JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 8, 2007

Early Safety Analysis: from Use Cases to Component-based Software Development

Yunja Choi, School of Electrical Engineering and Computer Science, Kyungpook National University, Korea

We propose an easy-to-use but formal approach for early safety analysis in the context of component-based software development and illustrate its application with a case example. Our approach aims at adopting formal safety analysis while maintaining flexibility and consistency throughout the development process. To this end, we use semi-formal use cases with templates that can be systematically translated into the formal specification language RSML^{-e}, whose execution environment integrates automated verification tools such as the model checker NuSMV. Consistency between use cases and the high-level component design is maintained through a systematic transition, so that the result of the safety analysis can be easily reflected in the design model.

1 INTRODUCTION

As the dependency of our society on control software has increased tremendously, software safety has become an important issue; control software is ubiquitous in our daily lives, from mobile phones to aircraft controllers. Safety-related issues in such control software have to be pre-identified from the beginning of system development so that corresponding prevention mechanisms can be incorporated into the system design [17].

Numerous approaches have been suggested for early safety analysis with varying degrees of automation of the analysis method; the hard-core formal methods community promotes using formal specification languages for safety-critical systems so that formal verification and validation can be thoroughly performed on formal requirements [4, 14, 15]. Some of the approaches try to generate certifiable code directly from the verified specifications [25], bypassing the design stage, in order to ensure that the safety properties verified in the requirements are preserved in the implementation.

In many application areas, where the level of safety assurance often needs to be adjusted in trading off with flexibility and reusability, the use of formal specification may not be a practical approach. Especially in component-based system development, where a flexible design architecture is desirable, the use of a flexible requirements specification language is often necessary for seamless transition from requirements to design. In such cases, informal or semi-formal requirements can be translated into a formal specification for reasoning about the safety aspect of the requirements [10, 16, 18, 24]. Such translations can be difficult to perform in practice when understanding of the target formal specification language requires sophisticated mathematical skills, which has been one of the major barriers against formal methods being a routine part of software engineering processes in practice. Moreover, how to incorporate the analysis result from requirements engineering into the early stage of the design process has not been well understood.

In this work, we suggest an easy-to-use formal approach for safety analysis in the early stage of requirements engineering in the context of component-based software development. Our approach is two-fold; first, the issue of providing safety analysis capability while maintaining flexibility and usability is addressed by using semi-formal use cases with templates [7], which can be systematically translated into a formal specification language RSML^{-e} [26]. Verification of safety requirements is performed on the translated use cases using the model checker NuSMV [21] integrated into the execution environment of RSML^{-e} . Second, the issue of incorporating the result of the safety analysis into the early stage of design is addressed by defining a systematic transition from use cases to the high-level component design based on the KobrA approach [2].

RSML^{-e} is a state machine language designed with an emphasis on readability and writability for non-computer scientists, whose practical value has been proven through industrial applications [20]. Thanks to the simple syntax of the language, we can provide a systematic mapping from use cases to RSML^{-e} . Moreover, its execution environment Nimbus [13] provides automated visual simulation and formal verification capabilities. Once use cases are translated into RSML^{-e} , validation of the use cases using simulation and formal safety analysis with the help of model checking are rather straightforward [5]. The analysis result is incorporated into the system model by providing systematic transition from use cases to high-level component design. We demonstrate our approach on a hypothetical elevator system developed using the component-based software development approach KobrA [2].

The remainder of this paper is organized as follows; Section 2 introduces our semi-formal use cases with templates, which is a basis of our approach. Section 3 suggests guidelines for systematic transition from use cases to the initial component design in the KobrA method. Section 4 illustrates our safety analysis approach, including a short description of RSML^{-e} , a systematic translation from use cases to RSML^{-e} , and the safety analysis process. We conclude with discussion in Section 5.

2 REQUIREMENTS SPECIFICATION USING USE CASES

Use cases [7] are a popular means to specify behavioral requirements of software systems, mainly due to their intuitive style which can be easily understood by engineers as well as non-technical stakeholders. This nature of use cases can promote efficient communication among stakeholders — a major reason why they are preferred to formal specification languages in practice.

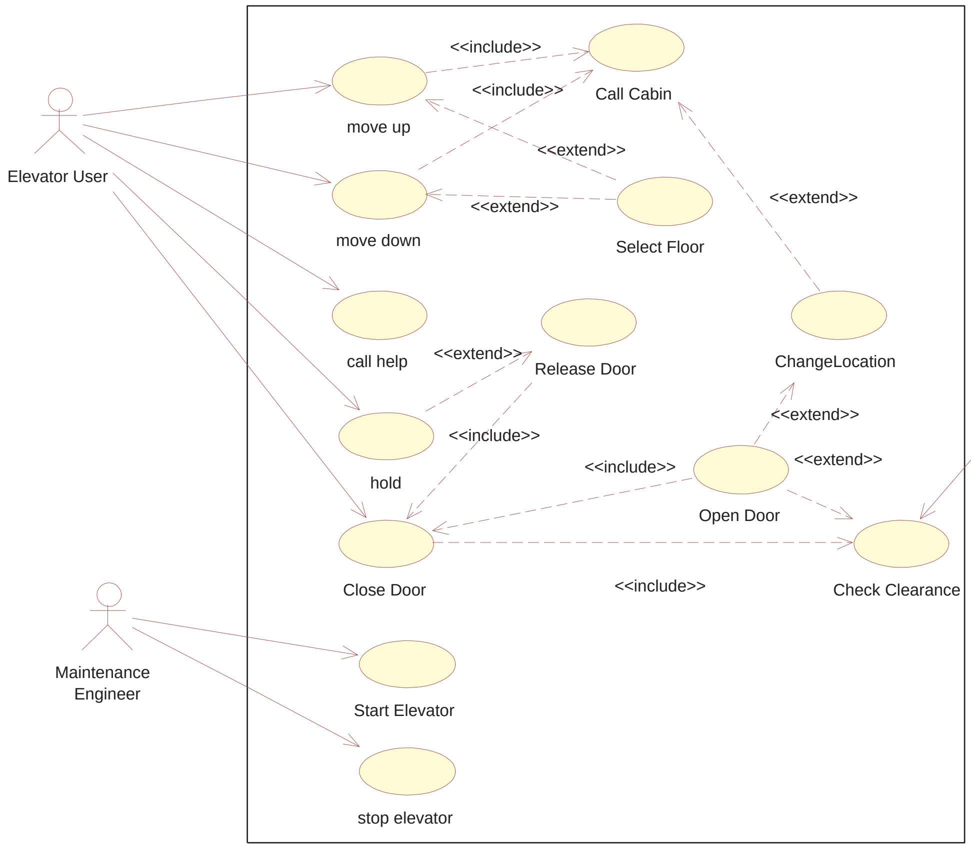

Figure 1: Use cases for an hypothetical elevator system

Use cases consist of three artifacts; use case diagrams, textual descriptions of use cases, and use case scenarios. Use case diagrams illustrate the relationship among use cases and actors, giving an overview of the system behavior.

