A Heuristic-Based Approach to Refactor Crosscutting Behaviors in UML State Machines

Muhammad Uzair Khan[1], Muhamamd Zohaib Iqbal[1 2]
[1]Software Quality Engineering and Testing Lab (QUEST), National University of Computer and Emerging Sciences, FAST NU, Islamabad, Pakistan
[2] SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
{uzair.khan, zohaib.iqbal}@nu.edu.pk

Shaukat Ali
Simula Research Laboratory, P.O.Box 134, Lysaker, Norway
shaukat@simula.no

***Abstract*— UML state machines are commonly used to model the state-based behavior of communication and control systems to support various activities such as test cases and code generation. Standard UML state machines are well suited to model functional behavior; however extra-functional behavior such as robustness and security can also be directly modeled on them, but this often results in cluttered models since extra-functional behaviors are often crosscutting. Such modeling crosscutting behavior directly on UML state machines is a common practice. Aspect-Oriented Modeling (AOM) allows systematically modeling of crosscutting behavior and has shown to provide a scalable solution in the recent years. However, due to lack of familiarity of AOM in both academic and industry, extra-functional behavior is often modeled directly on UML state machines and as a result those UML state machines are difficult to read and maintain. To improve the readability of already developed UML state machines and ease maintenance, we propose a set of heuristics, derived from two industrial cases studies, implemented in a tool to automatically identify commonly observed crosscutting behaviors in UML state machines and refactor them as Aspect State Machines. Such refactoring makes the state machines easier to maintain and comprehend. We present the results of applying our proposed heuristics to the existing UML state machines of two industrial case studies developed for model-based testing.**

Keywords— Model Refactoring, UML State machine, Aspect-Oriented Modeling, Heuristics

I. INTRODUCTION

UML state machines [1] provide a standardized way of modeling functional behavior of state-based systems to support a variety of activities including testing [2-5], code generation [6], and domain specific modeling [7, 8]. UML state machines provide advanced features such as hierarchy and concurrency to model the behavior of complex industrial systems. However, when extra-functional behavior (e.g., robustness, logging) is modeled with UML state machines, the resulting state machines are cluttered with redundant modeling elements, because the extra-functional behavior is often crosscutting. Such cluttering significantly reduces readability, understandability, and results in state machines that are very difficult to maintain [9].

This work was supported by ICT R&D Fund, Pakistan under the project ICTRDF/MBTToolset/2013. Muhammad Zohaib Iqbal was partly supported by National Research Fund, Luxembourg (FNR/P10/03).

In this paper, we present our work on automatically refactoring crosscutting behavior from existing UML state machines into aspect-oriented state machines. The work presented here is motivated from our previous work on modeling and testing of two industrial case studies; a Videoconferencing System (VCS) called Saturn developed by Cisco Systems [2, 10] and Automated Bottle Recycling System (ABRS) developed by Tomra AS, Norway [6] . Both case studies posed challenges of modeling complex systems with many crosscutting behaviors.

For the VCS, we developed a robustness modeling methodology to model robustness behavior that is typically crosscutting behavior [9]. Such crosscutting behavior was modeled using a slight extension of UML state machines through a profile called AspectSM as Aspect State Machines (ASMs). With ASMs, a modeler can model crosscutting behavior separately from the base state machines and thus reducing the cluttering and improving readability. For the ABRS system, we modeled several crosscutting behaviors, although, AspectSM wasn't used.

In this paper, we present a set of heuristics to automatically refactor crosscutting behaviors from existing UML state machines as ASMs. Even though, we refactor crosscutting behaviors as ASMs, other approaches to model crosscutting behaviors and standard UML state machines using advanced features such as hierarchy can be used in place of ASMs.

The main contributions of this paper are as follows: 1) We define a set of heuristics (derived from the industrial case studies) for identifying crosscutting patterns in standard UML state machines; 2) We provide rules for refactoring the state machine based on the patterns identified by the heuristics into ASMs and base state machine; 3) We provide tool support implementing heuristics and rules for refactoring; 4) We present the results of the application to two industrial case studies.

