Equivalence Checking on System Level using Stepwise Induction*

Niels Thole*†
*Institute of Computer Science,
Universität Bremen, Germany
nthole@informatik.uni-bremen.de

Görschwin Fey*†
†Institute of Space Systems,
German Aerospace Center, Germany
Goerschwin.Fey@dlr.de

Abstract

We present an algorithm for equivalence checking between two C++ objects that uses stepwise induction. To prevent the effort of checking each state for reachability, we utilize a hypothesis that approximately describes the reachable states.

1. Motivation

Equivalence checking is used in the iterative development of a system to check if the old and the modified models behave equivalently. The special case of equivalence checking in our paper analyses two C++ classes. In this check the two models are equivalent iff the call of equivalent methods on both models always leads to equivalent outputs, when only equivalent methods were called before. We present an algorithm to prove or disprove the equivalence by using stepwise induction. For this proof we use a hypothesis that approximates equivalent states of the two models. Koelbl et al. [KJJP09] also use stepwise induction for equivalence checking. However, they do not further analyze a counterexample if the proof fails and therefore cannot prove that two models are not equivalent. This is possible with our algorithm. Other works on equivalence checking on C programs [GMYF09, MSF06, SBCJ05] handle specific aspects like the equivalence of similar code, array operations and pipelining. Finder et al. [FWF13] use bounded model checking to show the equivalence between two functions. The paper shows the equivalence between two functions and shortens the runtime by using checkpoints.
For the equivalence check C++ classes are interpreted as *Finite State Machines* (FSMs) where states are assignments of variables and the edges correspond to the execution of public methods. For two of those state machines exists a relation between the methods of both models specifying which methods should be equivalent to each other. The state machines are combined by using the synchronous product between them but only keeping the edges where the two inputs are in relation to each other. FSMs are less powerful than C++ and some features from C++ like pointers cannot be part of the checked classes. We also assume that all functions always terminate.

*This work has been supported by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative.

Algorithm 1 Equivalence Check

Input:
cModel1, cModel2: the C++ models;
equiv_methods: relation of equivalent methods;
hyp: hypothesis for an invariant as logical formula;
Output:
decision of equivalence and an invariant (if the models are equivalent)

Description:

1: // Look for forbidden states and remove them from the hypothesis
2: $pre\_hyp = hyp; post\_hyp =$ **true**; $forbidden\_states =$ **false**;
3: $forbidden\_states = getStatesWithPredecessors(cModel1, cModel2, equiv\_methods, pre\_hyp, post\_hyp)$;
4: **if** $(initial\_state(cModel1, cModel2) \rightarrow forbidden\_states)$
5: **return** models are not equivalent;
6: $hyp = hyp \wedge \neg forbidden\_states$;
7: // Look for counterexamples that lead from a state that fulfills the hypothesis to a state that does not fulfill it
8: $pre\_hyp = post\_hyp = hyp$;
9: **while** $(generateCounterexample(cModel1, cModel2, equiv\_methods, pre\_hyp, post\_hyp) \neq$ **null**) {
10: $(start, follow, method) = generateCounterExample(cModel1, cModel2, equiv\_methods, pre\_hyp, post\_hyp)$;
11: // Get start and its predecessors
12: $post\_hyp = start$;
13: $predecessors = getStatesWithPredecessors(cModel1, cModel2, equiv\_methods, pre\_hyp, post\_hyp)$;
14: // Check if the initial state is one of the predecessors
15: **if** $(initial\_state(cModel1, cModel2) \rightarrow predecessors)${
16: $pre\_hyp = follow$;
17: $post\_hyp =$ **true**;
18: **if** $(generateCounterexample(cModel1, cModel2, equiv\_methods, pre\_hyp, post\_hyp) \neq$ **null**) {
19: **return** models are not equivalent;
20: } **else**
21: $hyp = hyp \vee follow$;
22: } **else**
23: $hyp = hyp \wedge \neg predecessors$;
24: $pre\_hyp = post\_hyp = hyp$;
25: }
26: **return** the models are equivalent and hyp is an invariant;

Our algorithm additionally requires a hypothesis for a likely invariant. At this step, the hypothesis is not proven to be an invariant and does not need to be. If the hypothesis is not a correct invariant, it is adjusted by the algorithm.
With the hypothesis the proof of equivalence can be done by stepwise induction. This proof utilizes the generated hypothesis and proves that, if a state fulfills the hypothesis and corresponding methods are called in both models, the methods return the same results and the resulting state fulfills the hypothesis as well. For this check, an underlying model checker is used as a black-box. If the proof is successful, the equivalence of the two models is proven. However, if a counterexample is returned, it is not proven that the models are not equivalent.
We present an algorithm to handle this problem and finally prove or disprove if the two models are equivalent. Additionally, if the models are equivalent, an invariant is returned that is an overapproximation of all reachable states.

2. Our Approach

The goal of this paper is an equivalence check between two software models in C++. This check exploits a hypothesis for an invariant of the two models. The hypothesis should be true in all reachable states of the combined state machine. The hypothesis is meant to be a likely invariant. Reachable states, that do not fulfill the hypothesis, or unreachable and fulfilling states can complicate the equivalence check. Both problems are solved later on.