Figure 1 shows an essential part of the use case diagram specifying the behavior of a hypothetical elevator system. Each actor outside of the box represents an external entity interacting with the system. The use cases inside the box represent system functionalities provided to the external actors, where each use case can include or be extended by other use cases. For example, the use cases *move up* and *move out* are to represent how a user interacts with the elevator system when she/he wants to move up or down floors. For both cases, a user needs to call the cabin by pressing the up or down button outside the elevator. Since this act of calling the cabin is necessary for both *move up* and *move down* use cases, they include the use case *Call Cabin*. On the other hand, the use case *Select Floor*, representing the action of selecting destinations after getting into the elevator cabin, is considered optional for the *move up* and *move down* use cases, and, thus, modeled as extending those use cases. In this early stage of requirements specification, we consider only the abstract notion of elevator without distinguishing the software part from the hardware component of

UseCase	MOVE UP
Actors	Elevator user
Intent	Move from the current floor to another floor above the floor.
Preconditions	THE ELEVATOR IS OPERATING
Flow of events	1) Included use case “call cabin” 2) extending use case “select floor” [Exception]
Exceptio ns	
Rules	Look at included Use Cases
Quality constraints	Look at included Use Cases
Monitored environmental variables	Look at included Use Cases
Controlled environmental variables	Look at included Use Cases
Postconditions	ELEVATOR MOVES TO THE DESTINATION AND OPENS THE DOOR

UseCase	CALL CABIN
Actors	Elevator user
Intent	To make the elevator cabin come to the floor of the user
Preconditions	The elevator is operating
Flow of events	1. Actor indicates his/her intention of moving up or down with the elevator (move up or down) 2. Elevator determines next moving direction based on the floor of request 3. Elevator moves the cabin to the requested floor : extending use case “change location” 4. Cabin opens its door [Exception: technical problem]
Exceptions	
Rules	The elevator stops at every floor where this use case is initiated on the way of moving. Repeated requests are ignored
Quality constraints	Arrival of Cabin and Opening of cabin door
Monitored environmental variables	Floor of request : 1 to N Moving request : Up or Down
Controlled environmental variables	Current position : 1 to N Moving direction : Up, Down, Halt
Postconditions	Current position = floor of request

Figure 2: Textual specification of use cases

the system, which will be identified by component decomposition in the later stage.

The use case diagram is supplemented by textual use cases with a varying degree of formality — from an informal, casual description to the use of a semi-formal template specifying details of each use case. We adopt a semi-formal template for textual use case description in order to support formal safety analysis with some degree of automation. Figure 2 shows the textual use case of *Move Up* with its included use case *Call Cabin*.

The template is designed to specify details of use case behavior, including pre- and post-conditions of the use case, the flow of events, and controlled (output) and/or monitored (input) variables related to the use case. As specified in the flow of events, the *move up* use case first follows the flow of events of the use

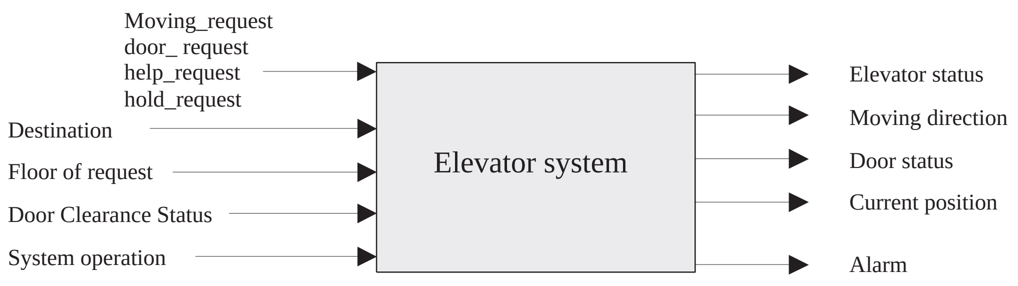

Figure 3: System Environment Model

case *call cabin* and then optionally follows the flow of events of the use case *select floor*, if pre-conditions of those use cases are satisfied. Note that all the rules and constraints of the included use cases also apply to the use case. The monitored and the controlled variables of the use case include the monitored and the controlled variables from its included use cases, respectively. Nevertheless, use case diagrams and use case templates focus on *what* the system does and not on *how* it implements; for example, the monitored variable *Floor of Request* of the use case *Call Cabin* would be measured from the signal initiating the *Call Cabin* and its value would be stored and used for making the elevator cabin stop at the *Floor of Request* on the way of moving, but such details are left to design decision.

We can construct the environment model of the system under development from the monitored and the controlled variables specified in use cases. The environment model views the system as a black box and describes only the input to the system from the external environment and the outcome of the system to the environment. As shown in Figure 3, the input (output) variables of the elevator system are the collection of monitored (controlled) variables specified in the use cases. This environment model, together with the use case diagrams, will be used as a basis to derive the high-level component of the system design.

Use case scenario describes possible usage scenarios of the system and is often specified in sequence diagrams or activity diagrams. In our approach, the use case scenario is considered a part of the context realization, which is a starting point of component design; this will be described in the next section.

3 COMPONENT DESIGN FROM USE CASES

In this section, we describe how use case specifications are used to derive a high level component design in the context of the component-based software development approach KobrA [2].