The rest of the paper is organized as follows; Section II briefly presents the two industrial case studies. Section III presents our heuristics and the proposed refactoring rules. Section IV briefly discusses our tool called *ReState*. The results of applying our approach on the two industrial case studies are presented in Section V. Section VI concludes the paper.

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.94

II. CASE STUDY

Below, we present briefly the industrial case studies that required modeling of crosscutting behaviors.

A. Video Conferencing System

The first case study is related to a project aiming at supporting automated, model-based robustness testing of a core subsystem of a video conference system (VCS) called Saturn developed by Cisco Systems, Norway as reported in [9]. The core functionality of Saturn manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels and there is also a possibility of transmitting presentations in parallel with audio and video. One participant can send presentations at a time and rest of the participants receive it. Saturn consists of 20 subsystems and each subsystem can work in parallel to the core behavior.

In [4], we modeled five crosscutting robustness behaviors of Saturn as ASMs that are: 1) Audio Quality, 2) Video quality, 3) Media quality recovery, 4) Network communication, 5) Crosscutting in guards conditions providing inputs to the VCS. To assess our approach, we obtained woven models of these crosscutting behaviors and used them to assess if our tool can identify and refactor crosscutting behaviors.

B. Automated Bottle Recycling System (ABRS)

The second case study was the industrial Automated Bottle Recycling System developed by Tomra AS, Norway. In our previous work, we published the case study as part of environment modeling approach [6, 11]. The bottle recycling system is fitted with a sorting arm that is used for separating three types of the recyclables, plastic bottles, cans and glass bottles. For testing of ABRS, we proposed the environmental modeling approach and developed state machines that captured the environment of the system, along with the possible error states. While the approach was successfully in testing the ABRS, it produced state machines that contained a number of cross cutting concerns.

In this paper, from the published case study in [6] we have taken four of our state machines of the various components developed for environmental modeling approach and have applied the refactoring. One of the state machines contained cross cutting behavior, modeled as the error state. Originally the error state was introduced in [6] to model the environmental errors. However, the error states were modeled as part of state machines, thus mixing core functionalities/behaviors with erroneous behaviors. The other state machines did not contain any such behavior.

III. HEURISTICS BASED APPROACH FOR AUTOMATED REFACTORING OF STATE MACHINES

Our proposed approach relies on heuristics to refactor existing UML state machines. The approach has two phases: i) crosscutting identification phase based on heuristics; ii) Extraction and refactoring of the identified crosscutting behaviors as one or more ASMs. The heuristics for identifying the crosscutting behavior are formalized in Object Constraint Language (OCL) [12] in the following section.

A. Heurisctis for Identifying Crosscutting Behaviors

We propose four heuristics as shown in Table 1 together with refactoring rules for generating ASMs.

