Mining of Comprehensible State Machine Models for Embedded Software Comprehension

Wasim Said, Jochen Quante

Robert Bosch GmbH, Corporate Research
Renningen, Germany

{Wasim.Said, Jochen.Quante}@de.bosch.com

1 Introduction

Embedded legacy software contains a lot of expert knowledge that has been cumulated over many years. Therefore, it usually provides highly valuable and indispensable functionality. At the same time, it becomes more and more complex to understand and maintain. Mining of understandable models, such as state machines, from such software can greatly support developers in maintenance, evolution and reengineering tasks. Developers need to understand the software in order to evolve it. Existing state machine mining approaches are based on symbolic execution, which means enumeration of all paths. This quickly leads to path explosion problem. One effect of this problem on state machine mining is that the extracted models contain a very high number of states and transitions, and therefore are not useful for human comprehension. This means that additional measures towards comprehensibility of extracted state machines are required. To reach this goal, we introduced user interaction measures that can reduce the complexity of extracted state machines by reducing the number of states and transitions [2]. However, the complexity of boolean expressions that constitute the guards of state machines remains high. Therefore, we also presented an approach for complexity reduction of these guards to be understood by humans [4]. In this paper, we give an overview of these approaches. Also, we report on our controlled experiments which show that the approaches are highly effective in making extracted state machines understandable, and these understandable models in turn do help in comprehension of complex legacy software.

2 User interaction measures

To reduce the number of states and transitions in the mined state machines, we defined user interaction measures that enable the user to select only the relevant information in code and abstract away everything irrelevant [2]. For example, the user can select a subset of state variables and extract a model of them instead of having a model for all state variables in code. The user can also select the interesting states of these variables or define new states and then get a model that describes the function's behaviour with respect to these states. In addition, the user can add constraints on the variables to have a state machine model under a specific scenario.

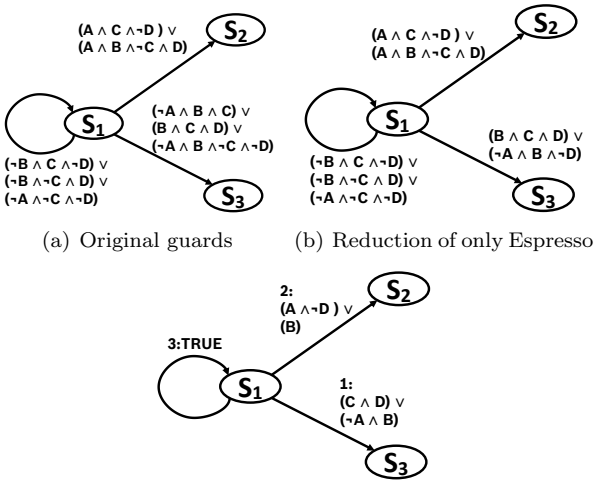
3 Reduction of guards' complexity

A transition condition (guard) is defined as the condition that must be fulfilled to trigger a transition between two states. Based on path enumeration, our guards are formed by the disjunction of all path conditions of the feasible paths between two states. Therefore, the guards of the extracted state machines from real-world systems are very complex. Moreover, using the above interaction measures to reduce the number of states and transitions makes those guards even more complex: The information that was previously contained in the state invariants is not lost, but it is moved into the guards. Our goal is to provide comprehensible state machines for human experts, and this certainly includes comprehensible guards. For this reason, we first applied two standard techniques for boolean reduction to reduce the guards complexity [1], which are binary decision diagrams (BDDs) and heuristic based logic minimization. Both approaches led to a significant reduction of guards complexity, but the reduced guards were still not comprehensible for humans in too many cases.

Therefore, we conducted a case study with industrial developers to check the human comprehensibility limits of boolean expressions in disjunctive normal form [1], as our guards have this form. The result was that the comprehensibility limits lie between five and eight conjunctions (expressions connected with OR operator). According to these limits, about 50% of our reduced guards were still not understandable for humans.

