paper that bridges the gap between user specification models and CDL models by automating the generation of contexts. Starting from the system use case model and a collection of scenarios describing interactions between the system and its environment, our approach provides a systematic way for extracting environment actors' behavior models. The approach is composed of three phases:

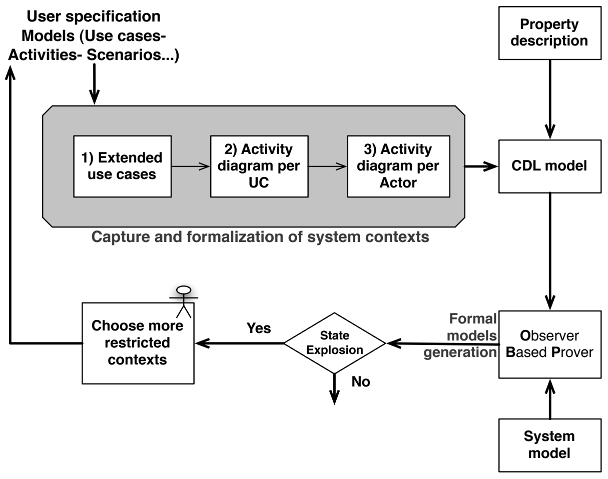
User specification Models (Use cases- Activities- Scenarios...)
Property description
1) Extended use cases
2) Activity diagram per UC
3) Activity diagram per Actor
CDL model
Capture and formalization of system contexts
Choose more restricted contexts
Yes
State Explosion
No
Formal models generation
Observer Based Prover
System model

Figure 1: Approach overview

3.1 UML profile for extended use cases

The entry point of our approach is the elaboration of an extended form of use cases (XUC). We have proposed a UML profile [15] to extend traditional UML use cases with the capability to describe interaction scenarios.

In the proposed profile, a use case body contains two main fields:

[3]models manipulated during development phases of the development process

- *MainScenario*: represents the main success scenario of a use case which describes the standard way of achieving the goal of the use case. Scenario consists in a succession of *steps* which can be related by *transitions*. Each step is to be related to a performer *Actor*. Use cases also propose the possibility to add extensions to the main success scenario in case the interaction takes a different route.

- *Extensions*: the extensions field contains all exceptions that might occur during the main scenario steps. Handlers are invoked when an exception is signaled.

3.2 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, i.e. UML activity diagrams. Our model transformation is partially inspired by the work presented in [10]. Authors in the cited article propose a method for representing functional requirements by automatically transforming use cases to activity diagrams. However, proposed use cases don't support the handling of identified exceptions. In this paper, we propose a model transformation of *XUCs* with handlers to UML2 activity diagrams.

Each exception identified in the use case is represented by a *decisionNode* in the activity diagram with a boolean condition. The condition is labeled with the exception name. Additionally with the decision node, an *activity* representing the exception handler is generated. If no handler is defined for the considered exception, an *ActivityFinalNode* is generated and linked to the decision node using a new *controlFlow*. Generated *ActivityFinalNodes* are labelled with a stereotype corresponding to the *OutputKind* of the related exception. For repeat/ go-to result type the action corresponding to the next step is obtained and linked with the decision node using control flow.

Once all elements are generated, the final step consists in linking all created activity diagram elements using control flow.

3.3 Extracting behavioral model of environment actors

We propose an algorithmic approach to generate a separate activity diagram for each identified actor in the considered use case. The algorithm extracts nodes and edges related to the same actor (based on *activity Partitions*) then links them together using control flow. To preserve the semantics of the global behavior described in the source activity diagram, events triggering the flow between actions and coming from the system or other actors are added to the target activity diagram. These events are represented with instances of *AcceptEventAction* and *SendSignalAction* classes [14]. Thus, the flow between actions in the activity of a specific actor is conditioned with the reception of the corresponding event. To describe the extraction algorithm we propose an algebraic definition of a UML2 activity diagram.

Definition 1 *An activity diagram is a tuple* $AD = \langle A, C, G, E, \Delta \rangle$ *where* A *is a set of activity nodes,* C *the set of control nodes,* E *the set of activity edges,* G *the set of activity groups where* $A \cup C \cup E \subset G$ *and* Δ *defines the control flow relationship such as* $\Delta : (A \cup C) \times A \rightarrow E$.

This definition is not exhaustive since it doesn't represent all the UML2 activity diagram elements. However it is sufficient to explain our proposed algorithm to extract each actor's activities from a general activity diagram.

