Equivalence Checking on System Level using A Priori Knowledge

Niels Thole[1,2] Heinz Riener[1] Goerschwin Fey[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Institute of Space Systems, German Aerospace Center, 28359 Bremen, Germany
{nthole,hriener,fey}@informatik.uni-bremen.de

***Abstract*—Equivalence checking is applied when a system description is refined iteratively to reduce the manual effort required to check the consistency before and after modifications. We present a novel functional equivalence checking algorithm which is especially designed to verify equivalence of two hardware descriptions on the system level. Our algorithm uses a stepwise induction proof guided by counterexamples and incorporates a priori knowledge provided by a designer to speed up reasoning. The a priori knowledge is given symbolically in form of a hypothesis, i.e., a logical formula, which approximates the set of all possible equivalence states of the two designs. The algorithm stepwisely refines the hypothesis until either a counterexample has been found disproving equivalence or the hypothesis overapproximating all equivalence states. Preliminary experiments for two case studies, a scalable parallel counter and a processor model, show the applicability of our approach in practice.**

I. Introduction

State-of-the-art design flows for electronic systems start with the development of an abstract, high-level description of the system in a programming language like Java or C++. The high-level description serves as an executable specification that captures the functional behavior of the system, but neglects low-level details like timing or power consumption. After thorough functional testing and verification, the executable specification is step-wisely refined until finally a hardware description on register-transfer level is obtained. These stepwise refinements typically correspond to a lengthy iterative process, where parts of the description are manually changed to reduce the abstraction in each iteration. The introduced manual changes, however, bypass the initial testing process and may introduce new subtle bugs such that additional functional verification becomes necessary.

Formal methods like functional equivalence checking allow for automation: the hardware descriptions before and after refinement are checked for functional equivalence such that the introduced changes are guaranteed to not affect the correctness of the descriptions. Existing equivalence checking approaches, however, do not scale well to complex system level designs.

In this paper, we propose a novel functional equivalence checking algorithm which is especially designed to verify the equivalence of two hardware descriptions on the system level. We use a coarse-grained definition of functional equivalence, i.e., two hardware descriptions D_1 and D_2 are functionally equivalent *if and only if* (iff) D_1 and D_2 produce the same sequence of outputs when from corresponding input states a sequence of corresponding operations has been executed. An additional correspondence mapping has to be provided for the two descriptions allowing for different public interfaces.

Our algorithm uses a stepwise induction proof guided by counterexamples and incorporates a priori knowledge provided by a designer to speed up reasoning. The a priori knowledge is given symbolically in form of a hypothesis, i.e., a logical formula, which approximates the set of all possible equivalent states of the two hardware descriptions. Our algorithm can handle hypotheses that neither precisely underapproximate nor overapproximate the set of reachable states. During reasoning, the initial hypothesis is adapted when new equivalent states are discovered or states contained in the hypothesis are proven non-equivalent. Eventually, either the hypothesis becomes an invariant strictly overapproximating all equivalent states or a counterexample is found proving non-equivalence of the two hardware descriptions. A good initial choice, i.e., a hypothesis that describes "almost" the set of all possible equivalent states, significantly speeds up the performance of equivalence checking. Experimental results indicate that our approach enables functional equivalence checking for complex hardware designs in short time when the initial hypothesis is chosen well.

Previous work on equivalence checking of high-level hardware descriptions written in the programming languages C and C++, e.g., [7, 6, 4, 3], focuses on similar source code or concerns modifications of arrays. These approaches typically use much "finer" definitions of functional equivalence such that they do not scale well to larger designs. Our approach targets complex system level designs.

The strength of the underlying algorithm for equivalence checking lies in the combination of induction and model checking. Kölbl et al. [5] also use an induction proof, however, in contrast to our approach their proof is not guided by counterexamples such that proving non-equivalence of two hardware descriptions becomes expensive. In the worst case k-induction [8] requires to unfold the transition relation to the diameter of the graph which is too large for designs used in practice. Our algorithm similarly to *Property Directed Reachability* (PDR) [2] step-wisely refines a candidate set of reachable states. While PDR always uses an overapproximation of the reachable states, our algorithm starts from an initial approximation of the equivalent states which does not need to be an over- or underapproximation. This allows to incorporate a priori knowledge of the designer and leads to a drastic performance boost when the initial approximation is chosen wisely.