TABLE I. HEURISTICS FOR IDENTIFYING CROSSCUTTING BEHAVIORS

ID	Heuristic for identifying crosscutting behaviors	Ref
1a.	**Description:** Similar incoming transition with same trigger, guard condition and effects. **Rule:** context UML::Vertex inv: self.incoming -> select (t1, t2: Transition | t1. name <> t2.name and t1.source <> t2.source and t1.trigger = t2.trigger and t1.effect = t2.effect and t1.guard.specification.body = t2.guard.specification.body and t1 <> t2)->size () >=1	Rule 1
1b	**Description:** Similar incoming transition with same trigger and effects but different guard conditions. **Rule:** context UML::Vertex inv: self.incoming -> select (t1, t2: Transition | t1. name <> t2.name and t1.source <> t2.source and t1.trigger = t2.trigger and t1.effect = t2.effect and t1.guard.specification.body <> t2.guard.specification.body) ->size() >=1	Rule 2
1c	**Description:** Similar incoming transitions with same trigger and guard condition but different effects. **Rule:** context UML::Vertex inv: self.incoming -> select (t1, t2: Transition | t1. name <> t2.name and t1.source <> t2.source and t1.trigger = t2.trigger and t1.effect<> t2.effect and t1.guard.specification.body = t2.guard.specification.body) ->size() >=1	Rule 3
2a	**Description:** Common trigger across different transitions (more than one triggers). **Rule:** context UML::Region inv: self.transition-> select (t1, t2 :Transition | t1.trigger-> intersection (t2.trigger) >=1	Rule 4
2b	**Description:** Common effect across different transitions (more than one action in body of effect). **Rule:** context UML::Region inv: self.transition-> select (t1, t2 :Transition | t1.effect-> intersection(t2.effect) >=1	Rule 5
2c	**Description:** Common constraint across different guard conditions (composite guard conditions). **Rule:** context UML::Region inv: self.transition-> select (t1, t2 :Transition | isSubString(t1.guard.specification.body, t2.guard.specification.body)>=1 * isSubString(,) takes two guard conditions as string and returns true if second guard is a substring of the first guard.	Rule 6
3a	**Description:** Similar outgoing transitions from a given vertex with same trigger and effects but different guard condition. **Rule:** context UML::Vertex inv: self.outgoing -> select (t1, t2: Transition | t1. Name <> t2.name and t1.trigger = t2.trigger and t1.effect = t2.effect and t1.guard.specification.body <> t2.guard.specification.body) ->size() >=1	Rule 7
3b	**Description:** Similar outgoing transitions with different trigger and guard conditions but same effect. **Rule:** context UML::Vertex inv: self.outgoing -> select (t1, t2: Transition | t1. name <> t2.name and t1.trigger = t2.trigger and t1.effect = t2.effect and t1.guard.specification.body <> t2.guard.specification.body) ->size() >=1	Rule 8
4	**Description:** Common constraints in State invariants. **Rule:**	Rule 9

	context UML::Region inv: self.vertex -> select (v1, v2: Vertex| isSubConstraint(v1.constraint, v2.constraint) ->size() >=1 * isSubConstraint(,) takes two constraints and returns true if second constraint is contained in first.	

For example, the first heuristic (1a) states that we can identify a crosscutting behavior when a given state in a state machine has more than one common incoming transition. Consider the example in Fig. 1, where *T1, T2* and *T3* are similar transitions. Once the crosscutting behaviors are identified, we apply the corresponding refactoring rules. Each heuristic is mapped to a different refactoring rule, allowing us to tackle various sub-variations of the heuristics. The next section explains one of the heuristics and refactoring rules applied when a crosscutting concern is successfully identified. The other rules are implemented in a similar manner.

B. *Refactoring rules for identifying crosscutting behaviors*

We have devised refactoring rules to deal with different types of crosscutting behaviors identified by the heuristics presented in the previous section. We present the heuristic 1a here along with the rule applied once a match is found.

Heuristic 1a. An example is shown in Fig. 1, where all the three incoming transitions on state *S* have the same trigger *Trigger*, the guard condition *Guard* and the effect *Effect*. And is identified as a possible crosscutting behavior.

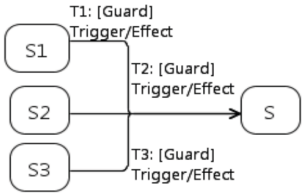

Fig. 1. State machine with similar incoming transitions

Refactoring Rule 1. For crosscutting behavior identified by heuristic 1a, we propose to generate the aspect state machine as shown in Fig. 2.

We remove the *S* state from the base state machine along with the similar transitions *T1*, *T2* and *T3*. Then an aspect state machine is used to model the identified crosscutting behavior. As the target state *S* has now been removed from the base state machine, we use the aspect oriented construct of *Introduction* to add a new modeling element, the state *S*, in the aspect state machine. The *selectedSates* pointcut selects the source states at which the transitions to *S* are to be added in woven model. The transitions *T1*, *T2* and *T3* are removed from the base state machine, leaving a clean base state machine with possibly less transitions and states.

Note that in this case the generated OCL queries are much simpler as compared to a forward engineering approach reported in [9] as we already have the required information about the target states.