The extraction algorithm creates an activity diagram for each identified actor (for each instance of the *activityGroup* class). Thus, the master rule to transform each activity group into a single activity diagram (AD) is defined as follows.

```
mapActivityGroup2ActivityDiagram()
   ADi = createInitialNode()
   processOwnedElements(ADi)
   createActivityFinalNode(ADi)
```

According to the UML2 metamodel [14], each element from the sets A, C and E can be associated with more than one activity group. In our approach we consider that activity and control nodes are located in only one activity group at a time since we assume that environment actors don't have to share actions in the generated activity diagram[4]. However, we can have edges that can link two activity nodes (or an activity node with a control node) owned by two different activity groups (we call these edges *crossing edges*). For the extraction process, we assume that a crossing edge is owned by activity group of its source node.

Rule 1: Create the initial node
In the source AD only one activity group could contain 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 the creation of a new initial node. For an activity group *Gi*, initial nodes are created using algorithm 1.

Algorithm 1 Create Initial Node
```
 1: if Gi contains an initial node then
 2:     In ← Gi.getInitialNode
 3:     Ei ← In.getOutgoingEdge
 4: else
 5:     In ← new InitialNode
 6:     Ei ← new Edge
 7: end if
 8: ADi ← new ActivityDiagram
 9: add In and Ei to ADi
10: return ADi
```

Rule 2: Process activity region owned elements
This rule consists of parsing the activity region owned elements in order 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. We consider that an edge is owned by an activity group *Gi* only if its source element is also owned by *Gi*. Algorithm 2 shows how rule 2 is performed.

Rule 3: Process crossing edges
For edges that link two nodes owned by two different regions, the algorithm consists in the creation of an instance

[4]Actions in the source activity diagram are generated from identified steps of the corresponding use case. These steps are described using simple structured sentences with only one subject per sentence. Thus, in our generated activity diagram, actions are related to one and only one actor at a time.

Algorithm 2 Process Owned Elements

```
Require: ActivityDiagram : ADi
 1: n ← getRootElement(Gi)
 2: e ← Gi.getFirstNode
 3: if n isInstanceof InitialNode then
 4:     ADD e to ADi
 5:     LINK e WITH n.getOutgoingEdge.getTarget
 6: else
 7:     for all ed ∈ n.getIncomingEdge do
 8:         processCrossingEdges(ed, ADi)
 9:     end for
10: end if
11: for all element e ∈ Gi do
12:     if e isInstanceof Edge then
13:         if e.getTarget ∉ Gi then
14:             ed ← Gi.getNextNode.getIncomingEdge
15:             processCrossingEdges(ed, ADi)
16:             LINK e WITH EventNode resulting from
                ed
17:             ADD e to ADi
18:         end if
19:     else
20:         ADD e TO ADi
21:         nd ← e.getTarget
22:         ADD nd to ADi
23:         LINK e with nd
24:     end if
25: end for
```

Algorithm 3 Process Crossing Edges

```
Require: ActivityDiagram : ADi and Edge : ed
 1: a ← ed.getSource
 2: aea ← new AcceptEventAction
 3: aea.setName(a.getGroupName + a.getName)
 4: ADD aea to ADi
 5: CREATE Edge between aea and ed.getTarget
```

Algorithm 4 Create Activity Finale Node

```
Require: ActivityDiagram : ADi
 1: ed ← Gi.getLastEdge
 2: afn ← new ActivityFinalNode
 3: if ed.getTarget isInstanceof ActivityFianlNode then
 4:     afn.setStereotype(ed.getTarget.getStereotype)
 5: else
 6:     afn.setStereotype(Success)
 7: end if
 8: ADD afn to ADi
 9: ed.setTarget(afn)
```

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. This represents interactions between different actors so that the global behavior of the activity is preserved.

Rule 4: Create the activity final node
Each created AD have to contain at least one activity final node. Activity groups that don'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.

4. EXPERIMENTS AND RESULTS

This section reports on a case study we conducted to validate our approach. We report here on a Mine Hunting System (*MHS*) case study developed by one of our industrial partners. For the studied system, we had an informal description of the system-to-be, several UML use case models describing different parts of it and a collection of UML sequence diagrams presenting interactions between system components and its environment. Our mission on this case study consisted of the verification of a set of operational requirements that lead to combinatorial explosion if checked on the whole system behavioral model due to the size of generated state space.

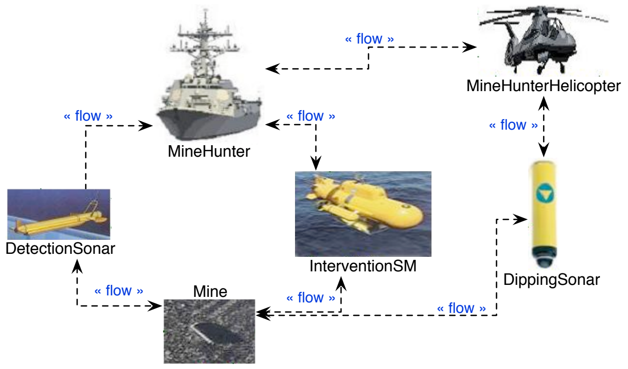

Figure 2: Overview of the Mine Hunting System case study

4.1 Mine Hunting System case study

The case studied consisted in starting from the high-level descriptionsof the *MHS* and applying the approach described in this paper to generated behavioral models of the system context. Figure 2 shows an overview of principal *MHS* components with events and data flow. Motivations behind this generation is to reduce the complexity of the proof and thus lead to decidable results. Having generated all environment actors behavioral models, the state space generated from the synchronization of the system automata and generated automata are used to check provided properties. In all cases, we were able to provide a managed state space to be processed by the used model checker.