We have implemented our algorithm for functional equivalence checking of system level hardware designs modeled

This work was supported by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative, and the German Research Foundation (DFG, grant no. FE 797/6-1).

in the programming language C++. Each hardware module corresponds to a C++ class. The set of all member variables of a class defines the state of the module, whereas public methods define terminating operations that can be executed to change the state. Our algorithm verifies whether two given modules are functionally equivalent under a given correspondence mapping allowing that the public interfaces of the two classes are different.

In summary, we contribute an algorithm that

1) checks the equivalence of two system level models,
2) exploits a hypothesis that contains a priori knowledge and does not need to be an under- or overapproximation.

The remainder of the paper is structured as follows: in Sec. II, we provide definitions. Sec. III describes our algorithm and Sec. IV presents preliminary experimental results for two case studies, a scalable parallel counter and a processor model. Sec. V concludes the paper.

II. Preliminaries

In this paper, the behavior of a hardware module is described as finite state machine. The exact definition of these finite state machines depends on the C++ class that models the hardware module.

Definition 1. A *Mealy transducer* $M = (S, S_0, X, Y, \phi, \psi)$ is a tuple, where S is a finite set of states, $S_0 \subseteq S$ is the finite subset of initial states, X is a finite set of inputs, Y is a finite set of outputs, $\phi : S \times X \to S$ is a function that determines the next state depending on the current state and the input and $\psi : S \times X \to Y$ is a function that determines the output depending on the current state and the input.

Suppose that $M = (S, S_0, X, Y, \phi, \psi)$ is a Mealy transducer. Every sequence $(i_1, i_2, \ldots, i_n)$ of inputs on M corresponds to a sequence $(s_0, s_1, \ldots, s_n)$ of states such that $s_0 \in S_0$ and $s_j = \phi(s_{j-1}, i_j)$ for all $1 \leq j \leq n$.

When we describe a C++ class as Mealy transducer, the set of states S is defined as $S = \mathit{Var}_1 \times \mathit{Var}_2 \times \cdots \times \mathit{Var}_n$, where n is the number of member variables and each set Var_i contains all possible assignments for the i-th member variable. The initial state s_0 is defined as the variable assignment of the class after calling the constructor and consequently $S_0 = \{s_0\}$. For the sake of simplicity, we currently use only one initial state. Multiple initial states would require overhead for defining which initial states of two models correspond to each other which could easily be implemented. The set of inputs X contains a label for every possible call of a public method f with every possible argument *arg*, i.e, X contains all elements $f(\mathit{arg})$ where f is a function of the C++ class and *arg* is a valid argument of f. As such, each possible argument for a function corresponds to one element in X. The set Y contains all possible outputs of all public methods. For *void*-methods the return value $\perp$ is used. The function $\phi(s, f(\mathit{args}))$ returns the assignments of variables of the C++ object after $f(\mathit{args})$ was called on the assignment s. Similarly, $\psi(s, f(\mathit{args}))$ is the return value of $f(\mathit{args})$ on the assignment s.

The used description leads to some restrictions to the C++ class. The finite number of states forbids mechanisms that lead to an infinite number of states. Furthermore, all methods need to terminate. Due to the halting problem, equivalence checking is not decidable if it is not known if methods terminate. On the abstract implementation of hardware systems these are common restrictions.

Two Mealy transducers are combined in a product machine. The product machine describes the behavior of the two C++ classes when running equivalent methods simultaneously.

Definition 2. Given two Mealy transducers $M_1 = (S_1, S_{0_1}, X_1, Y_1, \phi_1, \psi_1)$ and $M_2 = (S_2, S_{0_2}, X_2, Y_2, \phi_2, \psi_2)$ that describe C++ classes and a relation $\mathit{EqMeth} \subseteq X_1 \times X_2$ which contains the information about methods that should be equivalent, the *product machine* $M_c = (S_c, S_{0_c}, X_c, Y_c, \phi_c, \psi_c)$ is defined in the following way:

