Functional Requirements Validation by transforming Use Case Models into Abstract State Machines

Patrizia Scandurra
DIIMM, Univ. of Bergamo, Italy
patrizia.scandurra@unibg.it

Tao Yue
Simula Research Laboratory, Oslo, Norway
tao@simula.no

Andrea Arnoldi
DIIMM, Univ. of Bergamo, Italy
andrea.arnoldi@studenti.unibg.it

Marco Dolci
DIIMM, Univ. of Bergamo, Italy
marco.dolci@studenti.unibg.it

ABSTRACT
Use cases are commonly used to structure and document functional requirements while formal methods, such as Abstract State Machines (ASMs), are helpful to specify the behavior of a system and serve to validate system requirements. Therefore, automated support for the transition from use cases to formal models would provide significant, practical help for validating system requirements. This paper proposes the framework AsmetaRE to automatically transform Use Cases Models into ASM executable specifications, and then validate systems requirements through simulation and scenario-based simulation of the generated ASMs with the help of the ASM analysis toolset ASMETA.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications; D.2.2 [Software Engineering]: Design Tools and Techniques

General Terms
Design, Languages

Keywords
Requirements validation, use cases, abstract state machines

1. INTRODUCTION
Developing complex and safety-critical systems requires precise and unambiguous specification of requirements. A formal specification language is thus well suited to this task. Formal specification languages require, but also exacerbate, the need for tools. In particular, tools should support the elaboration (how to build the formal specification from requirements) and the validation (how to check the adequacy of the specification towards the informal needs of the various stakeholders). Use cases, such as the use case notation of the Unified Modeling Language (UML) [14], are commonly used to structure and document functional requirements of a system, while formal methods, such as Abstract State Machines (ASMs) [6], are helpful to specify the behavior of a system and to validate system requirements. Therefore, automated support for the transition from use cases to formal models, such as ASM models, would provide significant, practical help for elaborating and validating system requirements.

This paper proposes an approach to automatically transform use case models of systems requirements into ASM-based executable specifications and then validate these generated formal models through simulation and scenario-based simulation. The latter form of simulation allows automatic model-driven animation [7]: scenarios are automatically generated from the use case model and then animated through simulation. We assume that requirements engineers define use cases by following the approach Restricted Use Case Modeling (RUCM) [16]: an extension of the UML use case notation by a description template and restriction rules for reducing ambiguity and facilitate automated analysis.


This paper focuses on the framework AsmetaRE and on the transformation approach from use cases to ASMs. Section 2 reports some related works. Section 3 summarizes the RUCM and the metamodel UCMeta for use case modeling, and provides an overview of the ASM formal method. The framework is presented in Section 4. Finally, Section 5 concludes the paper and outlines some future directions.

2. RELATED WORK
A systematic literature review on transformations of textual requirements specifications into UML analysis models, including state machine diagrams is in [18]. A series of methods is proposed in [12] (one of the primary studies of the systematic review) to precisely capture requirements and then manually transform requirements into a conceptual model composed of object models (e.g., class diagrams), state machines, sequence diagrams, and functional diagrams. The
The approach in [13], another primary study of the systematic review [18], proposes to generate finite state machines from use cases in restricted Natural Language (NL). The approach requires the existence of a domain model. The domain model serves two purposes: a lexicon for the NL analysis of use cases, and the structural basis of the state transition graphs being generated. One can see that a great deal of user effort is needed to obtain a domain model containing classes, associations, and operations, which are all indispensable for generating state machines.

The work presented in this paper is related to the aToucan [17] tool, since AsmetaRE takes as input instances of a metamodel, named UCMeta, that are generated and used by aToucan as intermediate representation of use cases. aToucan is merely aimed at transforming a use case model produced with RUCM [16] into an UML analysis model made of class and sequence diagrams, activity diagrams and state machine diagrams, while establishing traceability links during transformations. More recently, in [15] an approach, implemented as part of the tool aToucan, is proposed to automatically transform a RUCM model into a high-level (system-level) UML state machine diagram for the purpose of supporting state-based system testing. The automatically generated UML state machine diagrams have to be manually refined such that executable test cases can be generated using existing Model-Based-Testing (MBT) tools. Our goal, however, is different. We aim at obtaining ASM-based executable specifications for carrying out requirements validation through simulation and scenario-based simulation.