In the following, we focus on the detection subsystem of the *MHS*, which is expected to detect, localize, classify and record in a database all underwater objects that might be assimilated to submarine mines. This subsystem is composed of two parts: *1- a detection sonar* for producing pictures of the underwater ground, *2- a classification sonar* that process data collected by the detection sonar and compares it with a data base. During its lifecycle, the *Detection Subsystem* shall interacts with several other *MHS* subsystems (Tactical system, Localization system, Geographic Information System, Intervention system).

All these external subsystems are considered as external actors that interact with the system under study (*detection subsystem*). We applied the approach presented in this paper to automatically generate behavioral models of all identified actors above. Generated models are then used to tightly restrict all possible events that could affect the behavior of the studied system.

An example of operational properties that we wanted to check over design models of the detection subsystem is:

The acquisition period of the sonar navigation settings (pitch, roll, direction, velocity) shall be done in less than 50 milliseconds in order to process the calculation of Beam forming[5].

This property is formalized using one of our property description patterns[6] (*Precedence* pattern) and automatically translated to an observer automaton (A_{obs}) using *OBP*. Then, A_{obs} is composed with the system automata (A_{sys}) in order to check the corresponding property using reachability analysis techniques. The purpose of the contribution presented in this paper is to automatically generate context automata to be composed with (A_{obs}) and (A_{sys}) to reduce the subsequent state space.

We start our experimentation using an existing use case diagram that highlights different actors involved in the *MHS* detection subsystem. The first phase consisted in extending each use case with the description of interactions occurring between the *MHS* and its context based on provided specification documents and models.

Due to space restrictions, we can not show all modeling artifacts generated at each step of our approach. Instead, we provide, in the next section, a detailed discussion that comments important results and issues over considered case studies.

4.2 Analyzing results

The approach proposed in this paper is complementary to that proposed in [6], which evaluates the use of contexts for model verification through the proposed CDL. In [6], CDL was evaluated through several aeronautic and military industrial case studies. Conclusions of this evaluation is that CDL considerably helps practitioners to formally verify whether designed models meet intended requirements. However, CDL is a *low level* language that requires design details since early development phases to produce precise enough specifications for formal verifications. Herein, we propose a model based framework that automatically derive precise specifications of the system context to ease verifications.

[5]a complex digital signal processing
[6]Implemented in the OBP tool-set

Table 1: Experiment results

	S_CP	*MHS*	*AAC*
Generated CDLs	12	4	5
Nb of Actors	4	6	3
Provable properties	155/188	28/36	17/23
Non provable properties	18/188	8/36	2/23
Non computable properties	15/188	0/36	4/23

To validate our approach, we applied it to three embedded systems applications in the electronic industrial domain, the *S_CP* case study described partially in Section 2, the MHS described above and a Cockpit Announcement and Alarm system (*CAA*). *CAA* is an avionic system responsible for alerting pilots about different events and alarms coming from different applications embedded aboard.

The experiment set-up consisted of verifying formal properties using CDL and the OBP tool-set [6]. The context part of CDL models were generated using the approach described in this paper. Table 1 resumes important metrics on the conducted experiments over considered case studies.

For instance, on the MHS case study, 4 CDL models was generated representing different configurations of the detection part of the submarine sonar. On the 36 requirement initially expressed, 28 were successfully checked on the component executable model of the system. The remaining 8 requirements don't fit into any of our property description patterns. In this case study, the tool set provided decidable results over all the provable requirements (0 occurrence of combinatorial explosion).

In summary, the approach proposed in this paper bridges the gap between user models and formal artifacts required by existing formal verification tools. This is done by automating the way CDL models are constructed. It is important to note that not only did the generated activity diagrams serve for the restriction of the system contexts, but also, in some cases together with *XUCs*, served for engineers to add more details in there specifications. For instance, exceptional situations that have been ignored in the specification documents were specified with their corresponding handlers, which leads to more precise descriptions of contexts and requirements. However, the main benefit of the proposed approach is that it contributes to overcome the combinatorial explosion by allowing partial verification on restricted scenarios specified by the context automata. Properties can now be automatically linked to specific contexts. During experiments, we noticed that approximately 40% of checked properties lead to a combinatorial explosion if checked on the system model without considering specific contexts.