- $S_c = S_1 \times S_2$
- $S_{0_c} = S_{0_1} \times S_{0_2}$
- $X_c = \mathit{EqMeth}$
- $Y_c = Y_1 \times Y_2$
- $\phi_c : S_c \times X_c \to S_c$ with $\phi_c((s_1, s_2), (f(\mathit{args})_1, f(\mathit{args})_2)) = (\phi_1(s_1, f(\mathit{args})_1), \phi_2(s_2, f(\mathit{args})_2))$
- $\psi_c : S_c \times X_c \to Y_c$ with $\psi_c((s_1, s_2), (f(\mathit{args})_1, f(\mathit{args})_2)) = (\psi_1(s_1, f(\mathit{args})_1), \psi_2(s_2, f(\mathit{args})_2))$

This model differs from the usual product machine which uses all possible combinations of inputs from both state machines instead of restricting itself to a subset.

Definition 3. Two models of C++ classes $M_1 = (S_1, S_{0_1}, X_1, Y_1, \phi_1, \psi_1)$ and $M_2 = (S_2, S_{0_2}, X_2, Y_2, \phi_2, \psi_2)$ are defined as equivalent under the relation $\mathit{EqMeth} \subseteq X_1 \times X_2$ iff for all traces of inputs on the product machine $(m_1, m_2, \ldots, m_n) \in \mathit{EqMeth}^n$ with the corresponding path $(s_0, s_1, \ldots, s_n)$ the outputs of the two models are identical for every method call, i.e.,

$$\forall i \in \{1, 2, \ldots, n\} \exists y \in Y_1 \cap Y_2 : \psi(s_{i-1}, m_i) = (y, y)$$

When the two elements of the output are equivalent for all functions in a state s, it is called *equivalent*. All other states are called *non-equivalent*.

For our proof we try to avoid extensive checks for reachability and utilize a hypothesis that should separate the reachable states from the non-reachable ones.

Definition 4. A *hypothesis hyp* is a Boolean logic formula over the variables of the product machine. The formula *hyp* defines a set $\mathit{Hyp} \subseteq S_c$ which contains all states $s \in S$ that fulfill the formula *hyp*.

We call a hypothesis *hyp* with the corresponding set *Hyp optimal*, if it is sufficient to show the equivalence with a single inductive step, i.e.,

1) $s_0 \in \mathit{Hyp}$
2) $\forall s \in \mathit{Hyp}, f \in X_c : s \textit{ is equivalent} \wedge \phi_c(s, m) \in \mathit{Hyp}$

Our algorithm generates counterexamples and uses them to refine the hypothesis.

Definition 5. A *counterexample* is a triple $(s_{\mathit{start}}, s_{\mathit{follow}}, m)$ and depends on a pre- and a post-hypothesis $\mathit{hyp}_{\mathit{pre}}$ and $\mathit{hyp}_{\mathit{post}}$ as well as on a product machine $M_c = (S_c, S_{0_c}, X_c, Y_c, \phi_c, \psi_c)$. A counterexample describes a starting state $s_{\mathit{start}} \in S_c$ that fulfills $\mathit{hyp}_{\mathit{pre}}$. When the method $m \in X_c$ is called in s_{start} the state s_{follow} is reached, i.e., $\phi_c(s_{\mathit{start}}, m) = s_{\mathit{follow}}$. In the counterexample the method returns non-equivalent output, i.e., $\exists y_1 \in Y_1, y_2 \in Y_2 : y_1 \neq y_2 \wedge \psi_c(s_{\mathit{start}}, m) = (y_1, y_2)$, or the following state does not fulfill the post-hypothesis.

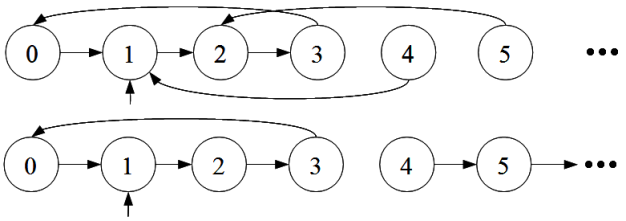

Fig. 1: Two models for counters

III. Our algorithm

We present an algorithm that does an equivalence check between two high-level models of a hardware system described in C++. This section is introduced by an example that presents the used models and shows an optimal hypothesis for them. In Section III-A, we will present our used algorithm. Sections III-B, III-C, and III-D present the functions used by our algorithm.