Albert II [5] is a formal language for expressing requirements of distributed real-time systems. It is founded on a complex real-time temporal logic that supports the reasoning process of the analyst. The ASM formal method is, instead, lightweight, operational, and without mathematical overkill for the particular end of validating requirements of general-purpose applications.

There is a previous work [11] aiming at generating ASMs from NL requirements. Though no longer maintained, some of our transformation patterns were inspired by it.

### Figure 1: Use case Withdraw Fund in RUCM

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1) The system displays an apology message MEANWHILE the system ejects the ATM card. 2) The system shuts down. 3) ABORT. Postcondition: ATM customer funds have not been withdrawn. The system is shut down.</td>
<td>IF ATM customer enters Cancel THEN 1) The system cancels the transaction MEANWHILE the system ejects the ATM card. 2) ABORT. ENDIF Postcondition: ATM customer funds have not been withdrawn. The system is idle. The system is displaying a Welcome message.</td>
<td>1) The system displays an apology message MEANWHILE the system ejects the ATM card. 2) The system shuts down. 3) ABORT. Postcondition: ATM customer funds have not been withdrawn. The system is shut down.</td>
<td>1) The system displays an apology message MEANWHILE the system ejects the ATM card. 2) The system shuts down. 3) ABORT. Postcondition: ATM customer funds have not been withdrawn. The system is shut down.</td>
</tr>
</tbody>
</table>

3. BACKGROUND ON RUCM AND ASMS

**RUCM and UCMeta.** RUCM [16] is a use case modeling approach that extends the UML use case diagram by proposing a use case template and 26 restriction rules for reducing ambiguity and easing automated analysis. UCMeta is a metamodel used by aToucan [17] to create machine-processable use case models conforming to the RUCM approach.

An example of RUCM use case description related to the withdrawal of an ATM system is shown in Fig. 1. A use case description has one basic flow (the happy flow) and can have one or more alternative flows (see first column in Fig. 1). Alternative flows are classified into three types: a specific alternative flow refers to a specific step in the reference flow; a bounded alternative flow refers to more than one step in the reference flow – consecutive steps or not; a global alternative flow refers to any step in the reference flow – consecutive steps or not; a global alternative flow refers to any step in the reference flow.

Each simple sentence subject-predicate in an RUCM use case description can either denote a condition or an action. RUCM defines also a set of keywords (complex sentences) to specify control structures: conditional logic sentences (IF-THENELSE-ELSEIF-ENDIF), concurrency sentences (MEANWHILE), condition checking sentences (VALIDATES THAT), and iteration sentences (DO-UNTIL).
keywords greatly facilitate the automated generation of ASM rule schemes as they clearly indicate when alternative flows start and which kind of alternative flow starts. Other keywords, special sentences, can be used to specify how flows in a use case or between use cases relate to one another. An alternative flow ends either with ABORT or RESUME STEP, which means that the last step of the alternative flow should clearly specify whether the flow returns back to the reference flow and where (using keywords RESUME STEP followed by a returning step number) or terminates (using keyword ABORT). Moreover, the keyword INCLUDE USE CASE denotes the invocation of a other use cases.

ASMs. The ASM formal method is an extension of the Finite State Machine formalism [6] where unstructured control states are replaced by states comprising arbitrary complex data. The states of an ASM are multi-sorted first-order structures, i.e. domains of objects with functions and predicates (boolean functions) defined on them. The transition relation is specified by rules describing how functions change from one state to the next. There is a limited but powerful set of ASM rule constructors. The basic one has the form of guarded update “if Cond then Updates” where Updates is a set of function updates \( f(t_1, \ldots, t_n) := t \) that are simultaneously executed\(^1\) when Cond is true.

Functions changing during computation are dynamic and they are further classified as: monitored (only read, as events provided by the environment), controlled (read and written), shared (read and written by the machine and by the environment) and out (only written) functions.