We thus developed another approach for reducing the guards' complexity. The approach is based two main observations:

1. The conditions explicitly exclude infeasible paths, which makes them more complex than necessary. We remove this complexity by masking infeasible paths from the conditions. This is done by adding infeasible path conditions to guards in cases when it helps to reduce the guard's complexity. The infeasible path can never be taken, anyway.



(a) Original guards (b) Reduction of only Espresso

(c) Reduction of Espresso with transition priorities

Figure 1: The effect of using transition priorities on guards understandability [4].

2. The expressions repeat over and over in the conditions, as they are the result of aggregating conditions during symbolic execution due to nested or sequential program structures. We deal with this issue by introducing transition priorities and exploiting them to simplify conditions.

Our approach combines heuristic logic minimization with transition priorities [4]. Transition priorities indicate the order in which the transitions' guards have to be evaluated, so that transitions that emerge from a common source state each get a distinct priority number. The transition condition with the lowest number is evaluated first (i. e., priority one is the highest). Only when it does not hold, the transition with the next higher number is checked. Our algorithm determines the most effective order of priorities, while additionally leveraging infeasible paths as "don't care" combinations.

Figure 1 presents an example of reducing the original guards (Fig. 1(a)) – once with only Espresso (a popular heuristic logic minimizer, Fig. 1(b)), and then with applying our combination of Espresso and transition priorities (Fig. 1(c)). It can clearly be seen that the reduced guards with transition priorities are more understandable than those without priorities. Our case study on the reduction of 151 guards with and without transition priorities showed that our approach (with transition priorities) achieves 99% reduction on the number of conjunctions and makes 91% of the reduced guards understandable, whereas using Espresso alone (without priorities) makes only 47% of the same guards understandable for humans.

4 Guards' comprehensibility

We evaluated the proposed approach of reducing guards complexity by conducting a controlled experiment with 24 participants. The goal was to check whether the participants can answer questions about real guards faster and with fewer errors. The subject state machines in the experiment were extracted from three embedded C functions from industrial systems. The guards were reduced twice: once with Espresso only, and then with our approach using transition priorities. The participants had to answer concrete questions to extract relevant knowledge from the guards. Then, we evaluated the results with respect to the required time and the correctness of the answers. The result showed that when the tasks were performed on transitions with priorities, 1) the processing time was always shorter on average, and 2) the number of correct answers was always higher. This indicates that guards with priorities are an adequate representation for human understanding of mined state machines.

5 The effect of mined state machines on program comprehension

For a quantitative evaluation of our interactive approach, we conducted a controlled experiment [3] to answer the research question: Do interactively extracted state machines make understanding of complex embedded code more effective? In this experiment, 30 participants had to answer comprehensibility questions about matching the given specification of two real industrial control functions with the implementation in C code. The independent variable was the availability of the extracted state machines, and the dependent variables were 1) the time needed to finish all tasks, and 2) the correctness of the results. The experiment confirmed that when extracted state machines are available, the share of correct answers increases and the required time to solve the tasks decreases significantly. This means that mined state machines do in fact help in program understanding.

6 Conclusion

Based on our controlled experiments, we can conclude that our extensions for state machine mining, namely interaction and guard reduction, make it a useful approach for realistic and helpful support for program comprehension, software reengineering and evolution.

References

[1] W. Said, J. Quante, and R. Koschke. On state machine mining from embedded control software. In *Proc. of 34th ICSME*, pages 163–172, 2018.

[2] W. Said, J. Quante, and R. Koschke. Towards interactive mining of understandable state machine models from embedded software. In *Proc. of 6th MODELSWARD*, pages 117–128, 2018.

[3] W. Said, J. Quante, and R. Koschke. Do extracted state machine models help to understand embedded software? In *Proc. of 26th ICPC*, 2019.

[4] W. Said, J. Quante, and R. Koschke. Towards understandable guards of extracted state machines from embedded software. In *Proc. of 26th SANER*, 2019.