IV. Automation of Refactoring Rules using ReState

Fig. 3 shows the architecture diagram of *ReState* Tool. The tool is implemented in Java. Use of Java as the implementation language allows us to use the extensive support for manipulating meta-models available through Java EMF [13]. Another possibility was to implement the transformation in a model transformation language. However, EMF was preferred since it provides a mature underlying infrastructure. *ReState* takes the UML state machine, the UML meta-model and the AspectSM profile as input and produces one or more ASMs and a clean base state machine as output.

The input state machine is read from .uml file using the *ModelLoader*, which internally uses the Java EMF library. The loaded model is passed to the *HeuristicsEngine* that tries to find matches against the proposed heuristics. The ReState tool uses matching rules written in java, translated from the OCL rules manually by the developers. The various proposed refactoring rules are stored in the *RuleRepository*. Once a match is found, the *ModelTransformer* applies corresponding transformation. In case of multiple matches, the designer is allowed to select a given refactoring. In case of multiple matches the decision is left to the designers.

The *ModelTransformer* applies the transformation rules and produces a clean state machine and uses the *AspectGenerator* component to generate aspects. Each identified match is modeled as a separate aspect state machine. The *AspectGenerator* contains the implementation to produce aspect state machines using AspectSM notation. This component can be further extended to support other notations in future.

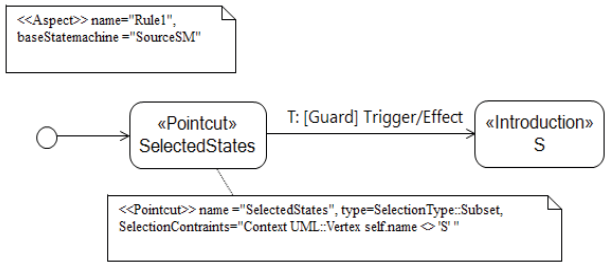

Fig. 2. Aspect state machine resulting from rule 1

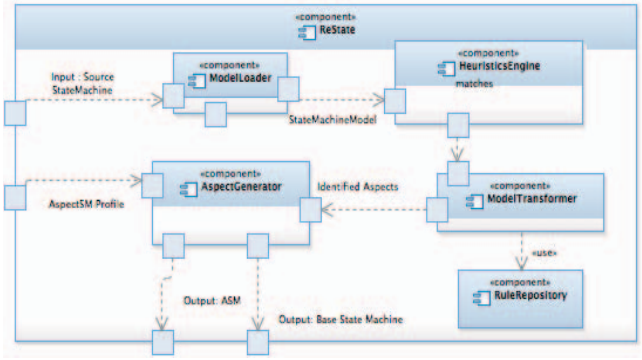

Fig. 3. Architecture of ReState Tool

V. Results of Automated Refactoring

To assess the proposed heuristics, for Saturn, we wove five-aspect state machines corresponding to each of the robustness

behavior into a base state machines (functional state machines of Saturn subsystems) and obtained five woven state machines. Each state machine contains (one) robustness behavior in addition to the elements of the base state machine. We applied our heuristics to each of these woven state machines to assess that if crosscutting behaviors can be detected and refactored as aspect state machines. The results of applying the heuristics revealed that for the first four robustness behaviors we were able to successfully detect and refactor crosscutting behaviors. These four robustness behaviors were identified with rules 2a and 4. The fifth crosscutting behavior wasn't detected by any heuristics. A summary of the results appears in Table 2.

For the ABRS, when we apply our heuristics on the state machine, one state (the Error State) and its incoming transitions were identified as a crosscutting behavior. The results are summarized in Table 2 and Table 3. As can be seen from the tables, only one aspect was identified from the four published state machines for ABRS system. Manual analysis of the state machines shows that there was no other crosscutting or redundant behavior in the state machines.