Example 1. We consider two implementations of counters and apply equivalence checking. The counters have a single variable *counter* of the type integer and the public method `countUp`.

In the first implementation the method `countUp` uses the modulo operation, i.e., `counter = (counter + 1) % 4`. The method returns the new value of *counter*. This counter is interpreted as a state machine $M_{mod} = (S, S_0, X, Y_{mod}, \phi_{mod}, \psi_{mod})$. The counter has one variable and its states correspond to all possible assignments for this variable, i.e., S = *Int*, where *Int* corresponds to all possible assignments for an integer variable. This is not constrained to all reachable values but all values that are possible for integer. The method `countUp` has no additional arguments and is the only possible input for the state machine, i.e., $X = \{\texttt{countUp}\}$. The only method returns integer values between 0 and 3 since the modulo operation is used, i.e, $Y_{mod} = \{0, 1, 2, 3\}$. The method ϕ_{mod} returns the next state. Since `countUp` is the only possible input, the next state corresponds to its execution, i.e.,

$$\phi_{mod}(counter, \texttt{countUp}) = (counter + 1)\%4$$

The output function ψ_{mod} returns the new value of counter and is by definition identical to ϕ_{mod}.

The second counter uses the if-command to reset the counter when it would reach 4, i.e.,

```
counter++; if (counter == 4) counter = 0;
```

When we interpret this counter as a state machine as well, we get $M_{if} = (S, S_0, X, Y_{if}, \phi_{if}, \psi_{if})$ where the sets of states and the alphabet of inputs are identical to the previous counter. The function can return all integer values except for 4, i.e., $Y_{if} = Int \setminus \{4\}$. The functions ϕ_{if} and ψ_{if} are different from their counterpart in the first state machine. For these functions

$$\phi_{if}(counter, \texttt{countUp}) = \begin{cases} 0 & \text{if } counter = 3 \\ counter + 1 & \text{otherwise} \end{cases}$$

and ψ_{if} is identical to ϕ_{if}. An excerpt of these counters is shown in Fig. 1. The top counter is the modulo-counter and the bottom one the if-counter. The vertices show the assignments to the variable counter and the edges represent the next-state functions. There are no labels on the edges since every edge corresponds to the function `countUp` and returns the next state as output. The initial states are marked with an additional arrow.

When we want to combine these two counters by using our product machine we require a relation that describes which methods should be equivalent to each other. The two `countUp` methods are meant to behave identically in the reachable states, i.e., $EqMeth = \{(\texttt{countUp}, \texttt{countUp})\}$. With this relation the product machine $Counter_c = (S_c, S_{0_c}, X_c, Y_c, \phi_c, \psi_c)$ can be generated. The set $S_c = S_{mod} \times S_{if} = Int \times Int$ describes all pairs of states of the two original state machines and the functions ϕ_c and ψ_c determine the next states and outputs of each original machine. An excerpt of the product machine is shown in Fig. 2. In the representation of the states, the top value corresponds to the if-counter and the bottom value corresponds to the modulo-counter. The reachable states of the product machine are equivalent, e.g., $\phi_c((2,2)) = (\phi_{mod}(2), \phi_{if}(2)) = (3,3)$. But the non-reachable states are not equivalent, e.g., $\phi_c((4,4)) = (\phi_{mod}(4), \phi_{if}(4)) = (1,5)$.

In this example the hypothesis hyp = ($if.counter$ = $mod.counter$) $\wedge\ 0 \leq mod.counter \leq 3$, where the prefix *mod* and *if* correspond to the according counters, would exactly describe all reachable states of the product machine and is an optimal hypothesis. Using this hypothesis will reduce the runtime because the algorithm will find no unreachable counterexamples.