5. DISCUSSION AND RELATED WORK

In section 3, we presented an extension of use cases to describe precise interactions between the system and its contexts and handlers to detected exceptions. Exceptional situations are less common and hence the behavior of the system in such situations is less obvious. Therefore, the proposed *XUC* profile represents a good starting point for the identification of environment actors that might interact with the system during its lifecycle.

The work which is most related to ours is [12]. In [12], authors propose an algorithm that transforms dependability-focused use cases with handlers into activity diagrams. The transformation takes textual use case description as source to produce activity diagram model respecting the use case hierarchy source model. Our approach differs in the sense that we go beyond generating activities corresponding to the use case scenarios and automatically extract formal behavioral models of actors that interact with the system using our synthesis algorithm.

Other related work on the use of use cases to generate behavioral models include [1, 3, 10]. Bastide proposes in [3] an integration of user tasks model (named CTT model) to provide an unambiguous model of the behavior of UML use cases. The metamodel resulting from this integration defines the use case main scenario elements. However, exception definition and handlers was not supported in the proposed metamodel. Authors in [1] propose 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.

The work presented in [10] proposes a model based approach to generate an activity diagram modeling the use case scenario. A functional requirement metamodel is proposed to represent the use case scenario with possible exceptions. The proposed metamodel does not take into account how the system could retrieve a normal state after the occurrence of an exception. On the other hand, the approach proposed in this paper introduces handler use cases to describe how situations that threaten the system safety or reliability are dealt with. For instance, engineers might start with the UML use case (mostly an existing one) then precisely model interactions happening during each system use case with possible exceptional situations and their corresponding handlers. At this point, behavioral models of system context are automatically generated for each identified environment actor.

6. CONCLUSION AND FUTURE WORK

This paper proposes a tool-supported framework for context specification from use cases. The process supports an important activity in context aware formal verifications and complements existing approaches by automating the construction of behavioral models.

Our approach automatically generates environment entity behaviors directly from use cases and scenarios. We have generated an activity diagram that describes 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.

As a future work, we firstly will extend our work to deal with the rest of activity control nodes in order to produce more precise activity diagrams. Secondly, we plan to integrate requirement engineering approaches that aims to link high level user goals with operational properties such the one presented in [2]. We believe the combination of model transformations for the generation of precise context descriptions and goal-oriented requirement engineering approaches for elicitation, elaboration and specification of system requirements can considerably improve the integration of formal methods into software engineering activities.

7. REFERENCES

[1] J. Almendros-Jimenez and L. Iribarne. Describing use cases with activity charts. *Metainformatics*, pages 141–159, 2004.

[2] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning operational requirements from goal models. *ICSE*, pages 265–275, 2009.

[3] R. Bastide. An integration of task and use-case meta-models. *Human-Computer Interaction. New Trends*, Jan 2009.

[4] A. Cockburn. Writing effective use cases. *1st edn. Addison-Wesley Longman*, 2000.

[5] A. Dardenne, A. Van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20:3–50, 1993.

[6] P. Dhaussy, P.-Y. Pillain, S. Creff, A. Raji, Y. L. Traon, and B. Baudry. Evaluating context descriptions and property definition patterns for software formal verification. *Model Driven Engineering Languages and Systems (MoDELS)*, 5795:438–452, Oct 2009.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. *ICSE*, pages 411–420, 1999.

[8] M. B. Dwyer, J. Hatcliff, C. Robby, Pasareanu, and W. Visser. Formal software analysis emerging trends in software model checking. *FOSE*, pages 120–136, 2007.

[9] J. Fitzgerald, P. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. Validateddesigns for object-oriented systems. *Springer-Verlag TELOS, CA, USA*, 2005.

[10] J. J. Gutiérrez, C. Nebut, J. N. Escalona, Mejias, and M. I. Ramos. Visualisation of use cases through automatically generated activity diagrams. *MoDELS Š08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, Berlin, Heidelberg, Springer-Verlag*, 5301:83–96, 2008.

[11] S. Konrad and B. H.C. Cheng. Real-time specification patterns. In *ICSE*, 2005.

[12] S. Mustafiz, J. Kienzle, and H. Vangheluwe. Model transformation of dependability- focused requirements models. *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, 2009.

[13] K. Nakagawa, H.andTaguchi and S. Honiden. Formal specification generator for KAOS: model transformation approach to generate formal specifications from KAOS requirements models. *ASE'07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, ACM*, pages 531–532, 2007.

[14] OMG. UML 2.1.2 superstructure specification. (formal/2007-11-02) edition (2007). 2007.

[15] A. Raji and P. Dhaussy. User context models : A framework to ease software formal verifications. In *12th International Conference on Enterprise Information Systems*, Funchal, Madeira Portugal, 2010. This article is accepted to appear in ICEIS 2010 proceedings.