There is a limited but powerful set of ASM rule constructors that allow to express simultaneous parallel actions (par), either in an atomic way, Basic ASMs, or in a structured and recursive way, Structured or Turbo ASMs, by sequential actions (seq), iterations (iterate, while, rec-while), and submachine invocations returning values. Appropriate rule constructors also allow non-determinism (existential quantification choose) and unrestricted synchronous parallelism (universal quantification forall).

4. THE ASMETA RE FRAMEWORK

AsmetaRE is built upon the Eclipse environment and the ASM specification and analysis toolset ASMETA [3]. Its essential components are shown in Fig. 2 together with the phases (denoted in the figure with a number) the stakeholders may undertake to perform requirements validation. First (phase 1), the framework receives in input a use case model UCMod (as instance of the UCMeta metamodel) generated by the aToucan tool [17] and expressing the stakeholders’ functional requirements. Second (phase 2), the main concern is to transform the input UCMod into an ASM under the form of a textual specification in AsmetaL\(^2\). The result of such model-to-text transformation is an ASM executable formal specification that serves as basis to perform requirements validation through the ASM analysis toolset ASMETA. Basically, two validation techniques can be performed with the help of two tools of the ASMETA toolset:

\(^1f\) is an arbitrary \( n\)-ary function and \( t_1, \ldots, t_n, t \) are first-order terms. To fire this rule in a state \( S_i, i \geq 0 \), evaluate all terms \( t_1, \ldots, t_n, t \) at \( S_i \) and update the function \( f \) to \( t \) on parameters \( t_1, \ldots, t_n \). This produces another state \( S_{i+1} \) that differs from \( S_i \) only in the new interpretation of \( f \).

\(^2\)AsmetaL is a textual notation, human comprehensible, and conforming to the ASMETA/AsmM metamodel for ASMs.

4.1 From Use Cases to ASMs

Below, an ASM encoding of RUCM used to construct an ASM model of the system behavior from the UCMod follows.

The generated ASM resembles the system and reacts to the input events of the use case actors (the ASM surrounding environment). These events include pre-/post- conditions and conditions appearing in the condition checking and conditional sentences. These input events are represented by ASM monitored predicates. In order to start an actor-system interaction for the execution of a single use case, a predefined monitored function useCaseChosen is also externally set to the use case to execute.

The execution of atomic system actions (simple sentences in RUCM) by the ASM (the system) is carried out by setting named ASM controlled predicates to true. Moreover, every generated ASM for a considered system maintains and changes its state (either physical state or state of knowledge about the external world) according to a meta-behavior that is described in Fig. 3 in terms of an FSM. The system may be in a state ranging in the set \( \{ \text{init, ready, executing, cancelled} \} \).
The initial state of the system is \textit{init} and becomes \textit{ready} after initializing the controlled portion (functions and domains) of the ASM’s signature for every use case. Upon a use case execution is required (i.e. the value of the monitored function \textit{useCaseChosen} is externally set), it becomes \textit{executing} and the pre-conditions of the chosen use case (the current use case) are evaluated. If they are not satisfied the system state becomes immediately \textit{terminated} and the overall system execution terminates, otherwise, the system proceeds normally by executing an ASM rule corresponding to the basic flow’s steps of the chosen use case (see details on the transformation rules below). At the end of the execution of the current use case, its post-conditions are evaluated and if satisfied the system terminates with success and returns back to the state \textit{ready} to execute a new use case; otherwise, the system state becomes \textit{terminated}. During the execution of a use case, if the system jumps to the execution of an alternative flow, specific or bounded, its state may become \textit{aborted} if it executes an ABORT command. In the state \textit{aborted}, the system may come back to \textit{ready} or become \textit{terminated} depending if the post-conditions of the alternative flow are satisfied or not, respectively. Still during execution, if the system jumps to a global alternative flow, its state becomes temporarily canceled and then \textit{executing} again.