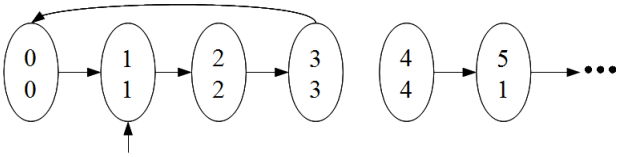

Fig. 2: An excerpt of the product machine

Algorithm 1 EquivalenceCheck

Input:
cModel1, cModel2: the C++ models;
eqMeth: relation of equivalent methods from *cModel1* and *cModel2*;
hyp: starting hypothesis, a logical formula over the variables of *cModel1* and *cModel2* that is true for the initial state;
Output:
Return "Equivalent" and an invariant if the models are equivalent and return "Non-equivalent" otherwise
Description:

```
1:  init = getInitState(cModel1, cModel2)
2:  hyp = hyp ∧ ¬getPredStates(cModel1, cModel2,
        eqMeth, hyp, true)
3:  if init → ¬hyp then return ("Non-equivalent", false)
4:  while a counterexample (s_start, s_follow, m) exists
5:    preds = getPredStates(cModel1, cModel2,
          eqMeth, hyp, ¬s_follow)
6:    if init → preds then
7:      if s_follow is a non-equivalent state
            then return ("Non-equivalent", false)
8:      else add the following states to the hypothesis and
            repeat this step for their descendants that do not
            fulfill the hypothesis
9:    else hyp = hyp ∧ ¬preds
10: return ("Equivalent", hyp)
```

A. The Algorithm `EquivalenceCheck`

Algorithm 1 shows the pseudo code of our main algorithm `EquivalenceCheck`. For our approach, we require