The KobrA approach

The KobrA approach is a structured and recursive method for component-based system development. Its framework process starts from considering the whole system as consisting of one component, which is an abstract and external view of the system. The description of each component is then split into two main parts: the specification and the realization. The specification describes the externally visible characteristics (contracts) of the component. Each externally visible functionality is realized, possibly with the decomposition of the component, in the realization process through interactions with lower-level sub-components capturing the architecture (or design) of the component. This means that the framework development process can be entirely recursive — a complete system can be a component, and any component can be a system that can be decomposed further.

This recursive nature of KobrA ensures high-quality system development through carefully controlled consistency, traceability, and realization relationships. Nevertheless, it also means high dependency among a component and its sub-components; a problem in the high-level component propagates to the low-level components.

Transition from use cases to the high-level component

In order to provide a consistent safety analysis mechanism in the development process, a seamless transition from requirements to high-level component specifications is a prerequisite. To this end, we suggest a set of guidelines for designing high level KobrA components from use cases.

1. The system under development defines a high-level abstract component class viewing the whole system as one component.
2. Each actor identified in the use case diagram defines a class associated with the component class.
3. Each controlled variable defines an attribute of the component class.
4. Each use case that has a direct interaction with an external actor defines an externally visible operation of the component.
5. Each include/extending/specialized use case without direct interaction with an external actor defines an operation of a sub-component of the component.
6. Each monitored variable for each use case defines a parameter of the corresponding operation.
7. A set of associations between an actor and use cases defines an interface.

Figure 4 shows the first-level structural specification of the elevator system derived from use cases; the main component *ElevatorContext* has 9 externally visible

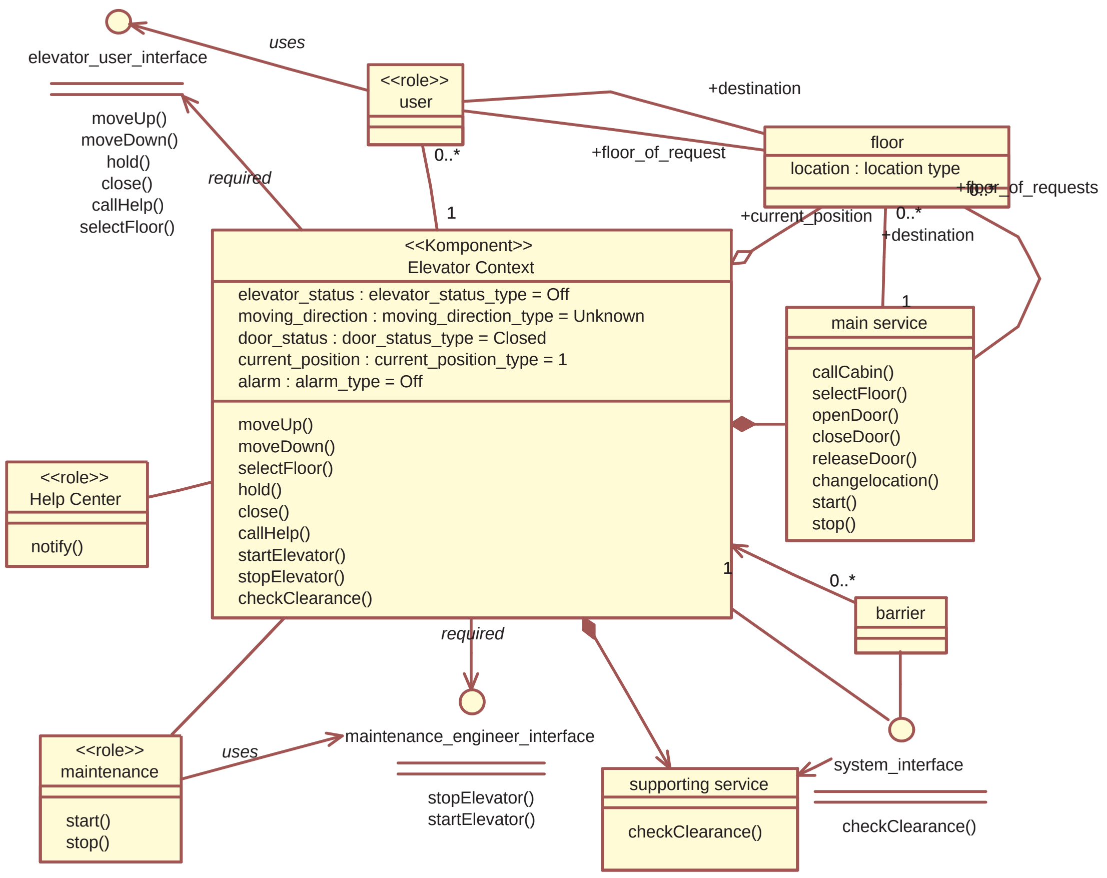

Figure 4: High-level structural diagram

operations derived from use cases directly interacting with external actors, 5 externally visible attributes from controlled variables. A set of operations (use cases) directly interacting with the same actor constitutes one interface specific to the actor; for example, *elevator_user_interface* has 6 operations as its signature. The classes *user, HelpCenter, maintenance,* and *barrier* are identified from actors, and the decomposition of the system component into *main service*, *supporting service*, and *floor* is a design decision to internally realize the externally visible operations. Operations defined in this decomposition are derived from include/extending use cases, and are visible only to the *ElevatorContext*, hidden from external actors of the system. Further decomposition can be performed for each sub-component in the next iteration, if necessary.

The internal behavior (realization) of each externally visible system operation is described using an activity/interaction diagram; Figure 5 shows an example of the realization behavior for the operation *moveUp* of the elevator context. This activity diagram in the highest-level component description can be considered the same as the use case scenario for *Move Up* from which the operation *moveUp* is derived.