Algorithm 2 getStatesWithPredecessors

Input:
cModel1, cModel2: the C++ models;
equiv_methods: relation of equivalent methods;
pre_hyp, post_hyp: pre- and post-hypothesis as logical formula;
Output:
a logical formula that describes all starting states of counterexamples and their predecessors that fulfill the pre-hypothesis

Description:

1: $result = false$
2: **while** ($generateCounterexample(cModel1, cModel2, equiv\_methods, pre\_hyp, post\_hyp) \neq$ **null**) {
3: $(start, follow, method) = generateCounterExample(cModel1, cModel2, equiv\_methods, pre\_hyp, post\_hyp)$;
4: $result = result \vee start$;
5: $pre\_hyp = pre\_hyp \wedge \neg result$;
6: $post\_hyp = \neg result$;
7: }
8: **return** $result$

The proof is done with stepwise induction and either shows that when the models are in a state that fulfills the hypothesis and methods that are defined as equivalent are executed, the methods return equivalent results and the following state fulfills the hypothesis, or returns a counterexample. Counterexamples consist of an originating state $start$, a pair of methods that are defined as equivalent $method$ and a following state $follow$. A counterexample is denoted as a triple $(start, follow, method)$ which contains the according information. In the counterexample, the methods either return different results or the following state does not fulfill the hypothesis. Counterexamples are used to adjust the hypothesis until either the equivalence is proven or disproven. Algorithm 1 shows the pseudo code.
The function $generateCounterexample$ returns such a counterexample and receives the two models $cModel1$ and $cModel2$, the relation of methods $equiv\_methods$, a pre-hypothesis pre_hyp, and a post-hypothesis $post\_hyp$ who should hold in the starting and following state respectively.
Another function that is used by our algorithm is $getStatesWithPredecessors$ which returns all counterexamples and their predecessors. The inputs are identical to $generateCounterexample$. Different from $generateCounterExample$, this function returns a logical formula that describes all starting states of existing counterexamples as well as the predecessors of those counterexamples that fulfill the pre-hypothesis. The pseudo code can be seen in Algorithm 2.
In lines 1 – 3 of the algorithm, *forbidden states* are excluded from the hypothesis. We define forbidden states as states in which a call of methods defined as equivalent returns different results and the predecessors of those states.
By setting the post-hypothesis to $true$ any counterexample is generated due to different results from the methods. For each forbidden state found, pre- and post-hypothesis are adjusted to exclude that state. This enables us to find the predecessors of forbidden states and will prevent the generation of a counterexample that has already been found. In line 4, it is checked if the initial state is a forbidden state. If this is true, the models are not equivalent and the algorithm terminates. Afterwards, the hypothesis is advanced to exclude all discovered forbidden states. In the next step in lines 8 – 25 we generate counterexamples where the following state does not fulfill the hypothesis. If there is such a counterexample $(start, follow, method)$ then $follow$ does not fulfill the hypothesis since $start$ cannot be a forbidden state.
We handle the counterexample similar to the previous step where we discovered the forbidden states and exclude $start$ and all its predecessors from the hypothesis, which can be seen in lines 12 – 13. If the initial state is not excluded that way, the counterexample was not reachable and $start$

and its predecessors can safely be excluded from the hypothesis in line 23. However, if the initial state is excluded the state $start$ and therefore the original counterexample is reachable. The counterexample was generated due to the state $follow$ not fulfilling the hypothesis but not due to different results from the methods. If $follow$ is forbidden, which is checked in lines 16 – 18, the models are not equivalent which is returned in line 19. Otherwise in line 21 $follow$ is included into the hypothesis. All successors of $follow$ are checked in the same way. This is repeated until no counterexamples remain.
If there was no counterexample that disproved the hypothesis, the models are equivalent. Additionally, the final hypothesis is an invariant which can be used to speed up further equivalence checks after the modification of one model or other checks.

3. Experimental Results

Experiments were done with CBMC[CKL04] as backend. A simple example was tested with multiple hypotheses. The check took only a few seconds with good hypotheses while $true$ lead to a timeout.

References

[CKL04] Clarke, E., D. Kroening, and F. Lerda: *A tool for checking ANSI-C programs*. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[FWF13] Finder, A., J. Witte, and G. Fey: *Debugging HDL designs based on functional equivalences with high-level specifications*. IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, pages 60–65, 2013.

[GMYF09] Gao, S., T. Matsumoto, H. Yoshida, and M. Fujita: *Equivalence checking of loops before and after pipelining by applying symbolic simulation and induction*. In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies*, pages 380–385, 2009.

[KJJP09] Koelbl, A., R. Jacoby, H. Jain, and C. Pixley: *Solver technology for system-level to RTL equivalence checking*. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 196–201, 2009.

[MSF06] Matsumoto, T., H. Saito, and M. Fujita: *Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs*. In *Proceedings of the International Symposium on Quality Electronic Design*, pages 370–375, 2006.

[SBCJ05] Shashidhar, K., M. Bruynooghe, F. Catthoor, and G. Janssens: *Functional equivalence checking for verification of algebraic transformations on array-intensive source code*. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1310–1315, 2005.