Interactive Model Repair by Synthesis

Joshua Schmidt, Sebastian Krings and Michael Leuschel

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
joshua.schmidt@uni-duesseldorf.de
{krings,leuschel}@cs.uni-duesseldorf.de

Abstract. When using B or Event-B for formal specifications, model checking is often used to detect errors such as invariant violations, deadlocks or refinement errors. Errors are presented as counter-example states and traces and should help fixing the underlying bugs. We suggest automating parts of this process: Using a synthesis technique, we try to generate more permissive or restrictive guards or invariants. Furthermore, synthesized actions allow to modify the behaviour of the model. All this could be done with constant user feedback, yielding an interactive debugging aid.

1 Introduction and Motivation

Writing a formal model is a complicated and time consuming task. Often, one iterates between changing a specification and proof or model checking. Once an error has been detected, the model has to be adapted to make it disappear.

To some extent, the correction phase can be automated: Using examples of positive or negative behaviour we can synthesize corrected guards, invariants or actions. Such examples can be collected during model checking or directly provided by the user.

For simplicity, we will focus on Event-B below. Our approach has been implemented for Event-B and classical B and could be extended to various other languages supported by ProB. As we do not have a user interface in place, our prototype is not yet available to the general public. We intent to ship it as a standalone tool or bundled with one of the next releases of ProB.

2 Synthesis Technique

Our synthesis approach is based on the one by Jha et. al. [7] and is implemented inside ProB [11,10] using its capabilities as a constraint solver as outlined in [9].

The main idea is the composition of program components, represented as formulas describing input and output. Each component defines a single line of the program written in three-address code: For instance, arithmetic operations can be encoded by components that map two input values i_1, i_2 to an output value o_1. In case of addition, a constraint would ensure that $o_1 = i_1 + i_2$ holds.

By setting up constraints for each I/O example one defines valid connections between program parameters, input and output values and components as well as in between components. In the example above, this could result in connecting o_1 to the input of another component in order to synthesize more involved operations.

Other constraints encode the position or line of components in the code block. Additionally, well-definedness constraints are added to enforce a syntactically correct program. This includes ensuring type compatibility, i. e., we define connections between locations referring to the same type and explicitly add constraints preventing connections between differently typed ones.

Once a candidate program has been found, we search for another semantically different solution. That is, we search for a set of input values where the output of the solutions differs. We ask the user to choose amongst the solutions based on this distinguishing input. We iterate through further solutions in the same fashion. The ongoing search for distinguishing inputs provides us with additional I/O examples. Eventually this leads to a unique solution. Once it is found, we return the program synthesized so far.

The synthesis technique in [7] relies on two oracles. One is used to compute the output of synthesized events while the other is used to assert the correctness of a solution. We implement them as follows:

The *I/O oracle* used to compute output based on given input is replaced by the model checker and the user. For a given input, we use ProB to compute the matching output. Essentially, this amounts to computing a single animation step from the given input state to the output state using the synthesized event. Afterwards, we check the invariant on the output state: If it holds, we assume the state pair is correct and the target state is used as the output. Otherwise, the user has to decide: The event can be disabled on the input state, the invariant can be relaxed or a new output state can be provided.

The *validation oracle* is used to check if a synthesized event is correct. To provide it, we ask ProB to find two states s_1, s_2 such that the synthesized event connects $s_1 \mapsto s_2$ and s_2 violates the invariant. If a solution is found validation fails. If ProB finds a contradiction validation succeeds. Of course, a timeout might occur. In this case we have to rely on other validation options like the provers ml/pp of Atelier B [4] or the SMT solvers [5]. As a last resort, we can again query the user.

3 Interactive Workflow

The process as outlined in Figure 1 is guided and enforced by ProB. The workflow itself is quite mature and has been fully implemented within the system. Given that our implementation is prototypical we have not considered a user interface by now. Currently, we only provide access via the developer command-line interface of ProB.

Repair is performed successively, i. e., we loop until no error can be found anymore and the user is satisfied with the model. Each step starts with regular

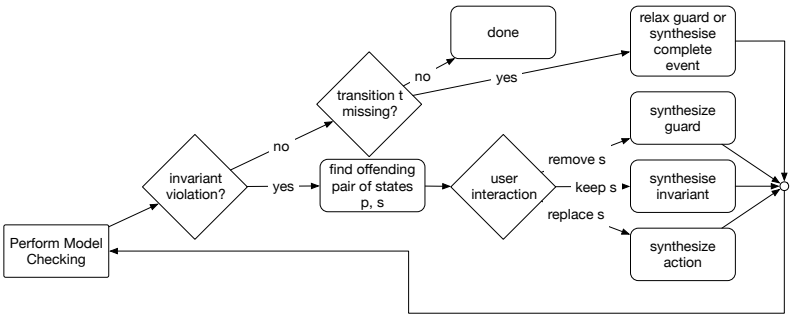

Fig. 1. Interactive Workflow

explicit-state model checking as supported by ProB. There are two possible outcomes:

- An invariant violation has been found. The user can decide to make the last transition, leading from a state satisfying the invariant to one violating it, impossible by synthesizing a stronger guard. Alternatively, the system can generate a weaker invariant or a new action in case the user edited the output state of a transition.
- The model has been checked and no error was found.[1] We query the user if a state transition is missing. In case any action is able to reach the missing state, we can synthesize a relaxed guard for the event if necessary. Otherwise, we can synthesize a new action, possibly with already appropriate guards.