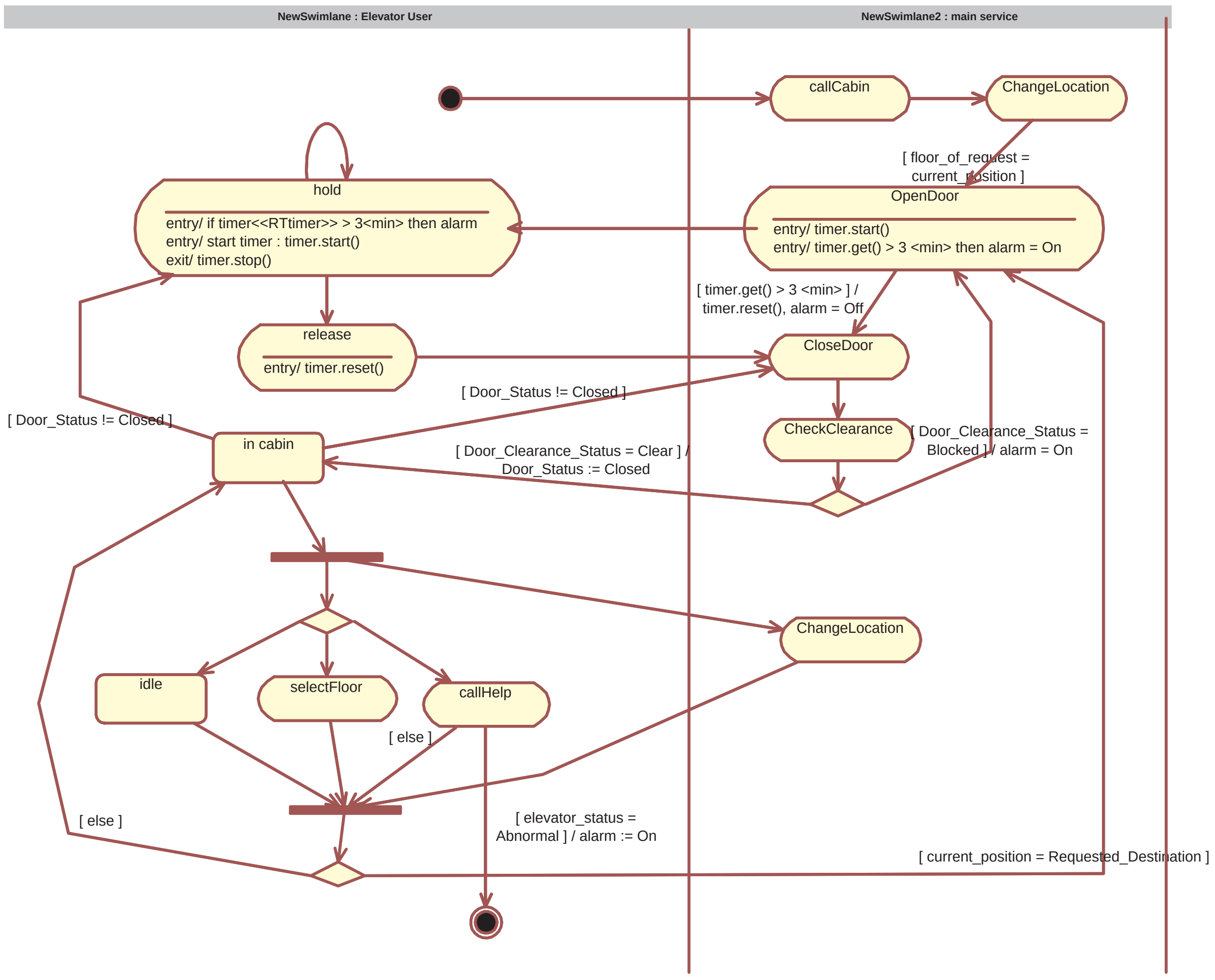
NewSwimlane : Elevator User
NewSwimlane2 : main service
callCabin
ChangeLocation
[floor_of_request = current_position]
OpenDoor
entry/ timer.start()
entry/ timer.get() > 3 <min> then alarm = On
hold
entry/ if timer<<RTtimer>> > 3<min> then alarm
entry/ start timer : timer.start()
exit/ timer.stop()
[timer.get() > 3 <min>] / timer.reset(), alarm = Off
release
entry/ timer.reset()
CloseDoor
[Door_Status != Closed]
[Door_Status != Closed]
in cabin
CheckClearance
[Door_Clearance_Status = Clear] / Door_Status := Closed
[Door_Clearance_Status = Blocked] / alarm = On
ChangeLocation
idle
selectFloor
callHelp
[else]
[else]
[elevator_status = Abnormal] / alarm := On
[current_position = Requested_Destination]

Figure 5: Realization of moveUp use case

Note that we allow non-deterministic behavior in this high-level description; once the elevator user is in the cabin and the door is not closed, either the *hold* action can be taken by the user or the *CloseDoor* action can be taken by the main service.

4 SAFETY ANALYSIS USING MODEL CHECKING

We have described our seamless transition approach from use cases to high-level components. Use cases are converted into the operations of high-level design with their behavior specified in the textual use cases and use case scenarios are manifested into the component realization. Therefore, any unsafe behavior of the use cases will propagate into high-level components, and then, to lower-level components because of the recursive nature of the KobrA approach. In this respect, the identification of safety-related issues in use cases is a must so that we can take the issues under consideration throughout the transition and decomposition process. As we can see later, most unsafe situations result from inter-dependent but under-specified use cases, which makes manual analysis difficult especially when we have a large number of use cases. Therefore, we adopt an automated analysis approach using model checking by systematically translating use cases to the formal specification language RSML^{-e} . This approach enables us to use the simulation and verification tools integrated into the execution environment of RSML^{-e} , facilitating an easy access to formal verification method.

RSML^{-e} and its verification environment

RSML^{-e} (Requirements State Machine Language without events) [26] is a formal data flow specification language semantically similar to Lustre [11] and SCR [3], and visually similar to David Harel's Statecharts [12], supporting for parallelism, hierarchies, and guarded transitions. In addition to that, RSML^{-e} provides rigorous specifications between the environment and the control software, which facilitates the validation of the early specification by execution. Its execution environment Nimbus [13] provides formal verification tools such as the theorem prover PVS [22] and the model checker NuSMV [21] as back-end verifier by automatically translating RSML^{-e} specifications into the input language of the verification tools. The successful use of RSML^{-e} and its execution environment Nimbus in the context of requirements engineering practice is reported elsewhere [20].

RSML^{-e} distinguishes itself from other formal specification languages by emphasizing on readability and understandability by non-computer professionals, enabling a smooth transition from the less formal, but more intuitive use cases to the formal specification.

Figure 6 illustrates the verification framework; use cases are used to derive the initial design of the top level component, as described in Section 3, translated into RSML^{-e} for formal analysis, and verified with respect to safety requirements using the symbolic model checker NuSMV. Here, the safety requirements are identified by a simple version of hazard analysis as we describe later. The result of the verification is reflected in the initial design.