Algorithm 2 getPredStates

Input:
cModel1, cModel2: the C++ models;
eqMeth: 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 (generateCounterexamples(cModel1,
       cModel2, eqMeth, pre_hyp, post_hyp)) {
3:   (c_1, c_2, ..., c_n) = generateCounterexamples
         (cModel1, cModel2, eqMeth, pre_hyp, post_hyp);
4:   cex = generalize((c_1, c_2, ..., c_n),
         pre_hyp, post_hyp);
5:   result = result ∨ cex;
6:   pre_hyp = pre_hyp ∧ ¬cex;
7:   post_hyp = post_hyp ∧ ¬cex;
8: }
9: return result
```

the initial hypothesis to be true for the initial state of the product machine. An initial hypothesis that is generated by a developer should usually fulfill this requirement as well as an overapproximation like *true*.

In line 1, we create a formula that describes the initial state of the product machine. We require this formula for different checks during our algorithm.

Afterwards, non-equivalent states and their predecessors are excluded from the hypothesis in line 2. If any non-equivalent state is reachable, the two models are not equivalent. If one such state is reachable, calling equivalent methods in a reachable state would return different results. This step is realized by calling the function `getPredStates` with the starting hypothesis as pre-hypothesis and *true* as the post-hypothesis. The post-hypothesis *true* is valid for all states. This means that counterexamples due to following states that do not fulfill the hypothesis do not exist and every generated counterexample is generated due to non-equivalent output.

If the initial state is excluded from the hypothesis in this step, a non-equivalent state is reachable and the models are proved to be not equivalent which is shown in line 3.

In the following loop that starts in line 4 the remaining counterexamples are handled. Each of these counterexamples arises from a following state that does not fulfill the hypothesis because counterexamples due to non-equivalent output were already removed from the hypothesis in the previous step.

With such a counterexample $(s_{start}, s_{follow}, m)$, we generate the formula *preds* which describes all predecessors of s_{follow} that fulfill the hypothesis in line 5. This is realized by calling `getPredStates` with the current hypothesis as pre-hypothesis and the post-hypothesis $\neg s_{follow}$. Thus all states that fulfill *preds* are predecessors of s_{follow}.

If the initial state fulfills *preds*, s_{follow} is reachable but does not fulfill the hypothesis. This is checked in line 6. Since s_{follow} is not contained in the hypothesis it needs to be checked if it is a non-equivalent state. If s_{follow} is a non-equivalent state, the models are not equivalent because a non-equivalent state is reachable which is returned in line 7. Otherwise, s_{follow} is added to the hypothesis. Direct descendants of s_{follow} that do not fulfill the hypothesis are handled in the same way, i.e., checked for non-equivalence and added to the hypothesis. This process is realized in line 8.

If the initial state does not fulfill *preds*, we can safely remove *preds* from the hypothesis in line 9.

This loop from line 4 to line 9 is repeated until no counterexamples remain. When the equivalence was not disproved until the end of the loop, the models are equivalent since no counterexamples remain to invalidate the equivalence which is returned in line 10. The final hypothesis is an invariant of the product machine and can support further tests.

It is possible to prove that this algorithm always terminates and always decides correctly.

B. The Algorithm `getPredStates`

An essential function that is used by our algorithm is `getPredStates` which returns all counterexamples to the given pre- and post-hypothesis and a subset of their predecessors. The subset contains all predecessors p where a path from p to a counterexample exists that only uses states that fulfill the pre-hypothesis. The inputs are identical to `generateCounterexamples`. Different from `generateCounterexamples`, this function returns a logical formula that describes all starting states of existing counterexamples as well as a set of predecessors of those counterexamples. Algorithm 2 shows the pseudo code of `getPredStates`.

The return value *result* of `getPredStates` is initialized with *false* in line 1, which describes the empty set. The loop that starts in line 2 continues while counterexamples still exist. Whenever counterexamples to the current pre- and post-hypothesis are generated in line 3, the counterexamples are generalized in a first step by using the function `generalize` in line 4. This allows us in each step to consider a set of states instead of only a small number of states. The generalized counterexamples are added to the result in line 5 and removed from the pre- and post-hypothesis in the lines 6 and 7. The removal from the pre-hypothesis prevents the same counterexamples from triggering again while the removal from the post-hypothesis makes every direct predecessor of the current counterexamples a counterexample as well. The loop is repeated until no counterexamples remain. Finally, *result* is returned in line 9.

C. The Function `generateCounterexamples`

The function `generateCounterexamples` returns k different counterexamples and receives the two models *cModel1* and *cModel2*, the relation of methods *equiv_methods*, as well as two hypotheses, where the pre-hypothesis *pre_hyp* should hold in the starting state and the post-hypothesis *post_hyp* in the following state. The variable k is a constant. If less than k counterexamples exist, all existing counterexamples are returned. If no counterexamples exist, `null` is returned. This function merely offers an interface to the underlying model checker that generates a single counterexample.

D. The Function `generalize`

We use the function `generalize` to generalize a number of given counterexamples to describe a set of counterexamples. As additional inputs, a pre- and a post-hypothesis are given. To generalize the counterexamples, we use three approaches:

