Reflexion Models for State Diagrams

Jochen Quante, Wasim Said

Robert Bosch GmbH, Corporate Research
Renningen, Germany

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

1 Introduction

State diagrams are commonly used for specification or illustration of behavior. They appear on all levels, from high-level system behavior down to behavior of a single function. However, state diagrams are often implemented manually, and there is no guarantee that the implementation corresponds to the specification. Furthermore, a state machine may be implemented in the code without having a specification in form of a state diagram, and it might be that the developer was not even aware that he is implementing one. We call this kind of implementations *implicit state machines*. Our practical experience has shown that such code is very hard to comprehend. In this paper, we describe our work [2] on applying the idea of reflexion models on state diagrams.

2 Reflexion Models

Reflexion analysis was first described by Murphy et al. [1]. They proposed it to overcome the gap between high-level models and code. Their approach supports both checking adherence of code to a given architecture and reconstructing the architecture from code. The basic idea is to provide a conceptual model along with a mapping between conceptual components and source code, and then to use that mapping to lift dependencies from the code (automatically extracted) to the conceptual level. The existing dependencies can then be compared to the expected or allowed dependencies from the conceptual model, which leads to a classification as *convergence* (code fits model), *divergence* (unexpected dependency in code), or *absence* (missing dependency in code). The resulting model is called *reflexion model*.

3 Application to State Diagrams

The original reflexion analysis check is done by static code analysis, which immediately delivers existing dependencies (variable access, function calls, type usage, etc.). To apply this idea to state diagrams, we need a way to determine states and transitions from the code. Similar to the architecture case, we rely on the user to provide a model of the intended state diagram. The mapping is done differently: States are characterized by their invariant, and transitions are described by their transition condition. Both conditions have to be expressed in terms of expressions on variables of the program.

Extraction of existing transitions is a bit more tricky. The question is which other states can be reached from a given state, i. e., assuming that a certain invariant holds before our function is executed, which other states can we possibly be in when the function returns? This is a question that can be answered using bounded model checking.

4 Bounded Model Checking

A bounded model checker (BMC) can check whether a certain condition ("assertion") always holds at a specific point in the program – up to a certain number of loop iterations (the bound). If it doesn't hold, BMC can provide a counter example that illustrates a violation. It can also do that check under additional assumptions. We use this capability to do the required transition existence check: We give the start state's invariant as assumption *before* invocation of our subject function and let a BMC tool check whether the negated target state invariant *always* holds *after* invocation of the function. The transition is only possible if the negated target state invariant does *not* always hold, as this means that there are cases where the target state invariant holds.

If we also want to check transition conditions, we can provide the transition condition as additional assumption and then check whether the target state can still be reached. However, some additional considerations are required for this check, as the same condition may also lead to other states than the target state, and there may be other conditions that lead to the same target state [2].

5 State Diagram Reflexion Model

The edges of the reflexion model can be calculated as follows (for checking existence of transitions only):

$$RM(s,t) := \begin{cases} \text{convergence} & \Delta_{spec}(s,t) \wedge \Delta_{code}(s,t) \\ \text{divergence} & \neg\Delta_{spec}(s,t) \wedge \Delta_{code}(s,t) \\ \text{absence} & \Delta_{spec}(s,t) \wedge \neg\Delta_{code}(s,t) \\ \epsilon & \neg\Delta_{spec}(s,t) \wedge \neg\Delta_{code}(s,t) \end{cases}$$

Δ_{spec} is the transition information from the conceptual model, which is specified by source and tar-

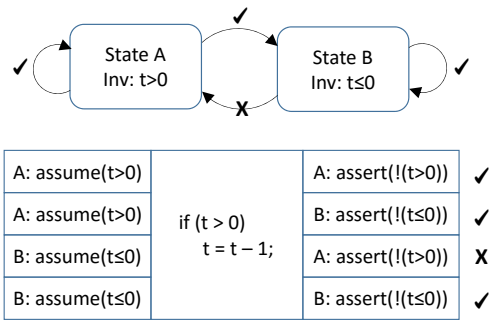

Figure 1: Example: Assumptions and assertions per state combination, BMC result and resulting reflexion state diagram for the given code fragment.

get state of each transition. Δ_{code} is the transition information as derived from the code, based on the possible sequences of invariants. For ϵ, there is no reflexion edge. *RM* gives feedback about the *existence* of a transition between two states. A similar computation is possible with consideration of transition conditions [2]. In this case, we additionally introduce *partial convergence* edges, which indicate that the given transition condition is necessary, but not sufficient.

6 Example

Figure 1 shows an example for application of our approach using BMC[1]. The code in the middle is our subject function, and we identified the state invariants $t > 0$ and $t \leq 0$. The invariants of all possible state pairs are then fed to BMC as shown in the table (one row per combination), and the respective BMC result is shown in the rightmost column. For example, the transition from state A to state B is checked by assuming A's invariant ($t > 0$) *before* and asserting B's negated invariant ($t > 0$) *after* execution of the code in question. As there are cases when ($t > 0$) is true before executing the code but is not true after execution (i. e., $t = 1$), BMC returns false – and thus we know that a transition is possible. When BMC returns true, we know that there is no variable assignment that fulfills the assumption and leads to the target state invariant – and thus there is no transition. We do the same analysis for all other state pairs. The BMC results can then be transferred to the diagram.

7 Use Cases

This approach enables a number of different use cases that we describe in the following. Firstly, it can be used for program understanding. When the developer has an initial idea which states are relevant, he can specify the state invariants and let the approach determine which transitions are possible between these states. Based on the feedback, the invariants can then be refined to identify interesting states. If he even has an idea about transition conditions, he can also add those and check if his expectations are matched. This approach allows the developer to do superficial code reading, then formulate a hypothesis and check it against the code. This should be much more efficient than getting a detailed understanding of the code based on intensive code reading alone.

A second use case is the elaboration of the relevant state space. Often, state is encoded into multiple variables – but only certain combinations of variable states are relevant. For example, the state may be coded into a bit field, but only one bit can be set at one point in time. This is something that can easily be found out by our technique: We provide all possible states and let the tool check which of these are reachable.

Another use case is consistency check of state diagrams with actual code. Given the specification state diagram, the tool can find out which transitions really exist and whether their transition conditions are implemented as specified. This ensures long-term maintainability, as the code keeps in sync with the models.

8 Conclusion

We introduced the idea of adopting reflexion models to interactively extract state machine models from code or check conformance of code with design-level state machine models. Our approach can greatly support developers in different activities, such as program understanding, conformance checking and software verification. Also, the approach is applicable to real-world software – it scales remarkably well, with the restriction that loop iterations must be bounded. It is especially well-suited and relevant for embedded control software, where most functions have state due to discretization of continuous processes. We have successfully applied the approach to several functions from the automotive domain.

In summary, using the CBMC model checker to generate reflexion models for state machine mining appears to be a very promising approach that is capable of providing useful support for software maintenance.

References

[1] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. *SIGSOFT Software Engineering Notes*, 20(4):18–28, Oct. 1995.

[2] W. Said, J. Quante, and R. Koschke. Reflexion models for state machine extraction and verification. In *Proc. of 34th Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 149–159, 2018.

[1] We use CBMC: `http://www.cprover.org/cbmc/`