Translation

RSML^{-e} consists of 7 basic constructs: input variables, state variables, input interface, output interface, functions, macros, and constants. Input variables are used to record the values observed in the environment, state variables are organized in a hierarchical fashion and are used to model various states of the control model, interfaces act as communication gateways to the external environment, and the function and macros encapsulate computations providing increased readability and ease of use.

In an abstract view, RSML^{-e} can be considered as a Mealy machine, a finite

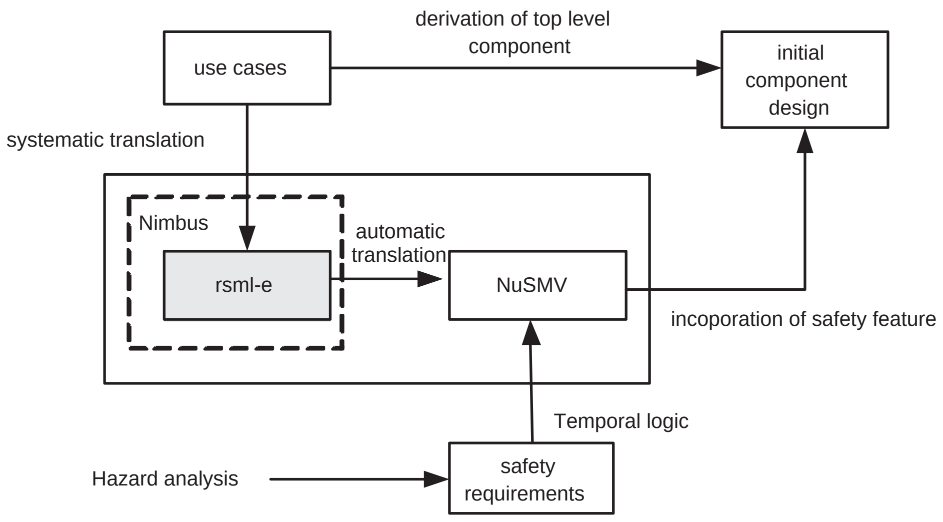

Figure 6: Verification framework

state machine where the outputs are determined by the current state and the input. A Mealy machine is a 6-tuple, $(S, \Sigma, \Gamma, T, G, s)$, consisting of a finite set of states S, a finite set of the input alphabet Σ, a finite set of the output alphabet Γ, a transition function $T : S \times \Sigma \rightarrow S$, output function $G : S \times \Sigma \rightarrow \Gamma$, and a start state $s \in S$. A state in a Mealy machine corresponds to a state vector in RSML^{-e} which is a combination of possible values of all the state variables. Likewise, an input alphabet (output alphabet) corresponds to a vector of input (output) variable values. In RSML^{-e} a transition function is specified in a so-called *AND-OR table*, which is a way of expressing conditions in a disjunctive normal form; each column of truth values represents a conjunction of the propositions in the leftmost column. If a table contains several columns, we take the disjunction of the columns.

The translation from use cases into RSML^{-e} is based on viewing the system as consisting of a set of action states, i.e., each use case constitutes a state in the system with two possible values *ready* and *run*. Initially, all the use cases are in the *ready* state. The transition from *ready* to *run* happens when monitored variable values (input variable values in RSML^{-e}) or other use cases initiate a corresponding use case action. Transition conditions for each state variable values are defined based on the preconditions and the flow of events specified in each use case. Figure 7 shows the mapping between the basic constructs of RSML^{-e} and the constructs of use cases.

For example, the use case *MoveUp* is translated into RSML^{-e} by declaring a state variable *move_up* with the possible values $\{run, ready\}$. The transition between *run* and *ready* is defined based on the flow of events as follows;

```
STATE_VARIABLE move_up : {run, ready}
PARENT: none
```

Mealy machine construct	RSML -e construct	Use case construct
A finite set of state S	State variables	A set of use cases
A finite set of input alphabet	Input variables	Monitored variable
A finite set of output alphabet	Output variables	Controlled variab le
Transition function	AND -OR tables	Pre -condition, flow of control
Output function	Output interface	Post -condition
A start state	Initial value	Undefined/design decision

Figure 7: Basic mapping between use cases and RSML^{-e}

```
INITIAL_VALUE: ready
CLASSIFICATION: State

Transition ready to run IF
TABLE
   /* from use case flow of event */
   @T(Moving_Request = Up)   : T;
   /* from use case pre-condition */
   elevator_status = Normal : T;
END TABLE

Transition run to ready IF
   @T(current_position = destination)
END STATE_VARIABLE
```

In this case, the transition from $call\_cabin = ready$ to $call\_cabin = run$ is initiated by user $Moving\_Request$ on the condition that $elevator\_status$ is normal. This condition is derived from the flow of events, monitored variables, and the precondition of the included use case *call_cabin*. Here, *@T(a)* is a SCR-style notation meaning that "a is true but was not true in the previous step". The end of the use case is specified by a transition from *run* to *ready* triggered by the satisfaction of the post condition $destination = current\_position$. Controlled variables, such as *current_position* and *moving_direction*, are translated into output variables in RSML^{-e} , which is a special kind of a state variable. The transition tables are adopted from the original RSML notation–each column of truth values represents a conjunction of the propositions in the leftmost column (a '*' represents a "don't care" condition). If a table contains several columns, we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form; for example, the transition from *ready* to *run* of the *move_up* state variable happens if $@T(Moving\_Request = Up)$ and $elevator\_status = Normal$, but the same transition of the *call_cabin* (specified below) happens if $@T(move\_up = run)$ or $@T(move\_down = run)$.