4 Running Example

In Figure 2 you find the Event-B model of a simple vending machine. The machine accepts coins and gives out a can of soda per coin. There are two errors we intend to fix as outlined in Section 3:

- The invariant is violated if $soda = 0$, i. e., selling the last can is not allowed.
- *get_soda* can be executed if $coins = 0$. The guard is too permissive.

First, ProB discovers the violating state $S \triangleq coins = -1 \wedge soda = 2$. The user now has to decide, whether $coins = 0 \wedge soda = 3 \xrightarrow{get\_soda} S$ is a valid transition. We select to discard the transition resulting in a negative example. Other positive examples are collected from the state space. After a few seconds, the missing guard $coins > 0$ is synthesized.

We update the machine and model check it again. Now ProB finds $coins = 0 \wedge soda = 0$ to violate the invariant. This time we decide this state is valid, i. e., we need to change the invariant. The system now tries to find a replacement for

[1] Either the check has been exhaustive or a timeout occurred.

```
machine SimpleVendingMachine
variables soda coins
invariants
  soda > 0 & coins >= 0
events
  event INITIALISATION
    then
      soda, coins := 3, 0
  end
  event get_soda
    where
      soda > 0 & coins >= 0
    then
      soda,coins := soda - 1, coins - 1
  end
  event insert_coin
    then coins := coins + 1
  end
end
```

Fig. 2. Vending Machine

```
event SYNTHESIZED_EVENT
  where
    soda > 2 & coins > 1
  then
    soda, coins := soda - 3, -2 + coins
end
```

Fig. 3. Synthesized Event

$soda > 0$ using positive and negative examples found during model checking. The solution, $soda > -1$ is then synthesized and gets conjoined with $coins >= 0$.

After updating the machine, no further invariant violation is found. To proceed further, we can synthesize a new event. For instance, we could synthesize a "three for the price of two" event by providing the two example transitions

$$coins = 2 \wedge soda = 3 \xrightarrow{?} coins = 0 \wedge soda = 0, \text{ and}$$
$$coins = 4 \wedge soda = 4 \xrightarrow{?} coins = 2 \wedge soda = 1.$$

For these inputs ProB synthesizes the event in Figure 3. Of course there is no guarantee that proper guards will be synthesized simultaneously, as it highly depends on the given I/O examples. Another iteration might be necessary.

5 Discussion and Conclusion

Compared to other approaches like [2,1] we synthesize entirely new predicates or actions based on input and output values instead of transforming an existing specification. Synthesis has been used for repair as well, see for instance [6]. An interactive approach has been suggested in [8].

In CEGAR [3] spurious counter examples are used to refine a model checking abstraction. Our synthesis tool is guided by real counter examples and provides an interactive debugging aid for model repair. Moreover, we not only rely on the model checker to find counter-examples but also make use of ProB as a constraint-solver. This leads to more flexibility in model repair, i. e., we are able to avoid or allow specific states and even extend a machine in case model checking has been exhaustive.

We believe that an interactive modeling assistant like the one we outlined above will have its merits both for teaching as well as for professional use. First tests using our prototypical implementation seem promising.

References

1. E. Bartocci, R. Grosu, P. Katsaros, C. Ramakrishnan, and S. A. Smolka. Model repair for probabilistic systems. In *Proceedings TACAS*, LNCS 6605, pages 326–340. Springer, 2011.
2. G. Chatzieleftheriou, B. Bonakdarpour, S. Smolka, and P. Katsaros. Abstract model repair. In *Proceedings NFM*, LNCS 7226, pages 341–355. Springer, 2012.
3. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. LNCS 1855, pages 154–169. Springer, 2000.
4. ClearSy. *Atelier B, User and Reference Manuals*. Aix-en-Provence, France, 2014. Available at `http://www.atelierb.eu/`.
5. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *Proceedings ABZ*, LNCS 7316, pages 194–207. Springer, 2012.
6. T. Gvero and V. Kuncak. Interactive synthesis using free-form queries. In *Proceedings ICSE*, pages 689–692, 2015.
7. S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings ICSE*, pages 215–224, 2010.
8. E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Proceedings CAV*, LNCS 9207, pages 217–233. Springer, 2015.
9. S. Krings, J. Bendisposto, and M. Leuschel. From Failure to Proof: The ProB Disprover for B and Event-B. In *Proceedings SEFM*, LNCS 9276. Springer, 2015.
10. M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME*, LNCS 2805, pages 855–874. Springer, 2003.
11. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, Feb. 2008.