1) Check for "don't care" variables: We check for each counterexample $(s_{start}, s_{follow}, m)$ which assignments of variables are not relevant for the counterexample and remove the assignments of those "don't care" variables. This provides a set of similar counterexamples instead of a single state. We start with a formula that describes a starting state $s_{start} = \bigwedge_{i \in I}(var_i = value_i)$, where I is an index set over all variables of the two models. The formula describes the assignment of variables in the starting state. We start with a set $J := I$ and try for each element of I to remove it from J. After removing an element j from J it is tested if all states that fulfill the formula $s'_{start} = \bigwedge_{i \in J}(var_i = value_i)$ lead to a counterexample when the method m is called. If some states do not lead to a counterexample, j is inserted into J again.

2) Check for intervals: In the next step, we try to generalize the set of counterexamples even more by finding intervals for the integer variables of the provided models. For each integer variable *var* that remains in at least one counterexample after removing the "don't care" variables, we determine the values in the counterexamples as well as the upper and lower bound according to the pre-hypothesis. The upper and lower bounds are detected by looking for terms within the pre-hypothesis that limit *var* and do not need to be the optimal bounds that can be taken from the hypothesis. With these values, we generate a sorted vector $(val_1, \ldots, val_k)$ of values that are assigned to *var* in at least one counterexample or are bounds according to the pre-hypothesis. We try to decrease the upper bound val_k of *var* according to the pre-hypothesis by replacing it with the highest value of a counterexample val_{k-1}. The value val_{k-1} must be less than val_k because the starting state of a counterexample needs to fulfill the pre-hypothesis. If this is possible, we try to decrease it even further and try val_{k-2}, val_{k-3}, ... In a next step, we try to increase the lower bound analogously.

We use two approaches to verify if the value of *var* needs to remain within an interval. The first checks if all states s with $s \rightarrow pre\_hyp \wedge \neg(upper > val > lower)$ can be starting states of a counterexample, where *upper* and *lower* are the upper and lower bound of the interval. In this case, all these states s can be removed from the hypothesis and only states where *var* is within the interval remain.

The second approach checks if the value of *var* is always within the interval after calling any function when the value was in the interval before and the pre-hypothesis was valid. We also need to check if the initial value for *var* is within the interval. In this case, it is not possible to leave the interval and we can safely remove all states where *var* is not in the interval from the hypothesis.

The two different presented approaches are used due to different possible scenarios. In our example of the adders, the if-counter can be successfully checked by the first method and the modulo-counter by the second method.

Additionally, we try to shrink the intervals even further by decreasing the best detected upper bound of the interval and increasing the lower bound. For each bound, we try to shrink the interval accordingly, i.e., decrease the upper bound or increase the lower bound by using an approach similar to binary search until we find the optimal bound for *var*.

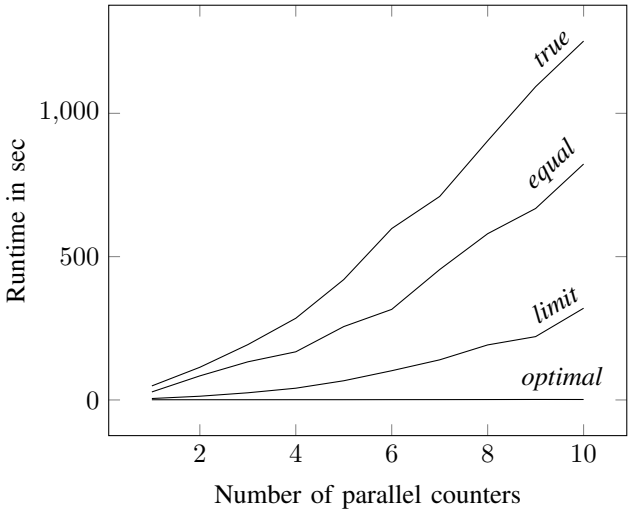

Fig. 3: Performance comparison for different hypothesis and numbers of counter variables.

3) Find equal variables: Finally, we test if some variables in the models are always equal. If a variable var_1 from the first model and a variable var_2 from the second model are different in all counterexamples, they could be required to be equal. We check for each of these pairs, if those variables are equal in the initial state. If they are equal, we verify, similarly to the second approach for intervals, that they are equal after calling any function when they were equal before and the pre-hypothesis was fulfilled. If this checks returns a positive result, we can return all states where $var_1 \neq var_2$ as counterexamples.

After all generalizations are done, we return the final set of generalized counterexamples.

IV. Experimental results

All experiments were conducted on a Lenovo T430 with an Intel Core i5-3320M CPU with 2.6GHz and 8GB of RAM. The operating system is Windows 7 32bit. For the generation of counterexamples, CBMC v4.9 [1] is used as a blackbox.

For the experiments two kinds of counters were implemented. The compared versions used the operands `modulo` or `if` to count from 0 to 9,999,999 like described in Ex. 1. Multiple counters are used in a single C++ class, so it is possible to count on n different and independent variables. The lines of code in the used classes range from 22 for the one-dimensional modulo-counter to 88 for the ten-dimensional if-counter. An optimal hypothesis *optimal* for this example consists of two parts, i.e., $optimal = equal \wedge limit$. The first part $equal = \bigwedge_{i \in \{0,1,\ldots,n\}}(if.counter_i = mod.counter_i)$ describes the equality between the two counters and $limit = \bigwedge_{i \in \{0,1,\ldots,n\}}(0 \leq mod.counter_i \leq 9999999 \wedge 0 \leq if.counter_i \leq 9999999))$ describes the bounds for the variables. The hypothesis *optimal* describes exactly all reachable states of the corresponding product machine.

Equivalence checking on counters is complicated because the paths to some reachable states are very long. In this example the equivalence check is further complicated by the fact that most non-reachable states of the product machine are non-equivalent.

The different runtimes of our algorithm depend on the used initial hypothesis and the number of parallel counters can be seen in Fig. 3. We tried the initial hypotheses *true*, *equal*, *limit*,