The three relationships, *extends, includes, specialization*, which are used to structure use cases, are all flattened out so that each use case is treated as an independent system behavior. The dependency among use cases is specified only by the transition relations. For example, in Figure 1, the *move_up* use case includes the *call_cabin* use case, which is extended by the *change_location* use case. In the translation, all three use cases are defined as independent state variables without any hierarchical relationships among them, but their inter-dependency is specified in the transition relation;

```
STATE_VARIABLE call_cabin : {run, ready}
PARENT: none
INITIAL_VALUE: ready
CLASSIFICATION: State

/*** this part is derived from the "include"
                             dependency ***/
Transition ready to run IF
TABLE
   @T(move_up = run)   : T *;
   @T(move_down = run) : * T;
END TABLE

Transition run to ready IF
   Floor_of_Request = current_position
END STATE_VARIABLE
```

The activation of *call_cabin* use case is initiated by the activation of the including use cases *move_up* or *move_down*. Note that these including use cases have no flow of events other than initiating their included use cases. There can be other cases where the flow of events specified in including use cases may constrain other conditions for the initiation of included use cases. Such constraints can be specified in the corresponding use case scenario.

Figure 8 shows a snapshot of the internal states of the elevator use cases after they are translated into RSML^{-e} ; each grey-colored box represents a state-variable with its possible values enumerated in outlined boxes. Currently active states and their values are visualized with red-colored boxes. The visualization helps engineers understand the current state of the system in an intuitive way by illustrating the hierarchical structure in one view. The execution environment Nimbus enables us to perform visual simulation to validate the behavior of the use cases.

Formal safety analysis

Our safety analysis process starts by identifying and describing possible hazards of the system and then determining the causes of each hazard, using a simple version of hazard analysis [19] or fault tree analysis [8].

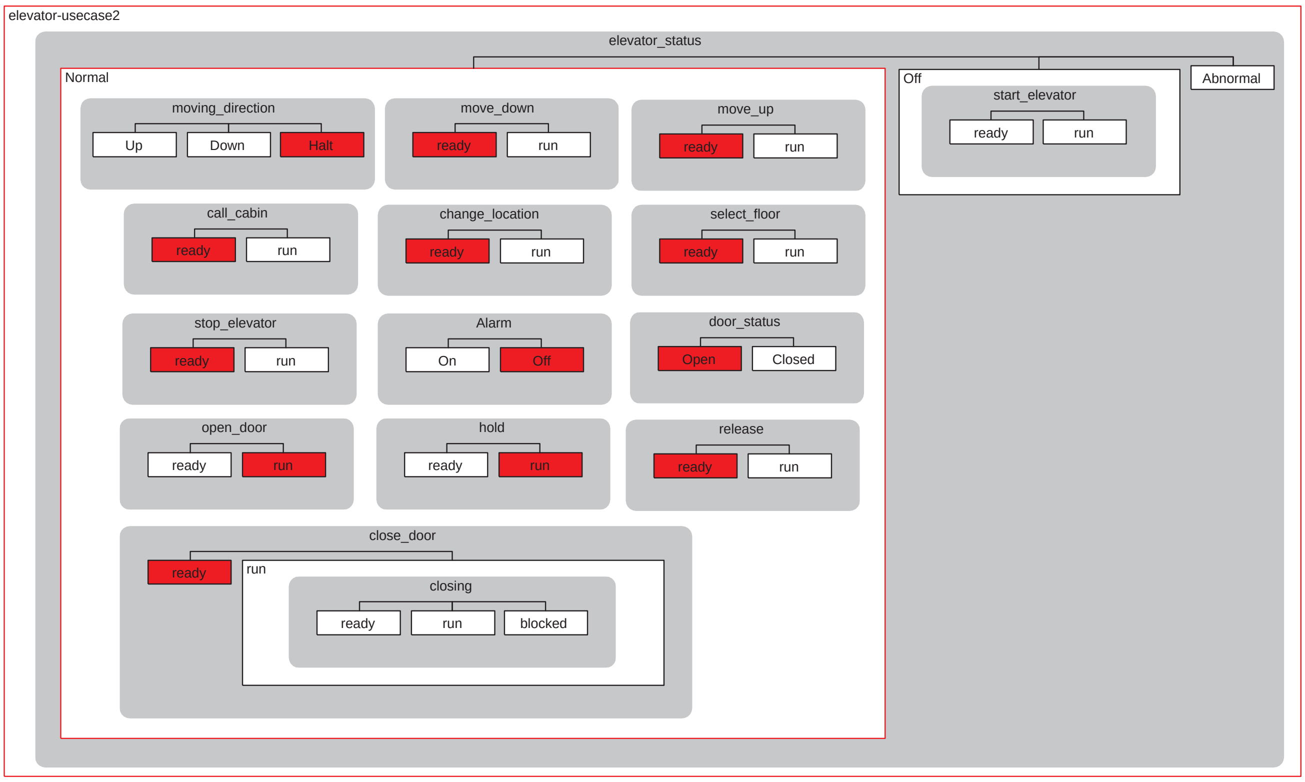

Figure 8: Visualization of use cases

Figure 9 illustrates the safety-related hazard identification with respect to "accidents involving elevator users". All possible causes of the specific hazard are to be identified and each identified cause can be further classified into sub-causes. The hazard analysis process continues iteratively until all the causes of the hazard in question are identified. Given all possible causes of a specific hazard, we categorize them according to the level of abstraction so that high-level causes can be verified on the requirements. In Figure 9, the two causes related to *cabin falls down* are on the hardware and algorithmic level, and, thus, we defer their verification until the system is further decomposed into more concrete levels. On the other hand, the causes related to *user falls out of cabin* or *user falls down into cabin path* are general enough to be checked on the requirements level. These causes are negated and specified in temporal logic to be verified using the automated verification technique model checking [6]. For example, the cause *cabin moves while the door is open* is negated to *cabin does not move if the door is open* and specified in temporal logic as "$safety_1 : AG(Door\_Status = Open \rightarrow change\_location \neq run)$" using the corresponding vocabulary used to translate use cases into RSML^{-e} . By trying to verify the negation of the cause using model checking, we can identify whether and how the cause can actually happen; model checking performs an exhaustive search on the given model to verify the safety property. If the safety property turns out to be true, we conclude that the cause is not possible in the model. Otherwise, it