Predefined ASM transition rules specifying the semantics of the RUCM commands (complex and special UCMeta sentences) are imported as model library in each generated ASM model. These rules are the result of a prior formalization in ASM of the behavioral semantics of RUCM keywords to specify concurrent steps (MEANWHILE), condition checking steps (VALIDATES THAT), alternative flow ends with ABORT or RESUME STEP, etc. The formal definition of such predefined rules is out of the scope of this paper. In the sequel, however, the definition of the ASM rules for the VALIDATE THAT and ABORT commands, respectively, is reported as example. The rule \textit{r\_validate\_that} checks if the condition \textit{cond} passed as argument is true or not; if false the flow \textit{f} passed as argument is activated (i.e. it becomes the new flow to execute) by invoking the predefined macro rule \textit{r\_alternative\_flow}, otherwise the execution continues with the next step of the current flow. The rule \textit{r\_abort} simply changes the current state of the system execution to the value ABORTED.

\begin{verbatim}
rule r_validate_that($cond in Boolean,$f in Flow) =
  if not ($cond) then r_alternative_flow[$f] endif

rule r_abort = currState := ABORTED

The remaining functionality that depends on the specific system as described by its use case descriptions is implemented through ASM rules that are generated as part of the transformation from an UCMod to an ASM. This transformation involves the mapping rules summarized in Figures 4, 5, and 6 for use case flows, sentences, and post-conditions concepts, respectively. Basically, an ASM module named \textit{UseCaseName} is generated for each use case, and imported in the main ASM. This ASM module contains the signature and rule definitions capturing the behavior of the system in correspondence of the flows steps and type, type of sentences, and post-conditions of the use case.

4.2 From Use Cases to Avalla scenarios

For each use case, two types of Avalla scenarios are generated: one type to assert the correct system execution from the use case pre-conditions to its post-condition (happy scenarios), and a second type (alternative scenarios) for the execution of alternative flows.

For the lack of the space, we do not detail the rationale behind the generation of such scenarios. We limit to report below an excerpt of the happy scenario for the use case \textit{Withdraw Fund} (see Fig. 1) of the ATM system. First the scenario checks (the Avalla command \textit{check}) the system is in the initial state. Then it enforces one machine step (the command \textit{step}) and checks the system is ready. After that, the use case to execute (\textit{useCaseChosen}) is set (the \textit{set}) to \textit{Withdraw Fund}, the monitored predicates corresponding to the pre-conditions are set to true, the activation condition of the global alternative flow is set to false, and one step is enforced. At this point, the scenario follows the basic flow of the use case and checks that the ASM effectively exhibits the expected behavior by checking the functions values in the tuple (\textit{currState}, \textit{currUseCase}, \textit{currFlow}, \textit{currStep}) representing the control state of the ASM.

scenario Withdraw_Fund
load ATM System/ATM System.asm
check currState=INIT and currStep=1 and previousFlowStep=1;
step
check currState = READY;
set useCaseChosen := WITHDRAWFUND;
set the system is displaying a welcome message := true;
set atm customer enters cancel := false;
step
check currFlow=VALIDATEPIN and currState= EXECUTING and currStep=1 and currUseCase= VALIDATEPIN;
set atm customer enters cancel := false;
step
check currFlow=VALIDATEPIN_BASIC and currState= EXECUTING and currStep=2 and currUseCase= VALIDATEPIN;
set the system recognizes the atm card := true;
set atm customer enters cancel := false;
step
...

5. CONCLUSION AND FUTURE WORK

This paper presented a framework that combines the approach RUCM/UCMeta for use case modeling and the ASM
Table: RUCM/UCMeta