TABLE II. RESULTS OF REFACTORING INDUSTRIAL CASE STUDIES

Rule	VCS	ABRS
1. a	-	1
2. a	2	-
4	2	-
Unidentified	1	-

TABLE III. RESULTS OF REFACTORING INDUSTRIAL CASE STUDIES

Model Elements	VCS (Average of 5 behaviors)			ABRS		
	Before	*After*	*% Saving*	*Before*	*After*	*% Saving*
States	212	8	96	19	18	6
Transitions	356	6	98	31	26	16
Triggers	2686	31	99	-	-	
Mean	-	-	97	-	-	11

Table 3 presents the modifications made in the both case studies. The *% saving* column shows the percentage of model elements reduced as a result of refactoring. For VCS, we saved on average 97% of model elements, whereas for ABRS we saved on average 11%. The results showed that our approach is quite effective in refactoring crosscutting behaviors.

VI. CONCLUSION

Aspect oriented modeling offers a number of benefits such as increased reuse, ease of maintenance and enhanced modularity. A number of approaches have been proposed to bring aspect orientation to various design models including UML state machines. However, aspect oriented modeling is relatively new and is not yet widely practiced in industry. Just like legacy code, there are a large number of projects with models that do not incorporate separation of concerns and consequently can benefit from refactoring's of their models.

Based on our experiences in modeling industrial case studies, we have presented an approach to refactor existing UML state machines; we split the state machines in a base state machine and one or more aspect state machines modeling crosscutting behavior. We propose heuristics for identifying crosscutting behavior in legacy state machines and provide a set of refactoring rules. Our refactoring rules are applied as guidelines and our tool gives the designer the final say on the refactoring's. For modeling cross cutting behavior, we rely on AspectSM profile. We present the results of applying our approach on the two industrial case studies, developed as part of our previous work on environmental modeling and robustness modeling. Our automated tool was successfully able to identify the five crosscutting behaviors from the state machines of the two-selected case study. The results show that such refactoring's can significantly simplify existing models. One current limitation with the approach is considering the priority of refactoring rules; we have not yet addressed the issue of rule precedence when situations requiring multiple refactoring arise. Our future work will empirically evaluate the effects of rule precedence.

REFERENCES

[1] OMG, "Unified Modeling Language, Superstructure Specification, Version 2.4," ed: Object Management Group Inc., 2011.

[2] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand, "Generating Test Data from OCL Constraints with Search Techniques," *IEEE Trans. Softw. Eng.,* vol. 39(10), pp. 1376–1402, 2013.

[3] A. Arcuri, M. Iqbal, and L. Briand, "Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing," in *Testing Software and Systems, LNCS*. vol. 6435, A. Petrenko, A. Simão, and J. Maldonado, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 95–110.

[4] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*: Morgan Kaufmann, 2010.

[5] R. Lefticaru and F. Ipate, "Functional search-based testing from state machines," 2008, pp. 525-528.

[6] M. Z. Iqbal, A. Arcuri, and L. Briand, "Environment modeling and simulation for automated testing of soft real-time embedded software," *Software & Systems Modeling,* pp. 1–42, 2013.

[7] H. Gomaa, "Designing software product lines with UML," in *2012 35th Annual IEEE Software Engineering Workshop*, 2012, pp. 160-216.

[8] B. Selic, "Using UML for modeling complex real-time systems," in *Languages, Compilers, and Tools for Embedded Systems*, 1998, pp. 250-260.

[9] S. Ali, L. C. Briand, and H. Hemmati, "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems," *Software and Systems Modeling,* vol. 11(4), pp. 633–670, 2012.

[10] S. Ali, L. Briand, A. Arcuri, and S. Walawege, "An Industrial Application of Robustness Testing using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms," in *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models 2011), LNCS*. vol. 6981, J. Whittle, T. Clark, and T. Kühne, Eds., ed: Springer Berlin Heidelberg, 2011, pp. 108–122.

[11] M. Z. Iqbal, A. Arcuri, and L. Briand, "Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies," in *Model Driven Engineering Languages and Systems, LNCS*. vol. 6394, D. Petriu, N. Rouquette, and Ø. Haugen, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 286–300.

[12] OMG, "Object Constraint Language Specification, Version 2.0," ed: Object Management Group Inc., 2006.

[13] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*: Pearson Education, 2008.