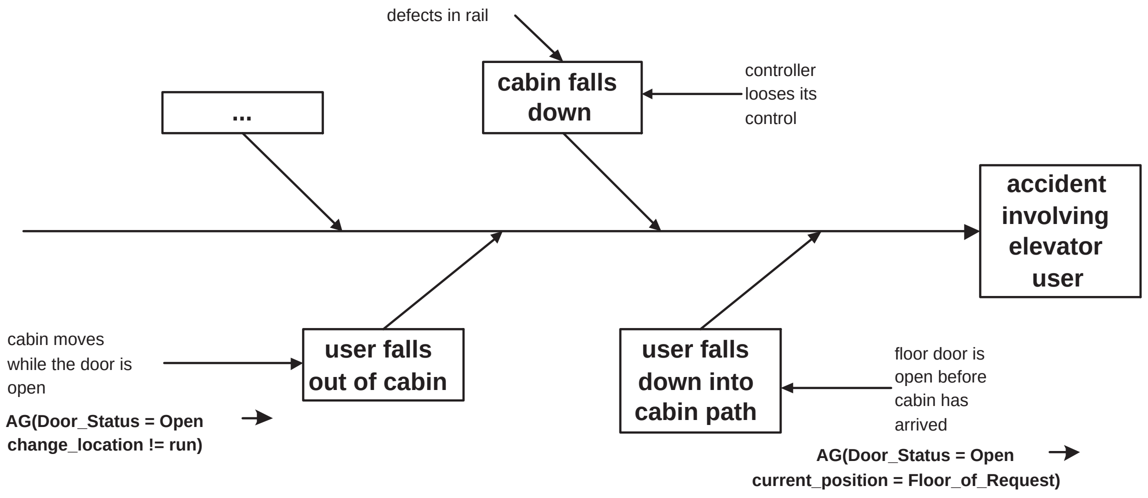
defects in rail
...
cabin falls down
controller looses its control
accident involving elevator user
user falls out of cabin
user falls down into cabin path
cabin moves while the door is open
AG(Door_Status = Open → change_location != run)
floor door is open before cabin has arrived
AG(Door_Status = Open → current_position = Floor_of_Request)

Figure 9: Hazard analysis and temporal logic

generates a counter example trace, which is an execution sequence leading to the unsafe state, i.e., the cause of the hazard. The execution sequence can be used to analyze and correct the model in order to prevent the unsafe situation.

Addressing the analysis result in design

$safety_1$ is verified as false on our use cases using the symbolic model checker NuSMV generating the following *unsafe* execution scenario;

1. Door_Request = Open $\wedge$ door_status = Closed $\wedge$ change_location = ready $\wedge$ close_door = ready $\wedge$ open_door = ready
2. Door_Request = Hold $\wedge$ change_location = run
3. Door_Request = Open $\wedge$ door_status = Open $\wedge$ open_door = run

In the first step, the door is closed and the cabin is not moving. In the second step, the cabin starts moving though *door hold* is requested, since, according to the *open_door* use case, the *door hold* request is ignored when the door is already closed. In the third step, the user requests the door open and the system opens the door even though the *change_location* use case is still in *run* state. This unsafe situation is possible because the *open_door* use case and the *change_location* use case are independently specified regardless of the situation of each other. To avoid such an unsafe situation, we introduce two additional preconditions in the *change_location* use case and the *open_door* use case; the *change_location* use case can be active only when the *open_door* use case is finished, and vice versa. $safety_1$ is verified as true after introducing these pre-conditions to the use cases.

These pre-conditions identified from the use case analysis are also imposed on the design of each component of the system in order to prevent such situations.

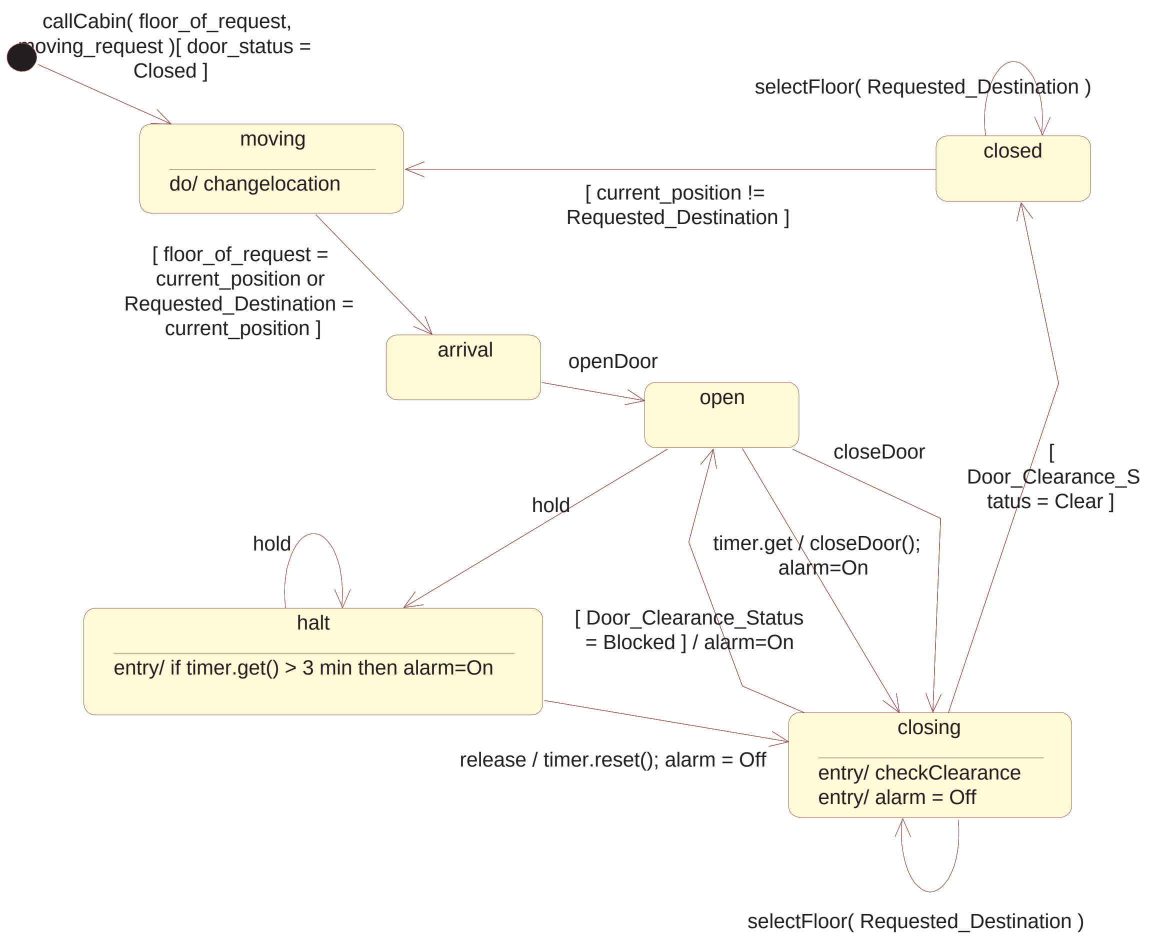
callCabin(floor_of_request, moving_request)[door_status = Closed]
selectFloor(Requested_Destination)
moving
do/ changelocation
closed
[current_position != Requested_Destination]
[floor_of_request = current_position or Requested_Destination = current_position]
arrival
openDoor
open
closeDoor
[Door_Clearance_Status = Clear]
hold
hold
timer.get / closeDoor(); alarm=On
halt
entry/ if timer.get() > 3 min then alarm=On
[Door_Clearance_Status = Blocked] / alarm=On
closing
entry/ checkClearance
entry/ alarm = Off
release / timer.reset(); alarm = Off
selectFloor(Requested_Destination)