and *optimal*. We can see, that hypothesis *optimal* is decided almost instantly in all cases and takes from 0.7 seconds for $n = 1$ to 1.6 seconds for $n = 10$. The additional runtime due to additional counters is almost linear due to the removal of "don't care" variables. When a better hypothesis is used, the runtime is further decreased. However, we can even show equivalence in an acceptable time when we start with the hypothesis *true*.

Thus, this experiment has shown that our algorithm scales well in this example with increasing complexity of the model. The experiment also showed that the runtime highly depends on the used hypothesis. While we can show equivalence for the hypothesis *true*, the runtime increases significantly compared to an optimal hypothesis.

In our second experiment we considered a simple processor with a pipeline as a more complicated model. Our processor uses a 3-step-pipeline and has 4 registers that store 3-bit-values. There is no external memory and the processor can use 3-op-codes to add or substract the values in the registers from each other. The C++ model provides getter-functions for all registers and a `nextStep`-function, that corresponds to a single cycle of the pipeline and loads a new command as argument. The processor forbids write/read- and write/write-conflicts within the pipeline and loads a `nop`-command instead, if a loaded command would cause a conflict.

The abstract model of the processor is basically a queue. When `nextStep` is called, the processor computes the result of the third command in the pipeline and writes the result in the according register. The detailed model loads the required inputs for the first operation in the pipeline, computes the result of the operation for the second operation and writes the result of the third operation back into the according register. The variables that store the data that is required for the next step, i.e., the read operands and the computed result, are stored in special registers that do not exist in the abstract model.

First, we determine an optimal hypothesis for this equivalence check. The registers and operations in the pipeline need to be equal to their according element in the other model of the processor. All registers need to be inside their valid range, i.e., between 0 and 7, and all operations in the pipeline need to be valid. There must not be any conflicts in the pipeline. Finally, the registers for the loaded inputs and the computed result in the detailed model need to be correct according to the current registers and the operation that just loaded those values.

When we use an optimal hypothesis that contains this information, our approach can show the equivalence of the two models in 10 seconds. We can leave out some information from the hypothesis and will still get a correct result within an acceptable time. For example, when we remove the equality of the second operation in the pipeline, equivalence is shown in 1789 seconds.

However, changes to the more complex part of the hypothesis, e.g., omitting the correctness of the computed output value in the detailed model, will not be found by our heuristics for generalization and will lead to a time-out. In our experiments, we aborted this run after 12 hours.

In this second experiment, we showed that complex models cannot be handled within a feasible time when we use a simple hypothesis. However, when we use a good hypothesis, our approach can handle these complex models.

For each optimization we implemented, we provided an example in our experiments that would not be feasible without that optimization. When we check the adders and use the initial hypothesis *equal* but do not provide the check for intervals, the algorithm will time out. Similarly, when we use the hypothesis *limit* instead and do not provide the optimization for checks of equivalence, the algorithm will time out as well. The optimization of removing "don't care" variables helps when we analyze parallel counters, which could not be handled otherwise. On the other hand, when an optimization is not required for a specific hypothesis, the runtime only increases slightly. For example, when we remove the check for equality while using the initial hypothesis *equal*, the runtime decreases from 28 seconds to 25 seconds without the optimization for equality.

V. Conclusion

In this paper, we present an algorithm to prove functional equivalence of two hardware description on the system level. The presented algorithm uses a hypothesis that is stepwisely refined to approximate the set of all equivalent states of the two designs. The hypothesis allows to use the expert knowledge of a designer to speed up verification. Preliminary experimental results for two case studies, a scale parallel counter and a processor model, show that the runtime can be significantly reduced, even for complex designs, when the "right" hypothesis has been chosen.

References

[1] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

[2] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design*, pages 125–134, 2011.

[3] Alexander Finder, Jan-Philipp Witte, and Goerschwin Fey. Debugging HDL designs based on functional equivalences with high-level specifications. In *International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, pages 60–65, 2013.

[4] Shanghua Gao, Takeshi Matsumoto, Hiroaki Yoshida, and Masahiro Fujita. Equivalence checking of loops before and after pipelining by applying symbolic simulation and induction. In *Workshop on Synthesis And System Integration of Mixed Information Technologies*, pages 380–385, 2009.

[5] Alfred Kölbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to RTL equivalence checking. In *Design, Automation and Test in Europe*, pages 196–201, 2009.

[6] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In *International Symposium on Quality of Electronic Design*, pages 370–375, 2006.

[7] Kodambal. C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Functional equivalence checking for verification of algebraic transformations on array-intensive source code. In *Design, Automation and Test in Europe*, pages 1310–1315, 2005.

[8] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *International Conference on Formal Methods in Computer-Aided Design*, pages 108–125, 2000.