<table>
<thead>
<tr>
<th>Simple Sentence</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Sentence Content</td>
<td>A controlled ASM predicate named SimpleSentenceContent, and an ASM rule named r_simpleSentenceName that sets to true the predicate:</td>
</tr>
<tr>
<td>rule r_simpleSentenceName = SimpleSentenceContent; enddef</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition Check Sentence: VALIDATES THAT cond</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule r_validates_thats(String)</td>
<td>An invocation of the predefined ASM rule r_validates_thats(String) in Flow:</td>
</tr>
<tr>
<td>rule r_validates_thats(String) = rule validate_thats(String)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Include Sentence: INCLUDE UseCaseName</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule r_include_use_case(String)</td>
<td>An invocation of the predefined ASM rule r_include_use_case(String) in UseCase:</td>
</tr>
<tr>
<td>rule r_include_use_case(String) = include_use_case(String)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abort Sentence: ABORT</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule r_abort(String)</td>
<td>An invocation of the predefined rule r_abort(String) in Flow:</td>
</tr>
<tr>
<td>rule r_abort(String) = abort(String)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resume Step Sentence: RESUME STEP n</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule r_resume_step(String)</td>
<td>An invocation of the predefined ASM rule r_resume_step(String) in Flow:</td>
</tr>
<tr>
<td>rule r_resume_step(String) = resume_step(String)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conditional Sentence IF-THEN-ELSE-ELSEIF-ENDIF cond THENActions ELSEActions</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule r_conditional(String)</td>
<td>An invocation of the predefined ASM rule r_conditional(String) in Flow:</td>
</tr>
<tr>
<td>rule r_conditional(String) = conditional(String)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel Sentence MEANWHILE</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule r_flows_step_par(String)</td>
<td>An ASM rule r_flows_step_par(String) that in parallel (by exploiting the ASM par rule constructor) sets to true all the predicates corresponding to the concurrent sentences (simple sentences):</td>
</tr>
<tr>
<td>rule r_flows_step_par(String) = rule flows_step_par(String)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iterative Sentence DO-UNTIL DOActions cond</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule r_flows_iter(String)</td>
<td>An ASM rule r_flows_iter(String) that (by exploiting the ASM while-do rule) executes iteratively the ASM rules corresponding to the iterative sentences DOActions while the condition cond holds:</td>
</tr>
<tr>
<td>rule r_flows_iter(String) = while-do(String)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Transforming use cases and flows

Figure 5: Transforming sentences
formal support to specify and validate the functional requirements of a system. Though the restriction rules of the approach RUCM are based on a clear rationale, users might find them too restrictive or impractical in certain situations. The recent experimental results presented in [19] show, however, that the restriction rules are overall easy to apply and that RUCM results in significant improvements over traditional approaches (i.e., with standard templates, without restrictions) in terms of correctness, completeness, and understandability of use case specifications.

Often use cases, though accurate (according to the RUCM), may be intentionally incomplete, especially when dealing with large systems. ASMs and its refinement mechanism [6] cope with incomplete requirements too. ASM specifications often consist of a series of ASM models, starting with an abstract ground model and proceeding to greater levels of detail in successive refinements.

The approach and the tool AsmetaRE were validated through various case studies (see [2]), such as the ATM system, an Invoice Order system, a Sluice Gate Control system, and an Elevator system, to name a few. More controlled experiments on large case studies are postponed as future work. We also want to extend the framework to allow the use of additional RUCM sentences.

<table>
<thead>
<tr>
<th>RUCM/UCMeta</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Flow of an included Use Case</strong></td>
<td>An ASM rule named ( r_{\text{UseCaseName}_n\text{-postcondition}} ) that checks if the monitored predicates corresponding to the post-conditions are satisfied or not and then updates the system state accordingly. In addition, if the post-conditions are satisfied, the execution of the including use case is restored.</td>
</tr>
<tr>
<td><strong>rule</strong> ( r_{\text{UseCaseName}_n\text{-postconditions}} = )</td>
<td>if (post-condition1 AND post-condition2 AND ...)</td>
</tr>
<tr>
<td><strong>then</strong> par</td>
<td>curUseCase := referenceUseCase</td>
</tr>
<tr>
<td><strong>endpar</strong></td>
<td></td>
</tr>
<tr>
<td><strong>else</strong> curState := TERMINATED</td>
<td><strong>endif</strong></td>
</tr>
<tr>
<td><strong>Basic Flow or Specific Alt. Flow or Bounded Alt. Flow or Global Alt. Flow</strong></td>
<td>An ASM rule ( r_{\text{UseCaseName}<em>n\text{-postcondition}} ) or ( r</em>{\text{UseCaseName}_n\text{-AltFlowType}_n\text{-postcondition}} ) (depending if the flow is basic or alternative) that checks if the monitored predicates corresponding to the post-conditions are satisfied or not and then updates the system state accordingly.</td>
</tr>
</tbody>
</table>

**Figure 6: Transforming post-conditions**

6. REFERENCES