Figure 10: Safety consideration in high level design

Figure 10 shows the statechart for the initial component specification for the elevator system. Originally, the initial transition to the *moving* state was specified only by the triggering event *callCabin* without any guarding condition. The guarding condition *door_status* = *Closed* is added after we perform the safety analysis on use cases; all the transition into the *moving* state and the *open* state need to be guarded in a way that moving and opening the door cannot happen at the same time. Though the original statechart specified the *moving* state and the *open* state exclusively, it overlooked the situation that the elevator door might be open in the initial state.

5 DISCUSSION

We have presented our approach for early safety analysis in the context of component-based software development. Our approach addresses two key issues; how to support systematic and easy-to-use analysis methods on semi-formal use cases, and how to incorporate prevention mechanisms into the design stage based on the analysis result. For the first issue, we chose the formal specification language RSML^{-e} for its intuitive syntax and semantics as well as its execution environment enabling automated verification without additional cost. The second issue is addressed by suggesting guidelines for seamless transition from use cases to the initial component design, which will be recursively decomposed in the design process.

Our approach follows the use case driven system development framework with an emphasis on integrating safety analysis throughout the development process. There are other approaches, such as Catalysis [9] and DisCo [1], which follows the use case driven system development approach by formalizing use cases and providing incremental modeling methodology [23]. Nevertheless, these approaches do not address safety issues as an essential part of system development. Moreover, the transition from use cases to initial design is left to design decisions without systematic guidelines.

There are a couple of issues to be addressed to claim the value of our approach; first, as KobrA is a recursive development framework, we may need a recursive safety analysis framework as well that is interwoven with the KobrA framework. Though the work presented here bridges the gap between requirements and initial design with respect to safety issues, the problems related to further decomposition and refinement are not addressed. Second, the practical value of this work has to be evaluated on industrial applications in terms of usability and effectiveness of the approach. We leave these issues to future work.

REFERENCES

[1] The disco home page. http://disco.cs.tut.fi.

[2] Colin Atkinson, Joachim Bayer, and Christian Bunse et al. *Component-based Product Line Engineering with UML*. Addison-Wesley Publishing Company, 2002.

[3] J.M. Atlee and M.A. Buckley. A logic-model semantics for SCR software requirements. In S.J. Zeil, editor, *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'96)*, pages 280–292, January 1996.

[4] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Fracesmary Modugno, David Notkin, and John D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[5] Yunja Choi and Mats Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, pages 109–118, October 2002.

[6] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[7] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Publishing Company, 2000.

[8] U.S. Nuclear Regulatory Commission. Fault tree handbook, NUREG-0492, January 1981.

[9] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: the Catalysis Approach*. Addison-Wesley Publishing Company, 1999.

[10] Wolfgan Grieskamp and Markus Lepper. Using use cases in executable z. In *IEEE International Conference on Formal Engineering Methods*, September 2000.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[13] Mats P.E. Heimdahl, Mike Whalen, and Jeff Thompson. NIMBUS: A tool for specification centered development, September 2003. Presented at the 11th IEEE International Requirements Engineering Conference.

[14] Constance Heitmeyer, James Kirby Jr., Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[15] Farnam Jahanian and Aloysius K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, September 1986.

[16] Woo Jin Lee, Sung Deok Cha, and Yong Rae Kwon. Integration and analysis of use cases using modular petri nets in requirements engineering. *IEEE Transactions on Software Engineering*, 24(12):1115–1130, 1998.

[17] Nancy G. Leveson. Software safety in embedded computer systems. *Communications of the ACM*, 34(2):34–46, 1991.

[18] Nancy G. Leveson and Janice L. Stolzy. Safety analysis using petri nets. *IEEE Transactions on Software Engineering*, 13(3), March 1987.

[19] J.A. McDermid and D. J. Pumfrey. A development of hazard analysis to aid software design. In *COMPASS '94: Proceedings of the Ninth Annual Conference on Computer Assurance*, pages 17–25. IEEE/NIST, June 1994.

[20] Steven P. Miller, Alan C. Tribble, and Mats P.E. Heimdahl. Proving the shalls. In *Formal Methods Europe*, 2003.

[21] NuSMV: A New Symbolic Model Checking. Available at http://nusmv.irst.itc.it/.

[22] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language.* Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, April 1993.

[23] Risto Pitkanen and Petri Selonen. A UML profile for executable and incremental specification-level modeling. In *7th International Conference of the Unified Modeling Language*, 2004.

[24] Wuwei Shen and Shaoying Liu. Formalization, testing and execution of a use case diagram. In *5th International Conference on Formal Engineering Methods*, 2003.

[25] Joachim Thees and Reinhard Gotzhein. The experimental esterel compiler - automatic c generation of implementations from formal specifications. In *Proceedings of FMSP '98, The Second Workshop on Formal Methods In Software Practice.*, 1998.

[26] Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.

ABOUT THE AUTHORS

Yunja Choi is a faculty member at the School of Electrical Engineering and Computer Science at Kyungpook National University in Korea. Her research interests include software verification, requirements analysis, and automated verification of component designs. She can be reached at yuchoi76@knu.ac.kr. See also http://eecs.knu.ac.kr/edu02/eng/prof/d9$_1$1.*htm*.